Sustainable Software Systems: An investigation into comparing the performance and energy consumption of software components

A thesis submitted to the School of Computing Sciences at the University of East Anglia in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Francis Kenneth Carver

March 2024

Abstract

Information and communication technology (ICT) is an increasing contributor to greenhouse gas emissions, but ICT systems are a combination of hardware and software. Hardware is the part that requires raw materials and manufacturing processes, consumes electricity, and generates heat when in use. However, in most cases, this hardware is only present in order to run application software. Changing or replacing that software can affect both the amount of hardware needed and the resources consumed during its operation. Although ICT systems have the potential to improve the sustainability of other fields, improving the sustainability of the ICT systems themselves is a vital undertaking.

Modern software is typically constructed by combining and reusing other software in the form of libraries and components. Selecting these components is a key aspect of software development. The design and construction of ICT systems is subject to conflicting economic, practical, technological, and political constraints. Historically, the environmental impact of software development choices has had a much lower priority than economic or functional factors. Software developers face a confusing array of choices and a lack of reliable information with which to make decisions. A series of studies were conducted to determine the practicality and impact of replacing software components with functionally equivalent alternatives.

A representative category of components was chosen and the performance of a selection of components was compared to determine whether selecting more performant components could reduce hardware requirements. The results of the performance measurements showed that the fastest component in the selected group was, on average, 2642 times faster than the slowest. The implication of this is that the selection and substitution of components can potentially have a large impact on the performance, and therefore the hardware requirements, of a large-scale system.

A prototype apparatus was developed to measure the energy consumption of software in operation. This apparatus revealed that the popular WordPress website management software consumes considerably more energy in operation

than an equivalent static website. The implication of this is that for many websites, moving from WordPress to a static website could result in immediate energy savings.

The apparatus was also used to compare the energy use of the class of components examined in the performance tests. This comparison showed that energy usage is not directly related to software performance and that different components have different energy usage profiles under different usage scenarios. The implication of this is that performance alone should not be used to predict energy use and that the most accurate way to determine the energy impact of a software change is to include energy measurements in existing test suites.

Key contributions to knowledge include: A novel self-contained apparatus for comparing software performance and energy consumption; energy use and performance comparisons for a cohort of web server implementations; the relative energy usage of WordPress compared to static websites; a novel extendable java framework for template engine comparison; energy use and performance comparisons for a cohort of template engine implementations; a challenge to the notion of execution speed as a proxy for software energy usage; a challenge to the notion of task complexity as a proxy for software energy usage; and an investigation into The efficacy of component substitution as a strategy to improve software sustainability.

This dissertation is the result of my own work and includes nothing that is the result of work done in collaboration except where specifically indicated in the text.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

Above all, endless thanks to my wife Margaret and my daughters Elizabeth and Katherine for putting up with this unexpectedly lengthy process and the associated plunge in family income.

Special thanks to Professor Nicholas Caldwell for his continuing support and for remaining professional while juggling the sometimes conflicting demands of being both PhD supervisor and line manager.

And finally, thanks to all my friends and colleagues who have helped out with suggestions, encouragement, comments, critiques, feedback and opportunities to discuss and publicise my work.

Chapter 1

Introduction

1.1 About This Research

▌ This research is motivated by concern for the future of our environment.

The world is facing an environmental crisis, and all organisations and individuals need to play their part in addressing climate change. We have already overstepped key planetary boundaries (Steffen et al., 2015) and continue to cause irreversible problems. The longer we delay in reducing carbon emissions to zero, the greater the cumulative impact on the world (Pierrehumbert, 2019).

▌ This research is situated in the area of sustainability.

The climate crisis and the environmental impact of humanity are aspects of sustainability, but sustainability itself can be a controversial concept. The literature is full of definitions of sustainability. A popular introductory approach is to describe sustainability as the ability or capability to "endure" (Mengesha, 2024), (Dixit and Kaur, 2024), (Venters et al., 2023), and many others. Although this definition may be correct from a dictionary perspective, it does not align well with common usage and is open to misunderstanding about *what*, exactly, should endure. Another commonly cited definition is from Brundtland (1987) "meeting the needs of the present generation while not compromising the ability of future generations to meet their needs". However, Owens (2003) describes this definition as leading to a "striking paradox" with the quest for sustainability being neither a "consensual" nor a "straightforward" project. In 1993, Allen and Hoekstra described sustainability as "an immature notion" that "conjures up different images" in different readers. More recently Moore et al. (2017) lists "lack of consistent definitions in the literature" as one of the major

challenges of research involving sustainability, finding 24 separate definitions in a survey of 209 articles. In 2015, United Nations defined a set of 17 *Sustainable Development Goals* (Figure 1.5.1) that have become a baseline for categorising research. The mapping of this research to the United Nations Sustainable Development Goals is given in Section 1.5.

In this dissertation, *environment* is used to refer to the natural environment, and *sustainability* is used in the sense of minimising environmental, especially climate, impact.

> This research considers the sustainability of Information and Communication Technology (ICT).

ICT is important because it participates on both sides of the metaphorical sustainability ledger. ICT can provide assistance and solutions to increase the sustainability of a wide range of processes. However, ICT systems also consume resources and energy to manufacture, distribute, operate and dispose of. ICT systems can help in any of the UN Sustainable Development Goals, for example by reducing travel through virtual meetings, calculating fuel-efficient delivery routes, or controlling agricultural machinery for greater crop yields. However, the manufacturing, operation, and disposal of such systems have a large environmental cost.

At the start of this research, estimates showed that the ICT sector contributed between 1.8% and 5% of the total greenhouse gas emissions worldwide, depending on the model used (Belkhir and Elmeligi, 2018). To put this into perspective, the higher end of Belkhir and Elmeligi's estimates proportion is roughly twice the emissions due to air travel over the same period. Despite high-profile climate pledges, this contribution has not decreased. With the progressive adoption of energy-intensive technologies such as 5G communications, big data, blockchain, and artificial intelligence, Gelenbe (2023) estimates that global ICT systems now consume approximately 10% of the world's electricity output. Freitag et al. (2021) reviews and critiques a selection of articles and concludes that the energy use of ICT, and its associated greenhouse gas emissions, will continue to increase unless there is an economic constraint such as a carbon tax or a strictly enforced cap on emissions. Based on a study in China, Wang et al. (2022) predicts an annual increase in ICT energy use between 7.5% and 15% at least until 2030.

Becker (2023) takes a stronger position and suggests that ICT is "insolvent" and may never be able to pay back its environmental and social costs.

> This research focusses on sustainability *of* ICT, rather than the use of ICT *for* sustainability.

Penzenstadler (2013) drew the key distinction between the use of technology *for* sustainability and the sustainability *of* that technology. Although the context of Penzenstadler's work was on software engineering and software systems requirements, this distinction has proved important for later research. The formulation "sustainability *of* X" vs "X *for* sustainability" has often appeared in later writing, with *X* chosen appropriately for the topic of each paper.

> This research focusses on the impact of software on the sustainability of ICT.

ICT is a very broad field, including basic communication and computing hardware such as cables, switches, and power supplies, but also computers (and other computer-controlled equipment) and the software systems that run them. Computer-controlled technology is important because it is *changeable*. Unlike, say, a steam engine or door hinge, which once constructed can only work in one way, computer systems can be reprogrammed to perform similar or different tasks in different ways. Although it is mainly the physical hardware of such systems that requires resources to construct or dispose of and consumes energy to operate, this hardware is controlled by software that has the potential to increase or reduce the amount of resources and energy required.

The potential environmental impact of changes to software is two-fold.

1. Software changes can affect the operational energy requirements of a particular hardware system.
2. Software changes can also increase or reduce the amount of ICT hardware required to address a particular need.

Real computer systems are often large (in the sense of many parts and/or many lines of code rather than physical size) (McCandless, Doughty-White, and Quick, 2015), complex, and difficult to fully understand (Peitek et al., 2021). I have spent more than 30 years working in software development and have observed first-hand how difficult it can be to determine whether any particular choice or change may result in a positive or negative contribution to the end result.

> In summary, my research explores some potential contributions to the huge problem of making software-driven computer systems more environmentally sustainable.

1.2 Research Scope and Objectives

Within the broad area established in Section 1.1, the research described in this dissertation is limited to a specific scope and objectives. The reading, reasoning and constraints that led to the specific scope and objectives of the investigation are explored in detail in Chapter 2.

In summary, this research is limited to the following scopes.

- Performance and energy usage of software used in large-scale web systems.
- Evaluation of software written in the Java programming language.
- The potential impact of replacing software systems or components with more energy-efficient ones.

Within the scopes listed above, this research has the following specific research objectives:

1. Investigate the context of web software and how it is developed (Chapter 3).
2. Explore the differences in performance of a selection of template engine components (Chapter 4).
3. Construct a test apparatus to compare the energy use of software during operation (Chapter 5).
4. Using the test apparatus, compare the energy use of common web software and the feasibility and effectiveness of substituting template engine components (Chapter 6)

To support the experiments for objective 4, a software system was developed to automatically adapt incompatible components and data to be used in a single experimental framework. The development of this system is described in Appendix E.

1.3 Research Timeline

The research described in this dissertation was conducted part-time and spanned an eight-year period from late 2016 to early 2024. Some key events in the research timeline are shown in Table 1.1.

Table 1.1: Research timeline

2016	Preliminary literature searches and planning for the research proposal and feasibility study.
2017	Official period of study began in January. Background research for chapters 1, 2, and 3. Feasibility study (Section 4.3).
2018	Design and acquisition of equipment for energy experiments (Chapters 5 and 6), but no available lab space. Planning for a study on sustainability in HE.
2019	Planning for improved performance experiments (Section 4.6). Web site analysis and initial emails for the HE study.
2020	University closures, general disruption and personal health concerns due to COVID-19. Continued the HE study.
2021	Improved performance experiments (Section 4.6). Completed the HE study.
2022	Finally got access to suitable lab space and started energy experiments (Chapters 5 and 6). Development of GILT language and processor (Appendix E).
2023	Completed energy experiments. Data analysis and writing up.
2024	Writing up. Submission and Viva.
2025	Post-Submission corrections.
2026	Post-Submission corrections.

1.4 Contributions To Knowledge

This research has generated several contributions to knowledge, listed in Table 1.2. Individual contributions are discussed in more detail in Section 7.3.

1	A Novel Self-Contained Apparatus for Comparing Software Performance and Energy Consumption *(Section 7.3.1)*
2	Energy Use and Performance Comparisons for a Cohort of Web Server Implementations *(Section 7.3.2)*
3	The Relative Energy Usage of WordPress Compared to Static Websites *(Section 7.3.3)*
4	A Novel Extendable Java Framework for Template Engine Comparison *(Section 7.3.4)*
6	Energy Use and Performance Comparisons for a Cohort of Template Engine Implementations *(Section 7.3.5)*
7	A Challenge to the Notion of Execution Speed as a Proxy for Software Energy Usage *(Section 7.3.6)*
8	A Challenge to the Notion of Task Complexity as a Proxy for Software Energy Usage *(Section 7.3.7)*
9	The Efficacy of Component Substitution as a Strategy to Improve Software Sustainability *(Section 7.3.8)*

Table 1.2: Summary of contributions to knowledge

1.5 Mapping to UN Sustainability Goals

The United Nations (UN) has established seventeen *Sustainable Development Goals* [Figure 1.5.1].

Figure 1.5.1: The 17 UN Sustainable Development Goals (United Nations, 2015)

This research contributes to the following UN goals:

- **9 (*Industry, Innovation, and Infrastructure*)** The research aims to improve the environmental sustainability of the ICT industry that forms a vital part of the global infrastructure.
- **12 (*Responsible Consumption and Production*)** The research is aimed at reducing the energy and other resources consumed by the global ICT industry.
- 13 **(*Climate Action*)** The motivation for this research is to help reduce the climate impact of ICT by reducing its energy consumption and related greenhouse gas production.

1.6 Thesis Structure

The main content of this thesis comprises a series of studies exploring different aspects of the bigger picture, brought together at the end for combined reflection and conclusions.

Chapter 2 introduces the context and terminology used in this research and then examines related research to identify a research gap and define research objectives.

Chapter 3 explores how software is developed and the forces that shape it to determine the potential for improvements in energy usage.

Chapter 4 investigates the differences in performance between a selection of template engine software components when used for a variety of scenarios.

Chapter 5 designs, constructs and evaluates a self-contained low-cost apparatus for comparing software energy usage. The apparatus is used to compare the performance and energy usage of a variety of web server applications, both when serving "static" websites and when using the popular *WordPress* content management application.

Chapter 6 makes use of the comparison apparatus described in Chapter 5 to investigate and compare the performance and energy use of the template engine components examined in Chapter 4.

Chapter 7 reflects and summarises the results of the research from previous chapters.

Chapter 2

Context, Literature and Research Objectives

This chapter explores the literature and prior work that informed the research described in this dissertation. The following sections provide an overview of related literature. Starting with some working definitions of terminology, the chapter proceeds in a series of subsections. Each subsection narrows the focus, from the big picture of global sustainability to the specifics of template software and experimental techniques, leading to clarification of a research gap in Section 2.16 and the enumeration of specific research objectives in Section 2.17.

2.1 A Note on Terminology

Much of the terminology in this field has not yet stabilised, and several key terms are used with different meanings or senses in adjacent fields. This section introduces and defines the key terminology that will be used throughout this dissertation and provides a sense of the competing terminologies in play in the literature. Note that, when discussing the literature, it is sometimes necessary to echo the terminology used by the authors.

Sustainability Sustainability is a wide field, including all seventeen of the UN Sustainability Goals (United Nations, 2015). The research discussed in this dissertation is in the overlapping area between two broad concepts, computing and sustainability. The mapping of this research to the UN sustainability goals is discussed in Section 1.5.

The overlap between computing and sustainability is known by many names such as *sustainable computing*, *computing sustainability*, *computing and sustainability*, *green computing*, etc. However, these names are imprecise and convey different meanings to different readers (Venters et al., 2014). Penzenstadler (2013) divided such research into two sub-areas: "Software Engineering *For* Sustainability", and "Sustainability *Of* Software Engineering". Although Penzenstadler's writing was about the specific discipline of software engineering, the distinction holds. Later writing by several authors uses a similar formulation but with other subdisciplines such as "system architecture" (Dixit and Kaur, 2024), or "computing education" (Mann and Smith, 2007).

The field of computing has its own set of terminology, some of which conflict with usage in other disciplines. The word *sustainability*, for example, has traditionally been used in software engineering in a sense equivalent to longevity or maintainability. In that sense, a computing system would be more sustainable if it could continue to do its job for a longer time. Likewise, the word *environment* is commonly used to refer to a combination of the computer hardware platform on which the software is to run and any external settings or configurations that affect the operation of the software.

> In this dissertation, *environment* is used to refer to the natural environment, and *sustainability* is used in the sense of minimising environmental, especially climate, impact.

ICT Information and Communication Technology (ICT) is a wide field that includes all aspects of hardware, software, networking and communication, data, digital media, and a host of associated protocols and formats. ICT is still relatively new compared to other fields of science and engineering. In this actively developing field, the terminology has not fully stabilised.

Computing Within the wide field of ICT, this research focusses on systems that use software technology. In academic terms, this includes a wide range of subdisciplines including *software engineering*, *software development*, *informatics*, *information science*, *data science*, *computer science* and *computer systems engineering*. The discussion continues as to whether some of these subdisciplines would benefit from merging (Fitzgerald, 2024).

Unit	Type	Definition
Joule	energy	The SI unit of energy. The amount of work done when a force of one newton displaces a mass through a distance of one metre in the direction of that force.
Watt	power	The SI unit of power. 1 Joule per second.
Watt-Hour	energy	1 Watt for 1 hour. 3,600 Joules.
Kilowatt-Hour	energy	1000 Watts for 1 hour. 3,600,000 Joules.
BTU	energy	The amount of heat needed to raise the temperature of one pound of water by one degree Fahrenheit at its greatest density. Approx 1,055 Joules.
Horsepower	power	The amount of power used by a horse to lift a 550 pound object from a depth of 1 foot in one second. Approx 746 Watts.

Table 2.1: Energy and power units

> In this dissertation, the umbrella term *computing* is used to include all aspects of the specification, architecture, design, construction, evaluation, operation, maintenance, and management of solutions and products that use software technology.

Energy and Power A major contributor to the environmental impact of computing systems is the consumption of energy, specifically electricity to power electrical and electronic systems. Other forms of energy are also used in the manufacture and operation of computing systems, for example for cooling or transport, but this research concentrates on electrical energy usage. The literature in this field uses a mixture of terminology and measurement units. In experimental literature where energy consumption is measured or estimated, the units are usually *Watts* (for power) and *Joules* (for energy), but some writing on this topic uses *Watt-hours*, *Kilowatt-hours*, or *BTU* (British Thermal Units) for energy values or *Horesepower* when comparing electrical power systems against internal combustion. The relationship between these units is given in Table 2.1.

> In this dissertation, SI units *Joules*, *Seconds*, and *Watts* are used for experimental design and results, with appropriate scaling prefixes as required.

Performance, Speed and Efficiency Another key terminological distinction is between the closely-related concepts of *performance*, *speed*, and *efficiency*.

Performance in computing literature is typically used in a sense equivalent to speed. This meaning is clear when discussing a software program that starts and then runs uninterrupted until completion. However, the meaning of performance is not so clear when discussing a service application that runs "forever" (or at least until explicitly terminated) but waits, for example for user input or for data to arrive, then processes, stores, or forwards that input. In such cases, performance may mean speed, but only during certain operations, or it may have a meaning closer to responsiveness or throughput. In discussions of algorithms, efficiency is typically used in a sense largely equivalent to performance. Code is more "efficient" if it does its job in less time. However, this becomes more complex when data is involved, as some algorithms may be faster or slower for some sizes or classes of data. With the increase in interest in energy usage, a need has developed for an energy-related efficiency measure in which code is more efficient if it does its job using less energy, known by various terms such as the generally popular *energy efficiency* but also *power efficiency* (Manner, 2023) (Chien, 2021), *green efficiency* (Salam and Khan, 2018), or *carbon efficiency* (Dorkal, 2023). In the energy domain, there is no direct equivalent to performance or speed, although some writers have explored using metaphorical terms such as *leanness* (Wirth, 1995) or *frugality* (Gancarz, 2023) or *responsible consumption* (Becker et al., 2015) for software that generally uses less resources. The popular terms "power consumption" and "energy use" are equivalent to the inverse of (energy) performance. García et al. (2006) attempted to encourage "a consistent terminology for software measurement", for example by recommending the use of the term "measurement" rather than the potentially-confusing "metric" but, on the whole, the publications reviewed for this research continue to use inconsistent and sometimes ambiguous terminology.

> In this dissertation, *performance* and *speed* are used for the time domain. *Energy use* and *power consumption* are used for the energy domain. Where *efficiency* is used in a potentially ambiguous context, it will be prefixed with *energy-* or *time-*.

Domain-specific Terminology Software development is an abstract field correspondingly rich in metaphors, such as the use of *component* for a fragment of software that is, or could be, reused in a different context. Some software development contexts provide a *library* of such components, which are then described as *library code* or just *libraries* Sometimes, a collection of components or library code will be grouped into an Application Programming Interface,

commonly referred to as an *API*. Some such software components, libraries, or APIs are available for free or included with software development tools. Others, sometimes referred to as *Commercial Off The Shelf* or *COTS* software, are commercial products.

> This dissertation will use the term *component* for any kind of reusable software, regardless of whether it is available separately or is part of a library or API and regardless of its cost.

The research described in this dissertation investigates the performance and energy usage of template engine components (as introduced in Section 1.2). There are several key templating concepts that inform this research and appear often in the literature.

Template A template is a way of ensuring consistency and reducing repetition when creating similar documents. Common text and layout are entered just once and then reused to produce many pages or documents. Each template is described in terms of a specific template language and designed to be rendered by a compatible template engine.

Template Engine A template engine (sometimes also called "template processor" or just "template software") is a software system or component that manages the rendering of *templates*. When an output document is required, the template engine locates an appropriate *template*, loads the data for the desired document into a *page context*, then processes the template by interpreting it according to the template language and replacing *placeholder expressions* with appropriate data from the page context.

Template Language A template language specifies how a *template* is expressed. Most template languages consist of a combination of fixed text ready for output (known as *boilerplate*) and symbolic placeholders. Some template languages use symbolic expressions for everything, including fixed text. Template languages and their placeholder expressions vary widely, but within many template languages it can be useful to consider two types of placeholders: *value expressions* and *control expressions*.

Value Expression A value expression is a sequence of symbols in a glstemplate language that represent the name or location of a data value. When the template is rendered, the value expression will be replaced by a textual version of the data value. If the data value is already textual it will usually just be included in the output. If the data value is numeric or some more

complex data type, the way it is rendered will depend on the template engine and any options specified in the value expression. The details of the formatting and syntax of such expressions are specified as part of the template language.

Control Expression A control expression is not directly replaced by a value in the rendered output, but instead informs the process of rendering the template. Typical control expressions mimic those of conventional programming languages and include, for example: variables, decisions, and loops. Others may have side-effects, such as modifying the template context, emitting logging messages, or otherwise affecting the template expansion process, specific to a particular template engine or template language.

Page Context A page context is the way that a *template engine* finds data to use when processing *value expressions*. Most *template engines* treat the page context as if it is a simple named data store containing everything needed by the *value expressions* in the *template* currently being processed. However, page context implementations are also available that do not pre-load data before evaluating the *template*, but instead load or calculate data as needed during the processing of the *template*. As each rendered document will potentially be different, a new page context is provided for each document, and the rendering process evaluates the *template* using the specific data from that page context. Depending on the strategy used by the *template engine*, the *template* to use for each document may be determined by examining the page context, or it may be specified externally.

2.2 Literature Search Methodology

Research in the literature on web template technology presented several challenges and required the selection of a search methodology to reduce the number of inappropriate results.

2.2.1 Search terms and discipline conflicts

The key term “template” has a specific meaning for this research, but is also a term used in many other fields with different meanings. A simple literature

search on a general academic search engine (*Google Scholar*[1]) for the term "template" showed results in fields as diverse as management science (Jensen and Szulanski, 2007), organic chemistry (Hu, Masoomi, and Morsali, 2019), brain structure (Brett et al., 2001), sports medicine (Foster et al., 2009) and psychology (King, 1998). It is highly likely that the term is also used in many other fields and contexts with further different meanings. Even within the field of computing, the term is also used in image processing (Matthews, Ishikawa, and Baker, 2004), cryptography (Chari, Rao, and Rohatgi, 2003), and signal processing (Jain, Zhong, and Dubuisson-Jolly, 1998).

The search for the more precise term "template language" produced many results in the field of mathematics. A search for "template processor" revealed papers dealing with computer circuit design. A search for "template document" yielded a wide variety of results ranging from word processing tutorials to historical research processes. A search for "web template" produced mostly articles on "template extraction" - a technique for finding commonalities between web pages and arguably the inverse of this research. A search for "templating" responded mostly with articles on materials chemistry. The term that appeared to align best with the general area of this research was "template engine". This combination of words produced a larger proportion of results specific to this area. However, the sole use of this term would risk filtering out legitimate literature results that use alternative terminology.

2.2.2 Multiple subdisciplines and lack of consistent naming for the field of computing

An obvious step to eliminate search results from other disciplines is to select only results from the field of computing. However, this is also a challenge. As discussed in Section 2.1, the broad discipline of computing consists of a number of overlapping subdisciplines. Annotating a search with the word "computing" produces a large number of results, most of which are about computing as an activity within other disciplines rather than results from the discipline of computing. Selecting only results from one subdiscipline, for example, "informatics" or "software engineering", limits the results to that subdiscipline and excludes papers that position themselves within other subdisciplines.

[1] `https://scholar.google.co.uk/`

2.2.3 Selection of Databases

An alternative approach to reduce the number of inappropriate search results is to search within discipline-specific directories. Two key directories for the discipline of computing are *The ACM Digital Library*[2] and *The IEEE XPlore Digital Library*[3]. However, these directories are not concerned only with the topic of this research, so searches within these directories still result in some inappropriate results for the simple searches mentioned in Section 2.2.1. Other, more general directories such as *ScienceDirect*[4] contain some publications on computing, but these general directories also contain publications from many of the areas with conflicting terminology.

The academic literature searches for the following sections used the IEEE and ACM databases. The search terms varied for each section, but always used a similar strategy: select one or more target terms, then progressively add *NOT* clauses to exclude inappropriate categories until enough references were discovered.

2.2.4 Alternative Sources

Research in the field of computing is not exclusive to academia. Practitioners often conduct their own research and share their results and experiences through a wide range of channels such as blogs, magazines, independent conferences, and personal websites. Although these sources are generally not peer reviewed, they are indicative of current practice and current concerns and should therefore not be automatically excluded from this research. Where applicable, references and insights from this "grey literature" are included in the literature study in this chapter.

2.3 The Global Importance of Sustainability

Sustainability is one of the defining topics of the age (Brundtland, 1987). Where once the human race generally assumed that Earth's resources were inexhaustible, we are now aware of the existence of *Planetary boundaries* (Steffen et al., 2015) and the impact that human activity and development have on the environment. Pierrehumbert (2019) analysed the contribution of human activity to global warming and the potential effectiveness of large-scale

[2] `https://dl.acm.org/`
[3] `https://ieeexplore.ieee.org/Xplore/home.jsp`
[4] `https://www.sciencedirect.com/`

mitigation strategies such as carbon storage and "albedo hacking" and concluded that "There is no Plan B for dealing with the climate crisis". The only feasible solution is to immediately and significantly reduce greenhouse gas production. However, this is not a simple process. Beattie (2010) examined psychological aspects of why "saving the planet" is hard and Goodland (2002) explored the tangled human, social, economic and environmental aspects of sustainability.

2.4 What Sustainability Means for Computing

Sustainability is a complex topic with multiple dimensions (United Nations, 2015). Although there has been a lot of writing on computing and sustainability, much of it has focused on aspects such as social inequality and pollution (Hilty, 2011) rather than on energy use and greenhouse gas emissions. The exact impact of computing on greenhouse gas emissions is difficult to quantify. The popular media and literature are filled with wild claims. Freitag et al. (2021) summarised peer-reviewed research in this area and concluded that the contribution of ICT to greenhouse emissions is probably between 2.1% and 3.9% of global output. This is more than the entire output of the aviation industry (EESI, 2022). Knowles et al. (2022) highlighted an attitude of "digital exceptionalism" that assumes that computing is the general solution to environmental problems. This attitude is evident in the large body of *Computing for Sustainability* research that presents computing solutions without considering their wider impact. Coroama and Hilty (2009) made the case that environmental benefits and costs should both be considered in such research and recommended "decomposing the ICT monolith" to examine the energy use and energy benefits of its constituent parts. Toczé et al. (2022) introduced the concept of *unsustainability patterns*, such as unsolicited marketing ("spam") and excessive data transfer from *Internet of Things* (IoT) devices, which consume resources for little or no overall value. The authors also observed that the users of computing systems are often uninformed or powerless when it comes to choosing more sustainable ICT services. Koomey et al. (2009) stated that "far too little attention has been paid to the true total costs for data center facilities". Other researchers also pointed out that the response to system changes or improvements is not always beneficial. A change such as an improvement in speed, efficiency, or usability can trigger increased use of the system (through so called *rebound effects*) that can negate or even overshadow the original improvement (Hilty et al., 2006) (Gossart, 2015) (Adelmeyer et al., 2017).

The overall field of ICT has many names and subdomains, as noted in

Section 2.1, which has led to disjointed recognition and adoption of sustainability across the field. Naumann (2008) proposed "sustainability informatics" as "a new subfield of applied informatics" while admitting that some aspects of sustainable development had previously been included under the term "environmental informatics". Two years later, Tilson, Lyytinen, and Sørensen (2010) described digital infrastructures and their sustainability as "the missing IS research agenda". Penzenstadler et al. (2014a) presented sustainability as an additional *nonfunctional requirement* in addition to safety and security and considered second- and third-order effects of sustainability initiatives, including rebound effects. Meanwhile, Venters et al. (2014) characterised software sustainability as a "tower of babel" in which:

> the term software sustainability is frequently used to embody vague, diverse and contradictory ideas that are neither sound nor novel (Venters et al., 2014, p. 5)

Becker et al. (2015) produced the much-cited "Karlskrona Manifesto for Sustainability Design" consisting of nine key principles and commitments intended to "redefine the narrative on sustainability and the role it plays in our profession" [Table 2.2].

Manotas et al. (2016) performed a study of the perspectives of practitioners on "green" software engineering and collected responses such as:

> Our main concern is marketshare and that means user experience is a priority. We can be more efficient to try to cut costs, but since we don't charge by energy used this doesn't make us more attractive to users. So we tend to focus on other things like performance or reliability. (Manotas et al., 2016, p.241)

In 2017, Pinto and Castor proposed "energy efficiency" as "a new concern for application software developers" and Jagroep et al. (2017) called for an "awakening" awareness of energy consumption in software engineering. Fonseca, Kazman, and Lago (2019) produced a "manifesto for energy-aware software", lamenting that:

> as software engineers, we were never taught to consider, much less manage, the energy consumption of the software systems we created. (Fonseca, Kazman, and Lago, 2019, p.79)

Sustainability is systemic	Sustainability is never an isolated property. Systems thinking has to be the starting point for the transdisciplinary common ground of sustainability.
Sustainability has multiple dimensions	We have to include those dimensions into our analysis if we are to understand the nature of sustainability in any given situation.
Sustainability transcends multiple disciplines	Working in sustainability means working with people from across many disciplines, addressing the challenges from multiple perspectives.
Sustainability is a concern independent of the purpose of the system	Sustainability has to be considered even if the primary focus of the system under design is not sustainability.
Sustainability applies to both a system and its wider contexts	There are at least two spheres to consider in system design: the sustainability of the system itself and how it affects the sustainability of the wider system of which it will be part of.
System visibility is a necessary precondition and enabler for sustainability design	Strive to make the status of the system and its context visible at different levels of abstraction and perspectives to enable participation and informed responsible choice.
Sustainability requires action on multiple levels	Seek interventions that have the most leverage on a system and consider the opportunity costs: Whenever you are taking action towards sustainability, consider whether this is the most effective way of intervening in comparison to alternative actions.
It is possible to meet the needs of future generations without sacrificing the prosperity of the current generation	Innovation in sustainability can play out as decoupling present and future needs. By moving away from the language of conflict and the trade-off mindset, we can identify and enact choices that benefit both present and future.
Sustainability requires long-term thinking	Consider multiple timescales, including longer-term indicators in assessment and decisions.

Table 2.2: Principles from the Karlskrona Manifesto (after Becker et al. (2015))

Research in this area continues. Venters et al. (2021) revisited the earlier "tower of babel" characterisation and highlighted the need to:

> take into account the direct and indirect negative impacts on different dimensions of sustainability that result from the development, deployment, and continued use of the software system (Venters et al., 2021, p. 2)

In addition to research that explores and documents the current state of software and sustainability, some literature also considers potential solutions. Widdicks et al. (2018) suggests "Undesigning the Internet", an approach that reduces personal usage of computing resources by temporarily withdrawing from connected activities. Although this may potentially have environmental benefits, it also suffers from the general powerlessness and lack of control reported by Toczé et al. (2022). Background processes and systems consume energy regardless of whether a particular user is accessing the service. Emails and social media content continue to pile up even while you are disconnected. A different approach to addressing sustainability of software is to provide increased information and choice to end users. While still not common for desktop or server software, there have been several attempts to promote "energy labels" for mobile applications (Wilke et al., 2012) (Baek, Park, and Lee, 2018) (Behrouz et al., 2015) following media coverage of mobile applications that required excessive energy and drained phone batteries.

2.5 Literature Overviews and Summaries

Other researchers have also reviewed and summarised the literature in this area, many of whom have followed formal guidelines such as those provided by Kitchenham and Charters (2007). Hilty, Lohmann, and Huang (2011) attempted an overview of sustainability and ICT, and noted the rise of emphasis on the environmental impact of computing systems following the Gartner report *Green IT: a new industry shock wave* (Mingay, 2007). Hilty, Lohmann, and Huang also highlighted the potential of *rebound effects* illustrated by the continuing increase in ICT use and its energy consumption despite great increases in efficiency over the same period.

Penzenstadler (2012) compiled a systematic literature review that noted some of the terminology confusion mentioned in Section 2.1 but concluded that "currently, there is little research coverage on the different aspects of sustainability in software engineering". However, the scope of this literature review was limited to literature

on software engineering or requirements for software systems, so it may have rejected literature positioned in other fields of computing. Despite the scope limitations, this literature shows that there is a growing need for research on the environmental impact of software systems.

Calero, Bertoa, and Moraga (2013) surveyed literature between 1992 and 2012 for "software sustainability measures" and observed an increase in literature over the period with the first mention of "software sustainability" identified in 2003. Calero, Bertoa, and Moraga took a broad view of sustainability, with only a passing discussion of energy use and no mention of the more modern concerns of greenhouse gases and global warming. Meanwhile, Kern et al. (2013) also studied existing literature on quality aspects of "green" software and software engineering, but with an explicit emphasis on energy usage, energy measurement and energy saving. Kern et al. also suggested a *GREENSOFT* model to classify energy-related software engineering approaches. The comparison of these contemporary studies shows how a relatively small shift in emphasis and inclusion criteria can produce very different results.

Penzenstadler et al. (2014b) followed an earlier systematic review (Penzenstadler, 2012) with a mapping study of "software engineering for sustainability" that found considerably more publications than their previous study and noted the formation of several research clusters in which authors regularly collaborate with or cite other cluster members. Later literature studies tended to observe Penzenstadler (2013)'s distinction between "sustainability of X" and "X for sustainability" to explore more specific niches within one or the other. Examples of more specific literature surveys include sustainability requirements (Chitchyan et al., 2016), architecture (Paradis, Kazman, and Tamburri, 2021), tactics for improvement (Balanza-Martinez, Lago, and Verdecchia, 2023), and energy measurement methodology (Hindle, 2016). This period also witnessed a rapid growth in the use of smartphone "apps" and a concomitant increase in literature research on energy consumption and battery life in mobile devices such as Ahmad et al. (2015), Moreira, Alves, and Andrade (2020) and Schuler and Kotsis (2023). This literature highlights the breadth of the field and the increasing tendency to specialise by application niche or methodology.

More recent literature studies include Venters et al. (2023), which surveyed the broad field of sustainable software engineering and considered future trends, and Lee et al. (2024), which explored "energy concerns" in software engineering and found them "taken into account in all phases of software development and operation". This literature indicates the increasing desire for information about energy use in software engineering.

Considered as a whole, the literature studies above illustrate the importance of the field as well as a continued growth in interest and increased specialisation. Specific areas are examined in more detail in the following sections.

2.6 Sustainability in Requirements and Architecture

In the classic software development life cycle (SDLC), requirements and architecture phases take place before system implementation. This presents a natural point at which to consider sustainability issues. Formulating requirements for sustainability is challenging because, other than a few specific cases such as operational battery life, sustainability is a *non-functional* requirement without clear guidelines or measurable acceptance criteria. The most compelling non-functional requirement is often monetary cost, so Gu, Lago, and Potenza (2012) proposed codifying "green" strategies in financial terms and found it useful in some situations but not globally applicable. Condori-Fernandez and Lago (2015) and (2018) attempted to align sustainability requirements with other quality requirements and found some overlap with aspects such as modifiability, freedom from risk, and satisfaction. Bashroush, Woods, and Noureddine (2016) surveyed software architects about environmental impact and energy use and concluded that the sector lacks tool support, information, and prioritisation from stakeholders. However, as Penzenstadler, Femmer, and Richardson (2013) pointed out, identifying stakeholders and their objectives is not always a simple task. Kazman et al. (2018) looked specifically at energy consumption as a quality attribute and investigated strategies for modelling and prototyping to reason about designing for better energy use, but this approach still depends on consideration and inclusion of appropriate energy targets during requirements gathering.

There have been several attempts to construct models and frameworks to aid in the inclusion of sustainability during the creation of software systems. Dick and Naumann (2010) proposed enhancements to the SDLC to include a sustainability journal and retrospectives. This research was later developed into the *GREENSOFT* model that promoted "a cradle-to-grave product life cycle model for software products, sustainability metrics and criteria for software" (Naumann et al., 2011). The Sustainability Awareness Framework (*SusAF*) is a framework based on Design Science Research (DSR) (Vom Brocke, Hevner, and Maedche, 2020) that "provides interested stakeholders with a supported process for thinking about and expanding their anticipation of the possible sustainability effects of their IT products and services" (Betz et al., 2024) and includes questions and discussion guidelines to provoke consideration of five

dimensions of sustainability during the requirements process. Saputri and Lee (2016) proposed a framework to "analyze the dimensions of sustainability and structure it into software requirements". Their framework is based on questions for stakeholders and relies on quantifying and weighting answers.

These kinds of models and frameworks can be useful in adding structure to the consideration of sustainability issues when discussing system requirements and potentially during the architecture, design, development and operation of the system but they also have limitations. Any framework or model based primarily on discussions with stakeholders does not cope well with stakeholders who either do not know or do not care about the sustainability of the resulting system. Such models rely on the presence of "domain experts" (Christel and Kang, 1992) who understand what is needed. This is probably true when it comes to the functional and non-functional requirements that describe the *task* of the system but neither "domain experts" nor software developers are likely to understand the sustainability impact of their requirements choices in the same way (Noman et al., 2022) (Souza, Haines, and Jay, 2014).

Rather than include sustainability in the requirements elicitation phase of development, the Green Software Foundation (GSF) (Green Software Fioundation, 2024) offers six "key areas" [Table 2.3]. The first four of these areas act as implicit, non-negotiable, requirements for every project and the final two as guidelines for how to achieve them.

1	**Carbon Efficiency**	Emit the least amount of carbon possible.
2	**Energy Efficiency**	Use the least amount of energy possible.
3	**Carbon Awareness**	Do more when the electricity is cleaner and do less when the electricity is dirtier.
4	**Hardware Efficiency**	Use the least amount of embodied carbon possible.
5	**Measurement**	What you can't measure, you can't improve.
6	**Climate Commitments**	Understand the exact mechanism of carbon reduction.

Table 2.3: GSF key areas (after Green Software Fioundation (2024))

Even when stakeholders do understand and prioritise the sustainability aspects of the system there is also the further problem of a lack of information. Software architectures, designs, tools, and components do not come with sustainability ratings, so in many cases there is no way to judge whether any option will be "better" or "worse" from a sustainability perspective without implementing it.

In cases where requirements include sustainability, these sustainability

requirements then need to be incorporated into the architecture and design of the system. Ameller et al. (2012) interviewed a group of software architects about the inclusion of such non-functional requirements and found that non-functional requirements suffered from "terminological misunderstandings", were often managed independently from other requirements, and were rarely validated to confirm appropriate implementation. LaToza, Shabani, and Hoek (2013) interviewed software developers and noted that "architectural decisions often become technology decisions, which are in turn influenced by both technical and social factors" but such decisions are vulnerable to the later discovery of incompatibilities that can require major re-work. Venters et al. (2017b) also considered software sustainability as an aspect of architecture with particular emphasis on managing the different dimensions of sustainability from the Karlskrona Manifesto (Becker et al., 2015). Lago (2019) explored the use of *decision maps* to reason about the interactions and conflicts between different architectural requirements and found "much confusion" between long and short-term goals as well as difficulties associating their *quality concerns* with measurable values.

Other literature related to sustainable requirements and architecture tends to address specific technological areas such as cloud computing (Khomh and Abtahizadeh, 2018) (Chen et al., 2012), virtual machines (Marcu and Tudor, 2011), and re-engineering (Jelschen et al., 2012).

During the course of this research, the *Well-Architected Framework* (Amazon, 2024), promoted by Amazon Web Services[5] (AWS), was updated to include an explicit "sustainability pillar". AWS is the dominant cloud service provider, used by 2.38 million businesses in 2024 (HG Insights, 2024). AWS provides documentation and guidance for all software developers who design or implement code that works with AWS services, and the *Well-Architected Framework* is a primary entry point for anyone learning about cloud systems. The inclusion of an explicit sustainability pillar in this framework indicates the value placed on sustainability issues in the architecture of cloud computing systems, in addition to the traditional concern of minimising operational costs.

2.7 Challenges of Sustainable Software

Architectural decisions notwithstanding, software is created by software developers, and the process of implementing sustainable software poses many challenges. Pang et al. (2016) investigated attitudes of software developers to

[5] `https://aws.amazon.com/`

software energy consumption and discovered that more than 80% of the people surveyed did not take energy consumption into account when developing software. Pinto, Castor, and Liu (2014) examined the popular developer resource *Stack Overflow*[6] for attitudes on software energy consumption and encountered both a wide diversity of questions and answers that were flawed or vague.

Software is developed in a wide variety of ways ranging from the rigid "cathedral" to the chaotic "bazaar" (Raymond, 1999); from secretive commercial applications to public participation (Ballhausen, 2019); from requirement-driven to agile (Dick et al., 2013); and in team sizes from a lone coder to organisations with thousands of employees (Sawyer, 2004). Naumann et al. (2015) attempted to synthesise a practice of *green software engineering* from this broad field but ended with more questions than answers.

One of the major issues for green software development is the continually-evolving nature of software. Every change to a software system can potentially affect its sustainability and environmental impact. Betz et al. (2015) tackled this problem by introducing the concept of *sustainability debt*, while Couto et al. (2020) took a similar but more specific approach with *energy debt*. Ren et al. (2004) proposed a tool for change impact analysis of Java programs. All such approaches, however, rely on being able to assess the sustainability or the energy usage of the system being developed (Jagroep et al., 2016b). Software energy usage measurement is discussed in depth in Section 2.10 of this dissertation and an apparatus for software energy measurement is developed in Chapter 5.

In the absence of effective direct energy measurement, developers typically resort to guidelines and *rules of thumb* (Aggarwal, Hindle, and Stroulia, 2015). Examples of such techniques are *design patterns* (Gamma et al., 1994) and *code smells* (Fowler and Beck, 1999). Both the sustainability impact of design patterns (Sahin et al., 2012) (Noureddine and Rajan, 2015) (Litke et al., 2005), their associated refactoring (Silva et al., 2010) (Sahin, Pollock, and Clause, 2014), and smells/leaks (Gottschalk et al., 2012) (Palomba et al., 2019) (Pereira et al., 2020) (Vetro et al., 2013) have been extensively explored in the literature. Design patterns and code smells are often positioned as aids to software maintainability, so Mancebo, Calero, and García (2021) investigated the relationship between maintainability and energy consumption and concluded that classic measures of software maintainability, such as cyclomatic complexity, did not correlate well with energy usage, but that, in general, software with more code tended to use more energy overall.

[6] `https://stackoverflow.com/`

2.8 Components: Commercial, Free, and Open Source

Modern software development commonly makes use of existing software, in the form of components such as source code examples, libraries, or APIs. Understanding the characteristics of such dependencies forms a key aspect of reasoning about the sustainability characteristics of the system as a whole. Mileva, Dallmeier, and Zeller (2009) and (2010) investigated the popularity of libraries and APIs, Hejderup, Deursen, and Gousios (2018) developed a *software ecosystem call graph* for dependency management and Bauer and Heinemann (2012) attempted to model API usage to inform decision-making. All of these approaches, however, only make sense for energy sustainability evaluation if the energy characteristics of the components are well understood. This is a separate area of research. For example, the Java "collections" classes, part of the standard Java library, have been comprehensively evaluated for energy consumption (Hasan et al., 2016) (Pereira et al., 2016) (Pinto et al., 2016).

Capra, Francalanci, and Slaughter (2012) concluded that the design of software systems, particularly the selection and use of frameworks and software libraries, can have a "significant impact on the energy efficiency of software applications". The authors compared the total energy required to perform an operational scenario on two MIS (Management Information Systems) applications running on similar computer hardware and determined that one application required approximately twice the energy for the same task. They also observed that the relationship between performance and energy use was complex and non-linear, affected by factors such as the underlying hardware and operating system platform and the specifics of the evaluation scenario.

At the end of the 20th century there was a lot of research on software re-use and its benefits to productivity (Lim, 1994) (Basili, Briand, and Melo, 1996), and success factors (Frakes and Isoda, 1994) (Kim and Stohr, 1998). Initially, re-usable components were largely commercial products, known as *Commercial Off The Shelf* (COTS). Boehm and Abts (1999) considered the practicality of integrating COTS components, calling it "plug and pray". Lawlis et al. (2001) proposed a formal process for evaluating COTS software products, but Torchiano and Morisio (2004) pointed out a list of "overlooked" problems with integrating COTS components into an application including disagreements on terminology, missing or incomplete features, and a lack of standards.

This period also saw the rise of *open source software*, which upset the economics of COTS for everything except large applications or services (Lerner and Tirole, 2002). Once people began to understand the motivation of open source

developers (Hertel, Niedner, and Herrmann, 2003) (Lakhani and Wolf, 2003), it rapidly became the predominant method of code reuse (Mockus, 2007) (Haefliger, Von Krogh, and Spaeth, 2008) (Sojer and Henkel, 2010).

Open source software is characterised by being free in several dimensions including: freedom to obtain, freedom to use (both in whole and in part), and freedom to modify for any use. To be classified as true open source, software must be released under a licence approved by the Open Source Initiative (OSI) [7]. The full criteria for an OSI-approved licence are defined in the Open Source Definition (Open Source Initiative, 2024b). Software that meets some, but not all, of the OSI criteria is not true open source, even if it is free to obtain and use. Despite this, many software manufacturers like to claim that their products are open source. As an example, *Unreal Engine*[8] makes the source code available, but the licence terms[9] require a fee from certain classes of users. Similarly, Matlab[10], although appearing to be fully free, has restrictive licence terms[11].

With the growth of data-intensive software such as large language models (LLM), there have been new challenges to the concept of open source. OSI have released a draft version of an open source AI definition (Open Source Initiative, 2024a) while facing misinformation from Mark Zuckerberg's Meta Corporation[12] (Rudra, 2024).

However, open source software is not without its risks. In 2016, thousands of software products broke when an angry software developer removed his code from a public repository (Williams, 2016). This raised a lot of questions about the prolific and largely unquestioning use of open source components (Abdalkareem et al., 2017), and provoked interest in tracking the composition of software through a software *bill of materials* analogous to the list of parts for physical manufacturing (Stalnaker, 2023) (Xia et al., 2023), and a software component "fingerprint" technique known as *bertillonage* (Davies et al., 2013).

Badampudi, Wohlin, and Petersen (2016) conducted a systematic literature review on the decision-making process for software components, selecting between in-house, open source, COTS or outsourcing. They identified four key criteria: time (to implement once selected), cost (to purchase or subscribe), effort (to evaluate before selecting) and quality (of implementation, support, etc.). Aside from cost, which is often known at the start of the process, the other factors are initially unknown and will require effort to determine. This goes some way to explain why

[7] `https://opensource.org/`
[8] `https://www.unrealengine.com/en-US`
[9] `https://www.unrealengine.com/en-US/license`
[10] `https://uk.mathworks.com/products/matlab.html`
[11] `https://uk.mathworks.com/pricing-licensing.html`
[12] `https://www.meta.com/gb/`

cost-free open source components are so often selected. However in cases where the choice is between multiple cost-free options, there are no remaining criteria that do not require at least some additional effort.

2.9 Selecting Components and Libraries

Hucka and Graham (2018) conducted a literature review and survey to explore how scientists and engineers find and evaluate software. They discovered that developers use five key sources of information: general purpose web search; ask colleagues, look in social help sites such as *StackOverflow*, search in public software project repository sites such as *GitHub*, and look in scientific literature. Larios Vargas et al. (2020) studied the challenge of selecting third-party libraries, found 26 separate potentially conflicting decision factors, and argued that "the lack of a systematic approach may lead software developers to choose libraries arbitrarily, without considering the consequences of their decisions". This decision process has only become more difficult with the addition of AI systems and components (Rani et al., 2025).

Milkman, Chugh, and Bazerman (2009) explored psychological aspects of decision making in general and how it might be improved, while Nguyen et al. (2020) used collaborative filtering techniques to develop a system for recommending third-party libraries based on public repository data. De La Mora and Nadi (2018) used similar repository data to propose a "metrics-based" comparison of software libraries. Unfortunately, the public information used by these systems is limited to general metadata such as rating, popularity, and release frequency and provides no guidance on whether a library is more or less suitable for specific functional or non-functional requirements. Anwar et al. (2020) explored energy consumption as a factor in the choice of HTTP client libraries. They concluded that choice of library does make a difference to energy consumption, but as no suitable public data was available they needed to perform their own experiments to determine the energy consumption of each library.

2.10 Experimental Methodology Literature

Software has a rich history of performance measurement and "benchmarks". Lilja (2000) offers a good grounding in the area. Georges, Buytaert, and Eeckhout (2007) explored the specific challenges of performance measurement of Java code running in a *JVM* bytecode virtual machine while Gu, Verbrugge, and Gagnon (2006) investigated the performance impact of differences between

virtual machine implementations and Blackburn, Cheng, and McKinley (2004) discussed the "myths and realities" of performance and garbage collection. Such research was exclusively in the time domain, however, with no mention of energy use.

More recently, research has included elements of energy-efficiency and power consumption (Capra, Francalanci, and Slaughter, 2012) (Li et al., 2014) and even some attempts at standardised energy benchmarks (SPEC, 2008). Despite this, the relationship between performance and energy use is complex, with some research validating the "folklore" that better performance implies better energy consumption (Yuki and Rajopadhye, 2013), while others discuss "tradeoffs" between the two measurements (Joseph, Brooks, and Martonosi, 2001) and yet others point out that architecture choices can drastically alter the relationship before coding even begins (Khomh and Abtahizadeh, 2018).

Researchers in this area generally agree that understanding energy use of software is vital in order to manage power consumption (Snowdon, Petters, and Heiser, 2005), but methodology for determining energy use varies widely. In addition to "folklore" that energy use can be predicted directly from performance, researchers have also tried to derive models relating energy use to many other measurable aspects of software. The aim of such models is usually to remove or reduce the need for expensive and time-consuming power measurements by replacing them with automatically calculable metrics. In the words of Povoa et al. (2013) "a model for estimating energy consumption should be simple, i.e., to consider only a subset composed of the most influential variables on energy consumption". Such energy usage models fall roughly into two groups: *dynamic* models and *static* models.

Dynamic models base their energy usage predictions on observing various aspects of the software in operation. As an example, Povoa et al. produced their dynamic model by running a software "monitoring agent" and reading 47 different values such as the amount of CPU time in user and system modes, memory usage, time spent reading and writing disk storage, number of packets sent and received over network interfaces and so on. They then ran some "synthetic workloads" (p.6) and used linear regression to produce an energy model that they then compared against measured overall energy usage for the same workloads. Chowdhury et al. (2015) observed the "system calls", by which applications make use of operating system features, and also used linear regression to produce an energy model. Stoico et al. (2023) used layered queuing networks (LQN) to produce an energy model based on CPU and disk performance while processing images.

Static models base their energy usage predictions on analysis of the software and its environment without running the code. (Ibrahim, Rupp, and Fahmy,

2011) produced an energy model based on examining the machine instructions generated when software was compiled. Stier et al. (2015) produced an energy model based on the characteristics of system architectures. Ardito and Torchiano (2018) measured the energy usage of a Raspberry Pi when idle and when running software designed to fully utilise each specific hardware feature, then used this data to infer energy usage of an application from the use of these features in the code. Hao et al. (2013) took the similar approach of generating a software environment energy profile (SEEP) for the target device and then used that to annotate individual lines of source code with predicted energy usage values.

The kinds of models produced by the above, however, are rarely generalisable beyond their experimental context (Colmant et al., 2018). Derived regression models and rules of thumb are specific to the combination of hardware, software, data and test scenarios that were evaluated. Static analysis models take no account of time, data variations or external input when the software will be executed.

The remaining way to obtain information about the energy usage of software is to measure it. Research in the literature has used a range of measurement techniques including:

- Software modification (Seo, Malek, and Medvidovic, 2008) (Do, Rawshdeh, and Shi, 2009) (Sabovic et al., 2020)
- System emulation (Sinha and Chandrakasan, 2001) (Gurumurthi et al., 2002) (Wilke, Götz, and Richly, 2013)
- Power logging circuitry within the computer hardware (Dutta et al., 2008) (Kansal and Zhao, 2008) (Noureddine et al., 2012)
- General-purpose lab equipment (Flinn and Satyanarayanan, 1999) (Farkas et al., 2000) (Ge et al., 2009) (Ardito and Torchiano, 2018)
- Consumer energy meters (Kaup, Gottschling, and Hausheer, 2014) (Kaup et al., 2018) (Bekaroo and Santokhee, 2016)
- Battery simulation (Zhou and Xing, 2013) (Naderiparizi et al., 2016)
- Custom power-measurement electronics (Jiang et al., 2007b) (Stathopoulos, McIntire, and Kaiser, 2008) (Andersen and Hansen, 2009) (Astudillo-Salinas et al., 2016)

Techniques that rely on modification of the software to be evaluated (sometimes also called *instrumentation*) may not be suitable for evaluation of "black box"

components and also raise questions about the validity of evaluating something that differs from the original software. Simulation techniques require powerful systems to run the simulation, and also require trust that the simulation is representative of the system being tested. The use of internal power logging circuitry relies on the presence of the specialist circuitry, so is only suitable for testing with specific hardware. In many cases such circuitry only measures power consumption of certain hardware within the system, typically the CPU and memory (Intel, 2019), and does not measure energy used by other devices such as storage drives or additional hardware for graphics, network communication or AI acceleration. Professional lab equipment is expensive and uncommon in commercial settings. For example, Dzhagaryan et al. (2016) used a National Instruments PXIe-6361 Data Acquisition Device in a National Instruments chassis at a total cost of around $5500 while Manotas, Pollock, and Clause (2014) and Ardito and Torchiano (2018) used National Instruments USB data acquisition devices costing roughly $2000 each. Rice and Hay (2010), Povoa et al. (2013), and Milosevic et al. (2013) used similarly expensive equipment. Consumer energy meters have the advantage that they measure full system power and are cheaper than professional lab equipment, but often have limited accuracy and sample rate (Hindle, 2012b), and some also require manual triggering or reading of values, which limits their use in automated measurements.

Dezfouli, Amirtharaj, and Li (2018) analysed power measurement approaches from the literature and rejected all of the above as well as any custom electronics that they considered to be "complex". The authors selected instead a solution using an low-cost INA219 power measurement integrated circuit (Texas Instruments, 2015) controlled by a widely-available Raspberry Pi single-board computer (Raspberry Pi Foundation, 2023b). Hindle (2012a) initially used a consumer *Watts Up Pro* energy meter but, following issues with accuracy and resolution, also moved to a measurement solution based on the INA219 and a Raspberry Pi (Hindle et al., 2014), as did Chowdhury et al. (2015). The INA219 device requires an external load resistor and can only measure current on the "high-side" of a power circuit. Texas Instruments also provide a more flexible device, INA260, which includes the load resistor and can measure both high- and low-side current (Texas Instruments, 2016). A circuit board containing an INA260 and associated circuitry can be bought for around $10 (Adafruit, 2023).

2.11 Comparison Approaches

Despite the issues with general-purpose energy models discussed above, energy measurement can still be used for energy usage *comparisons*. In this approach, most of the variables that limit the applicability of energy models are controlled. Energy measurements for different software are run on the same hardware and system infrastructure and using identical test scenarios with identical data and other input. This leaves the software as the key variable, implying that any differences in measured energy use are due to differences in the software. Bunse and Stiemer (2013) used this approach to measure the impact of different *design patterns* (Fowler and Beck, 1999) on a software application. Zhang and Hindle (2014) compared multiple versions of a software application to determine changes in energy use as the applications evolved. Pereira et al. (2021) compared implementations of a collection of computer benchmarks in different programming languages to investigate the languages' energy-efficiency.

In practice, it is impossible to run the test scenarios for two different software options on the same hardware at the same time. The comparison researchers cited above all opted to control the hardware and configuration but run experiments at different times. Computer systems contain internal processes that vary with time and changing environmental conditions such as temperature can also be a factor, so there will always be some variability in the results. This variability is typically mitigated by taking multiple measurements and averaging (Zhang and Hindle, 2014) (Pereira et al., 2021).

The majority of such comparison literature evaluates energy usage of software for single-user desktop or mobile platforms rather than the large-scale services that predominantly consume energy in datacenters.

2.12 The Scale of the Problem

The modern world contains a vast number of electronic devices. There are even thousands of computers in space, as they form a key part of every modern spacecraft and artificial satellite (Eickhoff, 2011). Increasingly, even small devices, sensors, or appliances include network connectivity and collaborate, both with each other and with more traditional computer resources, to form the so-called *internet of things*. The internet is already by far the largest and most complex computer system ever built, and every addition makes it even bigger (Belkhir and Elmeligi, 2018). From initial research into packet-switched networks at the start of the 1960s, the first long-distance computer link in 1965

and the installation of the first ARPANET node in 1969, the internet has continued to grow (Leiner et al., 1997).

It can be hard to get to grips with the scale of the internet because so much of it is behind-the-scenes, The most visible aspect of the internet is the World-Wide Web (also known as the "www", or just "web"). This may seem relatively simple, because a user can only see at most a few pages at a time, and most people stick to familiar high-profile shopping, social, and informational websites (Similarweb, 2023). The web is a distributed system consisting of many communicating devices.

As originally conceived (Berners-Lee et al., 1992) the web was a purely client-server system. Anyone wishing to find some information would use a textual interface or a web browser such as the original *NCSA Mosaic* software (Strawn, 2014) or one of its conceptual descendants such as *Chrome*, *Firefox*, *Edge*, *Safari*, or *Internet Explorer* (which function as the client) to request and receive information from one or more data storage machines (known as the *servers*). This is still essentially how the web works today, although extra complexity has been added in the decades since the web's initial design.

Estimates vary as to the "size" of the web and many different approaches have been taken to try and come up with an answer, from random IP-address probing (Xing and Paris, 2003) and undisclosed proprietary algorithms (Murray and Moore, 2000) to lurid websites with unclear data sources (Live Counter, 2019). A slightly more trustworthy estimate is provided by Worldwidewebsize (2024) which at least explains its methodology and data sources (Kunder, 2008). This website (at time of writing, 18 March 2024) reports "at least 5.22 billion pages". It is important to note, however, that Worldwidewebsize (2024) uses a mechanism based on search engine results and is therefore unable to count unindexed pages such as private ("intranet") sites, databases, and the so-called "dark web". The real figure is likely to be much larger. A different 2023 survey reported just over a billion websites running on a little over 12 million servers (Netcraft, 2023) but made no claims about the number of web pages on each site.

In addition to being *big*, the web is also very *busy*. There are billions of client devices and many web servers handle millions of requests per day (Cisco, 2020). The growth of the *internet of things* is increasing this even more as sensors and other "smart" devices send and receive data without ever involving a human being. Although often described naïvely as "the cloud", the infrastructure of the internet is very physical. Every web request and response passes through a large network of devices, including routers, switches, caches, transmitters, receivers, and signal boosters. Each of these contains electronics, and the great majority of them also contain software.

A key to understanding the enormous numbers involved is that there is a *lot* of duplication. Many web requests are for the same popular pages, and many pages are very similar to others. The well-known Amazon.com shopping website, for example, currently offers more than 350 million products (SellerApp, 2023). To make this possible, Amazon uses a set of standard structures for its product pages, with each page differing only in details related to the specific product. The rest of the page is largely the same, with the same branding, layout, headers and footers and so on. Many very high-traffic websites need to run multiple servers serving the same website to handle the load, resulting in even more duplication.

Although an increasing amount of the modern web involves some code that runs on the device with the web browser (known as client-side processing), the great majority of the work of the web is done on the servers that provide the web browsers with their data. Servers are responsible for listening for requests from browsers using the HTTP protocol (Nielsen et al., 1999), or the more secure but otherwise similar HTTPS, and serving the correct responses. This process can be as simple as generating an error message or locating a stored file and sending the contents in response, or it can involve complex textual and numeric processing, reading multiple files, and collecting data from other servers and databases, before combining the result into a web page or other document ready to return to the client browser.

The huge scale of the internet results in a correspondingly huge consumption of energy. ICT devices and systems require a variety of forms of energy to operate, including specific voltages to run microcircuitry, power to move and spin mechanical parts, and electricity to run cooling fans, illuminate displays, and communicate with other devices. Active web servers are usually located in datacenters, where they have access to the physical security, power, cooling, and connectivity that they need. A single datacenter will often consume on the order of tens of megawatts. This would be enough to power thousands, or even tens of thousands, of homes (Law, 2022).

Electrical energy consumption contributes to global warming in several ways. The most direct is the generation of heat when the electricity is used to perform work. This heat is released into the surrounding environment causing localised warming. Most ICT equipment works best within a relatively narrow temperature range, and if the temperature around the equipment gets too high it needs to be cooled. This in turn requires more electrical energy. While the cooling process may reduce the temperature of the sensitive equipment, it also results in the release of yet more energy as heat into the external environment. Power distribution in a datacenter is usually centralised, which makes it a potential point of failure. Without power, none of the servers in the datacenter

would be able to operate. Servers need to run continuously, which means that a datacenter will usually provide additional power systems in case of problems. These additional power systems also consume electricity and release heat, as does the general operation of the datacenter including things such as lighting and security.

The proportion of incoming energy used by infrastructure such as cooling, power conversion and transmission, lighting, security and so on is described using a power usage effectiveness (PUE) metric. The minimum value for PUE is 1.0, which would indicate that the datacenter used no energy for anything except the computing and data storage systems. A 2016 survey reported typical datacenter PUE values in the USA of 2.0 or greater (Shehabi et al., 2016), indicating that more energy was being used by the operation of the datacenter itself than all the IT resources combined. Power transmission from power stations to data centres also loses energy (as heat) along the way, so only a proportion of the power produced by the power station is available at the data centre. As can be see in Figure 2.12.1, less than 20% of fuel energy typically reaches the servers (Zhao et al., 2016).

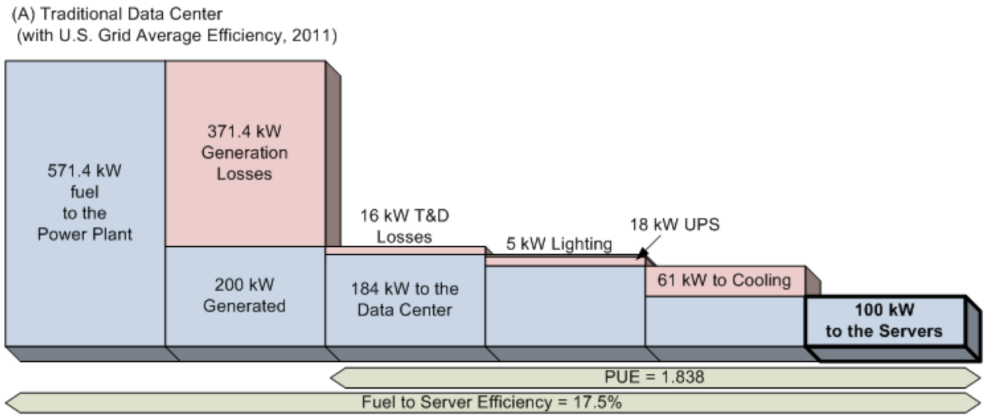


Figure 2.12.1: Traditional data center system losses (from Zhao et al. (2016))

Any power generated by burning fuel, either in a large power station or in a local generator, contributes greenhouse gases such as carbon dioxide to the atmosphere. Greenhouse gases accumulate in the Earth's atmosphere and contribute to the greenhouse effect. These emissions are more serious than localised heating effects, because they are cumulative. If a datacenter were to be shut down immediately, localised heating effects would die down relatively quickly and the local temperature would return to its normal state. The greenhouse gases, however, will remain in the atmosphere and continue to affect heat retention for much longer, potentially for tens of thousands of years before biological and geological processes absorb or reduce them (Pierrehumbert, 2019).

Potential improvements and their effectiveness will be explored in more detail in later chapters, but the key point to note is that all this energy usage and especially all the duplication and losses along the way imply that the field is ripe for change. Small changes in the energy use of key software can have a greatly magnified impact on waste heat and carbon dioxide emissions. Even something as small as a saving of one milliwatt per request on a page that is requested a million times a day, could save a kilowatt per day, every day in server energy usage. Taking into account typical datacenter PUE values and transmission losses, this tiny software change could potentially save over 6kW of power generation and its associated greenhouse gas production. The same logic applies to a page fragment or code component that is used on many different pages. A tiny saving in energy per fragment will result in a much larger overall saving when multiplied by the number of pages that use that component and the number of times those pages are requested.

2.13 Selection of Components to Investigate

In the early days of computing, the emphasis was largely on numerical calculations. The term "computer", was originally a job title for someone who performed calculations rather than a machine (NASA, 2016). Despite this, there has always been a strand of computing that deals with text. Some of the earliest electronic computers were used for code-breaking of textual messages (Copeland, 2004). Almost every modern use for computers has a need to produce textual output, even if just for error messages or activity logs.

Text output may be produced in a range of ways. The simplest is probably direct output of "strings", quoted blocks of text hand-crafted by software developers, included in the text of a program. Another option is loading and output of pre-generated blocks of text from storage such as a file or a database. Some text output, however, needs to include information that was not known when the software or the stored data was created. In such situations what is needed is a combination of fixed text and variable content: "a dynamic and computational text" (Rodgers, Huxor, and Caldwell, 1999). As an example, most business software comes with some sort of "mail merge" facility to produce form letters with a bit more individuality than "*Dear customer ...*".

One way to produce such dynamic text is to generate it in stages, creating a sequence of fixed and variable parts that have the appearance of a single document. This approach is typically done in-line within the software, which can make it relatively simple to create but generally fast in execution. However, text generated

in this way can be difficult to maintain as it requires updating the software for every change, and including large blocks of static text in program code can be cumbersome. In-line text generation is usually restricted to small amounts of text or single-use software.

The most common method for producing these kinds of documents uses a *templating* technique. A *template* is a document containing both fixed text and special symbolic tokens, known as *placeholders* (Arnoldus, Bijpost, and Brand, 2007). To generate a document a template is processed using a type of software known as a *template engine*, which replaces the placeholders with stored or calculated data to generate individualised output documents.

Templating is not limited to personalised mail. A similar approach produces a "substantial fraction" of the visible pages of the web (Yang, 2008). In 2005 Gibson, Punera, and Tomkins estimated that

> Templates represent 40–50% of the total bytes on the web, and this fraction continues to grow at a rate of approximately 6% per year.

Gibson, Punera, and Tomkins's study is relatively old in technology terms. The estimated growth has clearly not continued, or by now there would be more templated content than the web itself! However, the enormous growth of the web since that study has made the determination of such estimates considerably more difficult particularly if attempting to use the technique employed by Gibson, Punera, and Tomkins, which relied on statistical analysis of a "snapshot" of *the whole web*. Despite examining more than 250 papers that cite Gibson, Punera, and Tomkins, up to and including 2023, no better estimates have yet been found.

Templated text generation is also common in much of the less visible internet traffic. Emails and other textual messages, logging, diagnostic output, code generation (Fritzson et al., 2009) (Arnoldus, 2010) and a wide variety of formats and protocols (Barbosa et al., 2002), all benefit from this powerful technique.

The internet is so large and so complex that even determining its scale is impossible for practical purposes (see Section 2.12) Attempting to determine the exact proportion of web pages generated by templates is similarly difficult. However, *WordPress*[13] (a template-based web site management application) is estimated to run approximately 50% of the world's websites (W3Techs, 2022). Proceeding from this and Yang's "substantial fraction" in conjunction with Kunder (2008)'s web size estimate of "at least 5.22 billion pages", it seems fair to

[13] `https://wordpress.com/`

assume that template-generated web pages number at least in the hundreds of millions, and possibly billions.

Public website visits range from single figures to hundreds of thousands or even millions per month (Castillo, 2023). This implies that the total number of template pages generated is probably at least trillions per month. The impact of even small differences in the performance and energy usage of the template engine software could therefore be huge.

Templating technology is therefore a category of software component with characteristics that suit the overall objectives of this research. This technology is widely used on the web to process large numbers of page requests, so even small improvements can potentially have a large impact on the sustainability of the web as a whole. There are many different templating components available with widely differing properties, making this a suitable category for investigating the viability of component substitution as a way to improve sustainability.

2.14 Templating in the Literature

In the literature, templating is often mentioned in passing, either from an abstract viewpoint (such as seminal works on the philosophy of information by Bush (1945) and Nelson (1974)) or as a means of achieving a specific end, such as code generation (Drescher and Engelke, 2024), education (Goetz and Marquez, 2023), or as part of a more general knowledge engineering process (Caldwell, 1998). However, while these may contain *some* information on the selection and use of templating systems, they are not the core of this research.

The pool of literature that directly addresses the theory and practice of templates and template language appears to be much smaller and mostly limited to a particular period of time in the 2000s and early 2010s, during which such research seems to have been more popular. Citations to the papers mentioned below typically dry up around 2015, although there have been a small number of publications on this topic in more recent years.

Vosloo and Kourie (2008) set out to survey and classify the landscape of server-side web page generation frameworks. Tellingly they were unable to find other literature at this level of detail at the time.

This proliferation of discussions and solutions may be construed as

> an indication that the problem (of finding the right abstractions with which to implement Web-based UI) has not been solved satisfactorily in practice. However, it may also be that the problem merely has a great number of variable parts, and that it needs to be partitioned more usefully. (Vosloo and Kourie, 2008)

Vosloo and Kourie initially divide the web page technologies they examine according to the three aspects of the popular Model View Controller (MVC) pattern. Their research is primarily concerned with what they describe as "view concerns". They consider possible taxonomies of page-generation strategies including various forms of templates, eventually presenting the diagram shown in Figure 2.14.1.

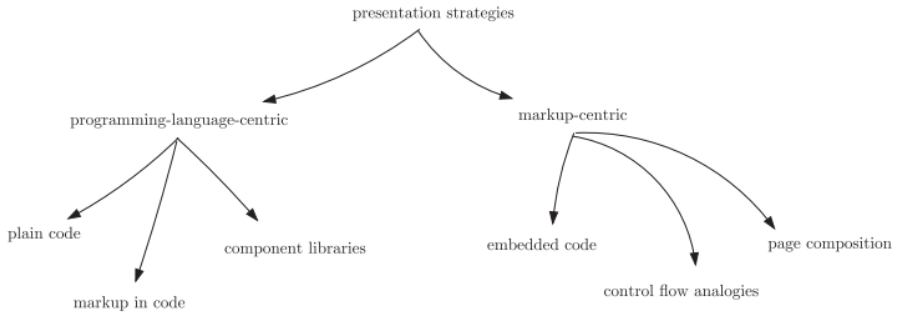


Figure 2.14.1: A taxonomy of strategies for view concerns from Vosloo and Kourie (2008)

The final paragraph of Vosloo and Kourie's "view concerns" section highlights the usefulness of taxonomy construction in finding under-considered areas:

> For example, none of the 80 frameworks studied relied on a template language with the same basic goal of the page composition variants (Section 5.8), but with a syntax external to the host markup language. ... The absence of such a category in the aforesaid taxonomy could perhaps be the trigger for developing just such a language. (Vosloo and Kourie, 2008)

There are, however, two key issues with Vosloo and Kourie's approach. The first is that only server-side web frameworks were considered for the study. While this does not invalidate their results, it does imply that the classification work would need to be re-done if page-generation tools in other technologies, or template techniques for non-web uses were to be included.

Another issue is a more general one with all such taxonomies. As pointed out by Usman et al. (2017) "most taxonomies are developed in an ad-hoc way", and "taxonomies are rarely revisited, revised or extended". New server-side frameworks and template technologies are continually being produced, and existing ones are continually being revised, updated and re-released. An ad-hoc summary from nearly twenty years ago is likely to be neither exhaustive nor fully correct and should be considered as a historical source rather than a complete reference. Even with this caveat, though, Vosloo and Kourie's list of 86 server-side web frameworks, most of which have at least some template processing features, begins to show the scale of the problem.

Laakso and Niemi (2008) also attempt to categorise a selection of web frameworks. As seen in Vosloo and Kourie (2008), web frameworks often have template features. Laakso and Niemi (2008) also have trouble finding prior work on which to base their research:

> To the best of authors' knowledge, very little research has been done about measuring web application frameworks. (Laakso and Niemi, 2008)

Laakso and Niemi (2008) do, however, recommend two articles: Zoio (2005) and Kolesnikov (2006). Zoio presents a detailed and practical comparison between two frameworks: *JSF*[14] and *Tapestry*[15] through the approach of building the same application twice, once in each of the two competing technologies. Kolesnikov describes the use of *Tapestry* in an MSc dissertation.

Laakso and Niemi (2008) take an experimental approach, as does Zoio (2005). Both design one or more scenarios, apply them to each of the implementations, and use the results to measure the suitability of different software systems. The comparisons generally concentrate on ease of development and deployment, although there is some mention of rendering performance, which may be analogous to efficiency, and thus affect resource usage.

Concentrating more specifically on templates, Parr (2004) sets out to

> formalize the study of template engines, thus, providing a common nomenclature, a means of classifying template generational power, and a way to leverage interesting results from formal language theory. (Parr, 2004)

[14] `https://www.oracle.com/java/technologies/javaserverfaces.html`
[15] `https://tapestry.apache.org/`

Parr, however, presents the opinion that a template is a pure and separate view and should be "totally divorced from the underlying data computations whose results it will display" (Parr, 2004, p2) While this is arguably a justifiable position, it does lead to a taxonomy that essentially has two categories: those which enforce such separation, and those which do not. The fully separated category mainly consists of a new template language, *StringTemplate*[16], developed alongside the paper, with almost every other solution relegated to "do not". Parr also limits his discussion to template-style solutions written in the Java programming language.

Although Parr's *StringTemplate* language has been used for other academic publications (Fritzson et al., 2009) (Arnoldus, 2010) (Hartmann, 2011) (Arnoldus, Brand, and Serebrenik, 2012) (Vollebregt, Kats, and Visser, 2012) and many more, Parr admits that commercial programmers often prefer the features of other, less pure but more powerful, template languages. Implicitly, therefore, there is still a need to categorise these other template languages beyond simply whether or not they enforce separation.

Arnoldus (2010) also addresses the language used in templates, but from a grammatical perspective. The important categorisation distinction in Arnoldus' work is that of *safety*. Having determined that "Writing templates, and code generators in general, is a complex and error prone task" (p103), Arnoldus introduces the concept of a "syntax-safe" template, which is incapable of generating grammatically incorrect output. This is particularly important when the output is intended to be machine readable, with a formal grammar, and Arnoldus concentrates on the specific application area of program generation, which exhibits these characteristics. Although Arnoldus does not attempt a categorisation of template languages, he does discuss several key factors (such as the presence of "hedges" (p111), separating placeholder expressions from other text), which are commonly found in those template languages that are more suitable for syntax-safe applications.

More recently, Mittapalli and Arthur (2021) conducted a survey of template engines implemented in the Java language. In their survey, they select six examples of templating technology. Four of the examples are general-purpose template engines: *Velocity*, *Freemarker*, *Mustache* and *Thymeleaf*, all of which are also included in this research. The other two template technologues are *Tiles*[17] and *Groovy*[18]. *Tiles* was originally a technology for constructing "portal" style pages consisting of separately defined panels (the "tiles" in the name) so is not directly comparable with other general-purpose template engines. *Tiles* was

[16] `https://www.stringtemplate.org/`
[17] `https://tiles.apache.org/`
[18] `https://groovy-lang.org/`

retired in 2017. *Groovy* is a complete programming language which compiles to the Java Virtual machine, much like the Java language itself. *Groovy* is much more than a template engine and, again, not directly comparable with other general-purpose template engines. Mittapalli and Arthur give no justification for why they selected only this particular subset of the available Java template technologies.

In addition to literature listing, comparing, and categorising template engines and template technology in general, there have also been publications criticising the technology.

Carvalho, Duarte, and Gouesse (2020) utilise the classic formulation to declare "Text Web Templates Considered Harmful", but the title is a little disingenuous. The authors start by pointing out some well-known issues with existing template engines and template languages such as incompatible placeholder syntax and logic features (discussed in Section 4.2 and addressed in Appendix E), the need to pre-calculate values for template engines without field access (discussed in Section 4.2.7), and how some template engines lose performance by needing to fully re-interpret each template every time it is used (discussed in Section 4.4.1). However, rather than address these issues directly, the authors proceed to recommend instead a selection of libraries for constructing HTML documents from within program code: *HtmlFlow*[19], *Kotlinx.html*[20] and *React JSX*[21]. They then proceed to run some simple performance comparisons (similar to those in Section 4.3) that show the pages constructed in program code as typically faster than the template engines in the comparison. However, there are some problems with the recommendations in this paper.

- The authors do not mention any of the specific drawbacks of creating pages in program code compared to using a template engine such as the need to edit, re-compile, re-test, and re-deploy the code to make any page changes, the incomprehensibility of such code pages to graphic designers and copywriters, or the inability to store or share templated pages using files or a database.
- The authors also do not include some of the fastest-performing template engines (such as *Solomon* and *JMTE*) in their performance comparisons. They test only *Freemarker*, *Handlebars*, *Mustache*, *Pebble*, *Thymeleaf*, *Trimou*, *Velocity*, and *Rocker*.
- The authors do not explore further beyond simple single-scenario

[19] `https://htmlflow.org/`
[20] `https://github.com/Kotlin/kotlinx.html`
[21] `https://legacy.reactjs.org/docs/introducing-jsx.html`

performance tests.

Pisu, Maiorca, and Giacinto (2024) are more specific in their criticism of template technology, focusing on a selection of potential security vulnerabilities. They begin by estimating the number of available template engines (using a GitHub search for "template engine" similar to that used in Section 4.1.1) and then examine a small subset for their vulnerability to two specific security issues Remote Code Execution (RCE) and Server-Side Template Injection (SSTI). RCE vulnerabilities can exist in template engines that support code execution through explicit method calls in templates or implicit method wrappers such as JavaBeans. The paper then explores techniques to test for the presence of these vulnerabilities and potential ways to prevent them. The security of template technology is clearly an important and under-explored area, much like sustainability, but is out of scope for this research.

2.15 Discussion

Templates are the technology of choice for a large proportion of the world's web pages, and thus it seems plausible that the generation of these pages contributes in some part to the large energy and resource usage of the internet and the world wide web.

The consideration of the design, implementation, and analysis of template engines and template languages in the academic literature is small compared to the amount of literature that mentions the use of templates as a means to another end. Such consideration is also largely limited to a relatively short time period, even though templating as a technology has existed since the early days of computer textual processing and still continues today. This short period of research does not imply that the study of template systems is in any way complete or obsolete, merely that researchers in this area have moved on to other topic areas or have taken their expertise to careers where academic publication is not required. The quantity of such papers is dwarfed by the number of software implementations, websites, and documentation that have not been through any form of academic peer review process.

Of particular significance to this dissertation is that the decline in research on template technology occurred along with the rise in interest in the sustainability of software. This indicates that the overlapping area of the sustainability of template engines and template technology in general remains under-researched.

Template engines and template languages vary widely. Attempts have been made to construct taxonomies in related fields such as web applications, and crowd-sourced information is available from sources such as Wikipedia[22], but so far no detailed and comprehensive comparison and analysis of a wide range of template engine implementations has been found. In the context of this research, so far no taxonomies have been found that include or mention energy consumption, energy efficiency, or resource usage in general.

It would seem, therefore, that there is an opportunity for research in this field to contribute to knowledge and thereby potentially assist in improving the overall resource usage of large software systems.

2.16 Research Scope

The above literature analysis suggests a research gap in the area of energy sustainability of server applications and components. More specifically, an exploration of ways to reduce the energy usage of web systems by comparing and selecting software applications or components for energy efficiency. This research gap sits within the second category mentioned in Penzenstadler (2013) - the sustainability of the computing systems themselves. However, this is still a very broad area.

The following sections clarify the scope of this research.

2.16.1 Exclude computing *for* sustainability

The area of computing sustainability that seems to have received the widest coverage in the literature is what Penzenstadler (2013) refers to as "*for* sustainability" (see also Section 2.1). This is a broad area that can potentially address any of the UN Sustainability Goals and which is characterised by the use of computer technology to achieve external aims. A typical example, picked largely at random, is Gwaka (2022) which investigates the potential of digital platforms to revitalise a livestock system in rural Zimbabwe. However, research in this category often has a significant oversight. It is common in research for sustainability to treat the computing resources required to investigate or address an issue as effectively free and without significant sustainability impact. This attitude is endemic in computing research *for* sustainability and is not to be taken as a specific criticism of Gwaka. While most research in this area is

[22] `https://en.wikipedia.org/wiki/Comparison_of_web_template_engines`

completely out of scope for this dissertation, there is an aspect of this kind of research that needs further consideration as it calls into question the sharp distinction between Penzenstadler's two categories. This grey area is software which is applied to improve the sustainability *of* ICT systems or the infrastructure on which the ICT systems depend. Examples of this area are discussed in Verdecchia, Lago, and Vries (2022) and to some degree in all the experimental literature included in Section 2.10. The software systems developed later in this dissertation also sit in this grey area.

2.16.2 Exclude sustainability of computing hardware and infrastructure

Although it is the computing hardware and datacenter infrastructure that uses energy for its operation, and the physical constituents of such systems that enact the environmental impact, there is other research addressing these issues. The main focus of this research, however, is on the software systems that control the hardware and drive data through the communications networks.

2.16.3 Exclude client-side and network software

Software exists everywhere on the internet. Software systems can be classified into groups based on the role they play in its operation. Note that these groups are only rough, as some devices or systems incorporate aspects of multiple groups.

- Single-user "client" systems such as phones, tablets, laptops and other computers, televisions, interactive signage etc.
- Sensors, data input devices, and other Internet of Things (IOT) "edge" devices.
- Network systems such as routers and switches, access points, network storage and caching devices.
- Service systems such as web servers, application servers and database servers as well as the shared applications that use them.

The energy impact of all these systems is important, but this research will concentrate on the energy impact of service systems. Energy management is already a high priority in battery-powered devices such as phones, tablets and some IOT devices. Single-user systems such as phones, tablets, and desktop and laptop computers are not subject to the same multiplication factors as

multi-user web-facing systems. Network systems and edge devices are generally included in the power management of the internet infrastructure, which has already been de-scoped from this research. That leaves service systems that fit the criteria for systems that are in continual use by multiple concurrent users and therefore offer large potential gains from a reduction in energy use.

The distinction between different types of service systems is less clear, with many Web-facing systems operating as some combination of web servers, application servers and database servers as well as managing storage and internal communication and executing application-specific software. This research concentrates on server systems that contribute in some way to high-volume public web traffic and datacenter energy use.

Note that although other forms of software are not directly investigated in this research, some of the outcomes of this research may also be applicable to those kinds of software.

2.16.4 Exclude operating systems and virtual machine systems

Operating systems and virtual machine systems are software constructs, but share some of the characteristics of both infrastructure and applications. As with network software, some aspects of this research may also apply to these kinds of systems, but such infrastructural software is not the primary scope of this research.

2.16.5 Exclude sustainability of software development tools and processes

As discussed in Section 3.2, the practice of software development makes use of a lot of software itself. This kind of software follows the general principles of all software. Development tools are themselves created with development tools. They make use of libraries and components and embody the decisions made during their creation and maintenance. The operation of software development tools itself uses energy (Zaidman, 2024). However, Wahler (2024) wrote “We conclude that the sustainability of the operations phase (in particular, the sustainability of the software product) needs to be considered together with the sustainability of the development process to see the larger picture”. The scale of operation of development tools is typically much smaller than that of internet applications. This research concentrates on the kinds of software that are used by millions of users every day and drive the major energy use of datacenters and communication

infrastructure.

2.16.6 Exclude theoretical efficiency of algorithms

Potentially the oldest field of research related to software energy usage is the study of the efficiency of algorithms. Algorithm efficiency is commonly expressed using *Big "O"* notation such as $O(n \log n)$. It is well understood that a sorting algorithm categorised as $O(n \log n)$ will typically perform more efficiently than one categorised as $O(n^2)$, for example. This is a valid theoretical field of study, but for many working software developers it is only of peripheral interest, as the algorithms typically studied in this way have already been coded into components, libraries, programming languages, and other development tools.

Most developers never need to make design decisions at this level, but there is a hidden catch to this layer of abstraction. In order to use libraries and language features, a developer needs a degree of faith in the underlying implementation choices. When using, for example, a sort function from a library, the developer either needs to trust that the implemented algorithm is suitable for the intended use, or perform some extra steps of selection, configuration, or performance testing. In many real-world cases the pressure of commercial development means that the faith approach is the most attractive. This research is not directly concerned with the efficiency of specific algorithms or data structures.

Although the theoretical analysis of algorithms is out of scope for this research, there is an important lesson to take from such studies. The amount of work done by software depends on the size and complexity of its input data. An approach that is "better" for small or similar input data may be "worse" for large or varied input data or vice versa. This lesson is sometimes overlooked when attempting to evaluate or compare software solutions, and is explored in more detail in Section 4.8.2.

2.16.7 Specific Research Focus

Even after the above exclusions, this research area is still too broad for a single PhD. This research will focus on the following specific areas:

Static and Dynamic Websites The original aim of the web was for the storage and sharing of documents (Berners-Lee et al., 1992). In some sense this is still true, as each GET request of the HTTP protocol transfers a document from a server to a client (Berners-Lee, 1996) (Nielsen et al., 1999). It is then up to the

client software to render that document appropriately for whoever or whatever has requested it. There is, however, an important distinction in where the document comes from.

In a *static* website, all documents, images and other resources are created ahead of time. When a GET request is received, the task of the server is to locate the appropriate resource and send it to the client. In most cases this is a relatively simple process, involving little more than decoding the requested URL into a file path and replying with the contents of the file at that location, or an error if nothing is found. This process is complicated a little by the need to also return an appropriate *Content-Type* header so the client software knows what to do with the data, but this is usually a simple look-up based on part of the filename.

In a *dynamic* website, some or all of the documents do not exist before the request, but are created when the request is received. Dynamic documents can be created in a wide range of ways, ranging from concatenations of existing files to programmatic approaches using complex calculations, data from databases and even requests to other systems. This wide range of possibilities makes it hard to reason about dynamic websites as a single concept, except to say that extra complexity can require both more time and more energy to serve.

This research includes a comparison of the energy usage of static and dynamic websites serving the same data in Section 5.5.1.

Page Templating Systems As discussed above, there are many ways to dynamically generate a document to respond to a server request. This research will concentrate on one popular method, *templating*. This process uses a single template document to construct many different response documents. A template consists of two kinds of data - fixed parts, known as boilerplate, which will be the same for all generated documents, and variable parts, indicated by placeholders, which can differ based on data and values that are specific to that generated document. When a templated page is requested, the template and variable data are passed to a template engine that combines them into a final document to be returned to the client.

Templating could, for example, be used on a shopping website such as Amazon.com. Each product page has the same basic structure, but the product images, descriptions, prices, and so on are specific to each product page. When the page for a specific product is requested from the server, the data for that product is fetched from a database and used to populate the variable parts of the template, and then the combined result is returned as the response.

The Java Programming Language There are a large number of programming languages, and testing a meaningful sample of template engines in all of them is not feasible in a single research project. For practicality, a single programming language needed to be selected. The TIOBE Index (TIOBE, 2024) is a resource used by many software practitioners and tracks the popularity of different programming languages over time. At the start of this research, the Java language was clearly the most popular and remained so until mid-2020. At this point in the research, the planning, feasibility study and selection of software components had already taken place (see Section 1.3).

However, it is important to note that the Tiobe index is based on web search results (TIOBE, 2025) so arguably tracks some indicator of "interest" (through scanning documentation and discussions) rather than measuring which software and tools are actually being used by software developers or which software is running on the world's servers. Java has been in the top 5 on the Tiobe index since 2000 (TIOBE, 2024) and has been a popular language for "server-side" applications for most of that period. By implication, there has been a lot of server software built using Java in the intervening period, much of which will still be serving requests.

For the purposes of this research, Java also offers a rich selection of page templating components to choose from (see Appendix A).

2.16.8 Scope Summary

- Reducing the energy usage of web systems.
- Software in the Java programming language
- Performance and energy usage of dynamic web components under different usage patterns
- The impact of replacing components with more energy-efficient ones

2.17 Research Objectives

The scope above led to several related research objectives, as introduced in Section 1.2.

2.17.1 Investigate the context of web software and how it is developed

Any attempt to reduce the energy usage of software systems needs to be compatible with the context in which those software systems are used, the way software is developed, and the forces that cause software to be constructed and tested in the way that it is. This research objective was addressed through both a literature study and a personal investigation. The personal investigation drew on my experience and the experiences of colleagues working in a wide range of software development positions and organisations. This research objective is addressed in Chapter 3.

This research objective was approached by exploring one of the key factors in any attempt to improve the environmental impact of web software, the scale of the internet (Section 2.12). Although computing is a new field compared to the natural sciences or even to engineering in general, software design and development is still influenced by the history of the field (Section 3.1). The process of modern software development was then explored (Section 3.2) followed by investigating the forces affecting software development (Section 3.3), the roles taken on by team members during software development (Section 3.4) and the range of developer choices that influence the resulting software systems (Section 3.5). Then what the computing industry is doing about environmental issues was explored (Section 3.6) and, finally, the results of this investigation were summarised in Section 3.7.

During the research for this objective, it became apparent that the skills and knowledge of software developers are key factors in how software is developed. A subsidiary study was conducted to understand the presence and nature of sustainability teaching in higher education in the UK. That study is not included in this dissertation.

2.17.2 Explore the differences in performance of a selection of software components

One potential approach to reducing the impact of computing systems is to replace all or part of these systems with alternatives that do the same job but consume less energy and thus contribute less greenhouse gases to the environment. However, simply replacing components without understanding the differences between them is unlikely to provide the most improvement. The selection of alternatives requires detailed information on the characteristics of available components, but this information is usually not available. One such

characteristic is performance. The selection of components with better performance can allow systems to run with fewer, or less powerful, computer platforms, and thus reduce the overall environmental impact of the systems. This research objective is addressed in Chapter 4.

This research objective was approached by first selecting a representative category of components (Section 4.1) and exploring the differences in features and usage among a range of components in the selected category of template engines (Section 4.2). Based on this understanding, a feasibility study was undertaken to explore the potential range of differences in template generation speed between components (Section 4.3). This feasibility study provided some interesting results (Section 4.4) but also revealed some methodological flaws and opportunities for improvement (Section 4.5).

The problems with the feasibility study were addressed in an improved performance study (Section 4.6). The landscape of template engine components had changed in the time since the initial feasibility study (as explained in Section 1.3), and a more systematic selection process was employed to select template engines for comparison. A new framework for interchangeable components was also developed to isolate components during measurement by dynamically loading them for the execution of the test (Section 4.7). The original set of test scenarios was then replicated in the new framework (Section 4.8). Once the basic function of the improved performance framework had been established, a progression of "measurement sets" was performed (Section 4.9) and analysed (Section 4.10).

Discussion of the performance experiments can be found in Section 4.11, and some conclusions are drawn in Section 4.12.

2.17.3 Construct a test apparatus to compare the energy use of software during operation

Software component performance is not the only, or even the most direct, measure of the possible energy saving from substituting software components. Measurement and comparison of the energy usage of software during operation can potentially provide a more explicit indication of relative energy usage of components and systems in live use. However, measurement of energy use during operation requires a test apparatus. This research objective is addressed in Chapter 5.

This research objective was approached by first considering existing approaches

(Section 5.1), then deciding on the requirements for the apparatus (Section 5.2). The hardware and software for a prototype apparatus were designed and constructed (Section 5.3). The constructed device was then validated by comparing a selection of popular web site server software (Section 5.5). The limitations and drawbacks of the apparatus were considered (Section 5.7) and possibilities for improvements and future research were discussed (Section 5.8) before finally drawing conclusions about the apparatus in Section 5.9.

2.17.4 Using the test apparatus, compare the energy use of common web software and the feasibility and effectiveness of substituting template engine components

The experiments in Chapter 4 discovered significant differences in performance between the template engine components. Although this information could potentially be used to select components that need fewer hardware resources to handle a particular load, it does not directly say much about potential differences in energy consumption. This research objective is addressed in Chapter 6.

This research objective was approached by first investigating the differences between the platform used to run the improved performance tests (a standard Intel-based laptop) before the apparatus was constructed (see Section 1.3) and the Raspberry Pi device used as a DUT in the measurement apparatus (Section 6.1). Once the device differences had been investigated, an experiment was planned to measure the power use while running the improved performance tests (Section 6.2). The results of this experiment did not provide enough data to draw reliable conclusions about the energy use of the components, so a further experiment was planned that collected more samples and averaged the readings over multiple runs (Section 6.3). The second experiment showed clear differences between the energy usage of the different components during the tests, but there remained questions about how well the simplistic nature of the templates used in the performance tests represented real usage. A further experiment was planned to compare the energy use of the components when used to generate a more complex and realistic web page (Section 6.4).

The outcomes of the experiments that contributed are discussed in Section 6.5 and some conclusions are drawn in Section 6.6.

During this research, an intermediate data format and a software tool to assist in the generation of appropriate data files for a range of template engines was developed. This format and tool are described in Appendix E.

Chapter 3

How Web Software is Developed

The emphasis of this research is on the development of software, so the scope of this research will be limited to devices and systems that include software as a key element. Software is more pervasive, and more complex, than many people think (Kang et al., 2015), though, with software controlled devices ranging from the barely visible and commonplace to multi-million-dollar and planet-wide systems.

For the results of this research to be adopted and make a difference, they need to be applicable in the context and situations encountered by software developers involved in working with real software systems that consume energy and resources, producing greenhouse gases and waste products.

This chapter explores the context of software development and the forces that shape how software development is done in order to guide the research in the following chapters.

3.1 The History of Computers and Software

Electronic computing is still a very new field compared to other areas of science and engineering. The first programmable electronic computers generally accepted as such, Colossus and ENIAC, were developed in the 1940s (Campbell-Kelly et al., 2023), still less than 100 years ago. Software, as we might recognise it, came somewhat later. Early computers were distinguished by being large, expensive, often unreliable, and very rare. Software was usually created from scratch for each new need. Initially there were no programming languages or code libraries so each programming task involved designing a solution and laboriously entering it into the computer memory as ones and

zeroes. Since then, computers have gained persistent storage, compilers to create the ones and zeroes from readable programs, and a proliferation of programming languages. Depending on definitions there are currently between hundreds and thousands of different programming languages (Pigott, 2020). The introduction of the web, the browser software to access it, and a slew of data and communication standards, meant that most new computers and operating systems had the ability to participate in a large scale distributed system. With the resources of the web and the communication abilities of the wider internet, collaborating on software development became much easier, leading to the rise of open source, and the spread of free (to use, at least) software libraries and components.

Software in the 2020s requires a huge amount more resources even to do what is arguably a similar job to that of previous generations. For example, *Wordwise* (Unknown, 2023) was a top word processor for the very popular BBC Microcomputer throughout the 1980s. The software was delivered on an 8 kilobyte Read-Only Memory (ROM) chip, and allowed users to write, edit, and print letters and other documents using just 32 kilobytes of memory on a single 8-bit processor running at a clock speed of 2 megahertz. The 2020s equivalent would be something such as *Microsoft Word*, which is usually purchased as part of a subscription to *Microsoft Office*[1] (also known as *Microsoft 365*). Depending on version, this software can take up to around 10 gigabytes of storage space. That's roughly a *million* times more than the 1980s equivalent. Even just the basic Windows 11 operating system can require over 20 gigabytes of storage. A typical 2023 personal computer might have an 8-core processor running at 2 gigahertz. That's roughly *eight thousand* times the processing power of the 1980s word processor.

This explosion in code size is not limited to desktop applications. Web server and web application software is far from simple. As of June 2023, the three most popular web server applications were *NginX*, *Apache*, and *Cloudflare* (Netcraft, 2023). NginX and Apache are open source software so the details of their code are available online. NginX has roughly 250,000 lines of code (Openhub, 2023b) while Apache has nearly 1,700,000 lines of code (Openhub, 2023a). Cloudflare is proprietary software, so details are sparse, but it was apparently based on the NginX codebase, so is probably roughly similar in size.

Of course, people have much higher expectations of modern software, but the facts remain that the single unwavering trend throughout the development of computers and software has been inflation, both of software size and of the resources needed to run it. The sharp increase in the number of "smart" devices joining the internet,

[1] `https://www.office.com/`

	Internal Need	**External Need**
Non-commercial	Free (open source, "freeware", personal projects)	Academic (teaching and learning or research)
Commercial	Internal (for use in a business)	For Sale (shrinkwrap, download, SaaS or hybrid)

Table 3.1: Software categories by need and finance

the increasing expectations of "cloud computing", and the development of various forms of artificial intelligence will only push this trend even harder (Falk and Van Wynsberghe, 2023).

This increase in system requirements also leads to the obsolescence of existing hardware that becomes increasingly incapable of handling modern software and data systems.

3.2 How Software is Made

It is common in the teaching of computing to repeat the notion that there are three types of software: *system software*, *utility software*, and *application software* (Raspberry Pi Foundation, 2023c) (Gupta and Pedamkar, 2023) (Various, 2023) and many more. This distinction was coined during the relatively early days of the development of computing, when most software was created from scratch in isolation to run on stand-alone computers, and is arguably no longer particularly useful (Hislop, 2009). Modern software is multi-layered and often distributed between multiple machines. Code is shared and re-used at all levels and the boundaries between the three original categories have become blurred and uncertain.

For the purposes of this research it is important to consider the reason for making the software in the first place. For example, software might be created to gain knowledge, to make or save money, to help people, or even just for the joy of it.

Table 3.1 shows an example way of categorising software by two dimensions, based on whether the development is driven by an internal or external need, and whether the development is commercial or non-commercial.

Table 3.2 summarises the characteristics of the four categories shown in Table 3.1 and introduces three more potential dimensions that apply different forces to the software development process: writing software for regulated environments such as

banking, medicine, or aviation; writing software for distributed and web systems; and writing software for embedded systems in hardware devices. These categories are neither strict nor exclusive - examples can easily be found that overlap or blur the distinctions - but they are potentially useful to examine the way that the reason for creating software affects both the creation process and the end result.

Category	Description
Free	
Personal Projects	Produced by an individual for personal purposes such as learning, home use, showing off to friends, etc. Not usually available for others to use. Source code not usually shared for re-use. Can turn into open source or commercial if it becomes popular. No budget or deadlines. Requirements very fluid or non-existent.
Freeware	Provided free of charge, usually by download, but not open source. May include monetisation options such as in-app purchases, support agreements, or voluntary payments such as a "tip jar". Could be considered as "for sale" but for a zero price.
Open Source	Once dismissed as just a hobby, open source applications have been available since the 1970s, but formalised in the late 1990s (Bretthauer, 2001). Open Source components are used in many commercial applications (Androutsellis-Theotokis et al., 2011). Zero price and source-code available. Often created and maintained by a single individual in his or her spare time with no budget or deadlines. Usually made to "scratch your own itch" (Raymond, 1999) (Raymond, 2010). Standout open source projects such as Linux[2], Apache[3], or MongoDB[4] that support a financial ecosystem and employ developers are very rare.
Academic	
Teaching	Needs to clearly express its learning points. Does not need to be complete or even to work. Often restricted to enrolled students or kept private to avoid plagiarism. Not easy to find or to re-use.
Research	Often developed under a deadline and a budget but for research objectives rather than profit. Usually developed by individuals or small teams. Researchers may not be experienced software developers, so learn as they go (Trisovic et al., 2022).
For Sale	

[2] `https://www.kernel.org/`
[3] `https://httpd.apache.org/`
[4] `https://www.mongodb.com/`

"Shrinkwrap"	Installed on the end-user's device. The dominant form of software delivery from the 1980s until the early 2000s. Must meet market expectations of functionality, reliability, and usability (Khoshgoftaar, 2001). Delivers distinct, marketable, versions while minimising the burden of end-user support. Has costs for packaging, distribution, and storage in addition to development and marketing. Only needs to cater to one user at a time. Requires extensive pre-release testing as mistakes are hard to rectify.
Download	The modern form of "shrinkwrap". Still runs on the end-user's device but minimises packaging and distribution costs. Usually has distinct versions, but "bug fix" updates can be released at minimal cost.
SaaS	An increasing amount of software is now "software as a service" (SaaS) (Fan, Kumar, and Whinston, 2009). An application runs on servers and is accessed via a web browser. Updates can be instant, with no release schedule, (known as "Move Fast and Break Things" (Taplin, 2018)). Costs of running the application are borne by the supplier. Able to scale to large numbers of concurrent users but may need multiple servers to handle the load and still remain responsive.
Hybrid	Increasingly popular, particularly for mobile devices. Software is downloaded, but also includes or requires an online service or API. Development is complex, split between multiple codebases requiring different skills and with different release rates. Some hybrid software, for example *Microsoft Office*[5], also known as *Microsoft 365*, includes both a web browser interface and installable local applications.
Internal	For internal use within an organisation to assist in other business functions. An example might be *Work Manager*, a job scheduling tool for service engineers developed by British Telecom (Garwood, 1997). Contributes indirectly to the success of the organisation by improving service and decreasing costs rather than directly providing revenue. The current needs of the organisation determine features, budget and timescales. Considered as a business expense to be minimised rather than a source of income to be maximised.

[5] `https://www.office.com/`

Regulated	Constrained by regulations specific to the kind of software being developed and the intended uses. For example, financial services oversight or safety constraints on medical equipment. Software development that is regulated in this way commonly involves more checks during development, lengthier testing before release, a lot more documentation, and generally slower release cycle.
Distributed	Developing software for distributed systems has its own challenges, particularly when a system is large or has specialist hardware or software requirements and cannot easily be exercised by a single developer. Parts of the system are developed and tested, as far as is possible, in isolation, before integrating the parts into a whole system. The effect of this is to delay feedback to software developers and decrease both development speed and the resulting rate of release of software improvements.
Embedded	Embedded software is developed as an integral part of a physical product and commonly operates at a very low level, often without any kind of operating system, and often on very resource-constrained devices. Development is usually done on a different machine and is cross-compiled for deployment to the device. Sometimes, the software is developed at the same time as the hardware so developers do not have access to a real device until late in the process. Developing embedded software requires detailed knowledge of the hardware and costs to rectify faults after shipping can be very high.

Table 3.2: Software categories in detail

As a final point, it is important to note that things can change. A software product can migrate between these groups as it develops and matures, which will necessarily affect the software development context and processes needed to work with it. As an example, the *Work Manager* scheduling software mentioned above started life as a standalone academic research project (Lesaint et al., 2003), developed into a distributed system and became a key internal product (Garwood, 1997), and was then "spun out" as a commercial online service (Trimble, 2006).

3.3 Forces affecting software development

Software development is not usually done in isolation. Developers produce software systems in the context of existing software and hardware choices and subject to conflicting forces.

3.3.1 Requirements and Expectations

An obvious force on development is the presence of "requirements" and the expectation from stakeholders that these requirements will be met. The classic software life cycle usually presumes the presence of customers and requirements, often decided before the design and programming phase of the product begins. While requirements are often specified in such situations, there is also a considerable amount of software development that happens without explicit requirements in the traditional sense. Personal and hobby projects rarely have requirements beyond "scratching the itch" (Raymond, 2010) of a particular software developer or small team, and research software is often more exploratory in nature and defined by its results rather than its specification. Software developed using one of the many agile processes (Hoda et al., 2017) usually has some form of requirements, although these requirements and priorities are decided during the development process and subject to change rather than established before development starts.

In situations where requirements are present they can have varying degrees of rigidity, from precise specifications that must be followed exactly to loose guidance subject to interpretation. In all such cases, however, the presence of requirements, and the accompanying expectations of customers and other stakeholders exert a force on the software to conform to the requirements and to avoid choices that might prevent those requirements being met. Requirements are generally established using a process of *elicitation*, used to determine the wants and needs of stakeholders.

Traditionally, requirements have been divided into "functional requirements" and "non-functional requirements". Functional requirements relate to the function of a system and can usually be phrased in a way that allows for a yes/no or pass/fail answer. Non-functional requirements often include vague or aspirational aims such as "should be secure" or "should be easy to use". Such requirements are sometimes referred to as "soft requirements", and can be difficult to measure.

Sustainability requirements are commonly included in "non-functional" requirements but are treated differently by different disciplines (Venters et al.,

2017a). For the purpose of this research, maximising sustainability and minimising environmental impact are considered as implicit requirements as recommended by the Green Software Foundation (Green Software Fioundation, 2024).

3.3.2 Timescales

Along with the pressure of requirements comes the pressure of timescales. These come in many forms from single hard deadlines to a sequence of deliverables or progress demonstrations with negotiable dates. The pressure of timescales can be more subtle and insidious than that of requirements, as it encourages the making of quick decisions and provides an obvious penalty to taking too long to research or come to a conclusion. This in turn can lead to choices being made without enough information, an outcome that is especially likely in situations where that information is both slow and expensive to obtain.

3.3.3 Development Costs

Cost is an issue in almost all software development projects, with the main exception being personal hobby projects that are usually limited more by time than money. Just as with timescales, the pressure to keep development costs low can also act to encourage quick (and cheap) decision-making, although a more thorough investigation is sometimes possible if the predicted cost of an incorrect decision outweighs the estimated cost of the research.

3.3.4 Available Skills

The knowledge and experience of team members affects the decisions made during design and development of a software system, both in terms of implementing requirements directly, and in terms of selecting third-party components and tools to use. Lack of experience among the team can also lead to decisions that negatively impact cost or timescales, and in turn increase the pressure to make decisions quickly and cheaply (Jiang et al., 2007a).

3.3.5 Ethics

Depending on the nature of the application or system being developed, ethical issues may have a different impact on the choices being made during the

development process. Ethics in computer science is largely concerned with the uses or impact of *what* is produced rather than *how* it is produced (Alidoosti, Lago, and Razavian, 2022). For example, software to control medical equipment or voting machines (Fleischman, 2010) might be subject to more ethical scrutiny than a calculator or a solitaire game. Individual software developers are not always involved in the decisions about how a software product will be used, particularly if the software they are working on is only a component or library that might be used by other, possibly unknown, applications. In such cases the ethical responsibility rests with the industry as a whole.

3.3.6 The Dilemma of Sustainability

While all the pressures mentioned above are well understood and often discussed in the context of software design and development, the dimension of sustainability is much less frequently acknowledged. Commercial software mostly exists in a money-focused world where income is balanced against outgoings. If the sustainability impact of a system is considered it is usually through the lens of cost - the cost of energy, the cost of adding or replacing hardware, the cost of infrastructure, and so on. Unfortunately, cost is a poor proxy for sustainability impact (Barbier, Markandya, and Pearce, 1990) and is confused even more by techniques such as offsetting, "carbon credits", and "greenwashing" (De Freitas Netto et al., 2020).

The issue of sustainability is a dilemma because, while the corporate forces affecting decisions largely overlook sustainability, that does not imply that the individual people involved in making and implementing decisions are oblivious or insensitive. The same person who might act to reduce waste and greenhouse gas emissions at home, or publicly fight for gender equality and social justice, can find themself in a position at work where sustainability is often sidelined or overshadowed by other factors that are more easily mapped to financial gain or loss.

3.4 Software Development Roles

Irrespective of the situation in which the software is developed, there are many different roles required to make a success of a software product. In personal products and single project products all or most of these roles will be filled by the same person. In larger teams or organisations these roles may be spread out with one or more people filling each role. Software Engineering theory points out

that simply adding more people to a project will not usually make it any faster or more efficient (Brooks, 1995), but simply reducing the number of people will not always be advantageous either. Taking on too many roles can lead to conflicting priorities, time wasted switching between contexts, and a lack of time for the deep thinking required to solve complex problems (Newport, 2016).

As the broad field of computing has developed, the names and expectations of the roles in software development have changed, evolved and blurred, but a few representative examples are listed in Table 3.3:

Although all the roles listed in Table 3.3 have their own responsibilities, they are often combined, and the way they are grouped together can be indicative of the assumptions and structure of the organisation or project.

3.4.1 The One-Person Team

The stereotypical grouping of roles for a small personal or open source project is for one person to do everything in Table 3.3, as well as other organisational functions such as publicity and marketing. This can be a lot of work, and some roles naturally have greater or lesser priority. Typically, in a one-person team, the emphasis is on the architecture, design, and programming roles, with product owner and testing roles usually running second. Sadly, this means that build, operation, documentation, and support often suffer. If the software is released as open source and becomes popular enough, a community of users or co-developers may emerge and provide some peer support and documentation.

Many of the components that are evaluated in later chapters are the product of one-person teams and this can particularly be seen in the lack of support and documentation.

3.4.2 The Business Startup Team

Exemplified by the "Move Fast and Break Things" attitude (Taplin, 2018), many business startups focus strongly on the commercial aspects of a product in order to bring in revenue before things are fully ready. Product owner, designer and architect roles have priority, with documentation and support attempting to make up for the lack of emphasis on programming, testing, configuration management and operation. Often these low-priority roles will be subcontracted, sometimes to just one person.

Role	**Responsibilities**
Customer	The person or organisation who sponsors a product or project, or a client representative such as a *Product Owner*. Arbitrates on choices and makes decisions that affect the overall suitability of the product or project.
User	A person who will use the product. Not often involved in software decisions
Project Manager	Works to ensure that a project completes on time and on budget and meets its aims. Prioritises requirements.
Architect	Makes technical choices that determine the direction of a product. Divides large systems into smaller subsystems and selects infrastructure such as operating systems, databases and cloud platforms.
Designer	May involve high-level decision-making similar to an architect, or other areas not directly related to software, such as graphical design and user experience design.
Developer	Creates, updates, and fixes the software parts of a project. May also do testing, configuration management, and deployment in some contexts.
Tester	Ensures that the system works as intended. May test manually by using a system like a real user, or create and run automated tests.
Configuration Manager	Builds and deploys complex or distributed systems. Ensures that each release contains correct and compatible versions of all components and services.
Operation	Ensures that the server-side parts of the application are available and functioning correctly. Also involves monitoring availability and managing responsiveness as well as backups and resilience, attack prevention and mitigation, and other cybersecurity functions.
Technical Writer	Produces and updates documentation, tutorials, usage examples etc.
Support	Responds to questions from clients and users, either answering them directly or passing them on to other roles.

Table 3.3: Software development roles

3.4.3 The Legacy Software Team

Once a software product evolves beyond the capability of smaller teams, it is common for the team to grow and for people in the team to take on progressively more specialised roles and responsibility for ever more specific niches. This runs the risk of communication issues as discussed in Brooks (1995). A typical response to this is to increase the strictness of the development process, that in turn places a lot more emphasis on testing, build, and configuration management roles, often also accompanied by a growth in the need for documentation and support.

An alternative response to increasing product and team complexity is to make use of a light-weight process that is, however, more formally defined than the ad-hoc approach of a startup team. There are a range of such processes, including SCRUM (Schwaber, 1997) and Extreme Programming (Beck, 2000a), which are commonly included under the umbrella term "Agile". An agile process is characterised by its priorities, for example "Working software over comprehensive documentation", as enumerated in the *Agile Manifesto* (Beck et al., 2001).

3.5 Developer Choices

A typical software development role involves a range of responsibilities such as understanding the domain, the problem to be solved, and the constraints on possible solutions; architecture; design; selection of libraries or other components; testing; documentation; planning, communicating with team members and many others. All these responsibilities involve making decisions that can affect the costs and resource usage of the eventual system. The choice between making a new software component and adapting or re-using one that has already been made, for example, is understood to have an effect on duration and cost of development (Gacek, 2002). However, such choices also potentially impact other factors such as long-term running costs, power usage, cooling and maintenance needs of the system. Research suggests that these factors are often obscured by the importance placed on "up front" development costs and time to market (Petro, 2017), and this is backed up by personal experience.

Evans Data Corporation estimated that in 2017 there were 22 million software developers worldwide, a number that was due to rise to 26 million by 2022 (Evans Data Corporation, 2018). All these software developers have different skills, experiences, attitudes and preferences. Software, and the development of software, can be very complex. There are as many ways to develop software as

there are individual developers, and even individual developers will vary what they do and how they do it depending on factors such as project requirements, team culture and prior experience.

The software development process can be viewed as a sequence of choices that affect the final software product. Some choices are large and set the scene for many others. Some choices are individual and self-contained. This section explores some of the areas in which such choices are made.

3.5.1 Solution Architecture

In most software projects, the biggest and most important decisions are usually in the architecture of the solution. This includes choices such as whether to build software that runs on a client system, a central server, or distributed between multiple participating services, how to deal with the expected number of users and quantity of data, where to store information and how to access it, and so on. Like so many things in software development, there is no clear and universally accepted definition of what constitutes an architectural decision rather than one of the many other kinds of decisions. On the whole, if a decision affects the overall structure of the software system and would have a major impact on the code if it were changed, it is usually considered as an architectural decision.

3.5.2 Programming Languages and Tools

There is a class of decisions, however, which sits somewhere between architect and developer responsibilities and can cause demarcation problems among teams - the choice of programming languages and other development tools. Such decisions do not directly affect the structure and capabilities of the final software system, so it could be argued that they fall outside the remit of architecture (Mills, 1985). On the other hand, they are major decisions that are not easy to change once a large amount of development work has been done, so it could be argued that they need to be included in the key architectural decisions made at the start of a project. (Spinellis, 2006)

In some cases it is possible for an architectural decision to be made that pushes the decisions on tools and programming languages "down" to smaller development teams, each responsible for different parts of the overall system. One technique to enable this is *microservices*, in which the architecture of a system is defined as a collection of collaborating smaller subsystems, each of which communicates only though a predefined interface, allowing development teams freedom to use

whatever development tools, languages, and approaches are most suitable for that subsystem (Chen et al., 2022).

3.5.3 Testing

Software testing is a complex field (Whittaker, 2000) that is largely beyond the scope of this research, but there are some key choices that software developers always need to make during the development process. The first of these key choices is *when* to test the code. A software developer continually faces choices about whether to test their code before, during, or after development. Even having decided *when* to test, there are also choices about *what* to test. At one end of the scale it's probably too much work for too little return to try and test every line of code individually, but at the other end waiting for a whole system to be created before attempting to test any of it makes finding which part of the code caused an error much harder (Singh and Singh, 2012). Some testing approaches such as Test-Driven Development (Beck, 2000b) have been observed to affect the structure and quality of the code itself (George and Williams, 2004).

3.5.4 "Green field" or Code Reuse?

As traditionally taught, the next step after elicitation of requirements and making of architectural decisions is to decide on the algorithms and data structures that will form the final software. Only once these designs are in place, can coding begin.

Although there is eventually an aspect of this kind of programming, it is hardly ever as simple as that (Cusumano and Smith, 1995). Real software rarely starts with a blank page, and must fit in with other parts of the system and conform to the choices that have already been made (Spinellis, 2006). Even so-called "green field" software development (creating something new rather than adding features or fixes to existing systems) almost always involves finding and making use of existing code or libraries that do part or all of the job (Bjarnason, Åberg, and Ali, 2023) (Sametinger, 1997).

Existing code can be used in one of two ways: *white box* (also known as *copy and paste*) - in which the external code is added directly into the project and then modified to suit, and *black box* - in which the external code is added to the project as a library, component or module and used as is. This decision of whether and, if so, how to bring third-party code into a system is made many times during a project, and the context of each choice will determine the outcome. A typical

software system will contain a mixture of both black box and white box reuse along with code created specially for this application.

White box code re-use is the most flexible, as the software developers importing the code have the freedom to include all or only part of the external software, and also to edit or modify it as needed for its new use. White box code re-use also has its problems. Importing partial code or modifying the code that has been imported can introduce problems and security issues that were avoided by the original. White box code reuse also provides the continual temptation to add further copies of the imported code throughout the new project. This not only increases the size of the codebase, but greatly complicates code maintenance if the imported code ever needs to be modified or updated.

Black box code reuse does not offer the opportunity to import partial or modified versions of a component, but does ensure that the imported component retains its full functionality. A black box component is typically used in the form of a *library* or *plugin* which only needs to be imported once for it to be used anywhere in the code. This can help reduce the size of the overall solution. This form of code reuse enables much easier updates if an improved version of the component becomes available. While the code for a black box component may be available for inspection, it cannot usually be modified, so if specific features, performance, or ways of operating are required, there is very little scope for adding them. In such cases, the main option is to look for an alternative component that has the desired characteristics.

3.5.5 Component Selection and Evaluation

The choice of whether and how to bring third party code into a system will be made many times in a typical project, but that should not be taken to imply that the choice or process is easy (Nazir et al., 2014) (Badampudi et al., 2017) (Paschali et al., 2017). Most programming languages, tools, libraries, and software components in modern software development are available free of charge. A large proportion of libraries and components are also *open source* (Androutsellis-Theotokis et al., 2011) with the code available for free online, making them suitable for both white-box and black-box re-use. The spread of open source software has changed software development in a range of ways since its inception (Gonzalez-Barahona, 2021). Many of these changes have been positive, such as a reduction of the cost and complexity of sharing and reusing existing code (see Section 2.8), but the adoption of open source has not been universally beneficial (Vasilescu, Serebrenik, and Van Den Brand, 2013). Open source software varies widely both in quality and in the choices of the developer

(such as programming languages and other required components or libraries) made during its creation (Bissyande et al., 2013).

Commercial software, by its nature, exists to earn money, and commonly some of that income is spent on marketing, advertising, and competition. Free software does not have such an income stream (although there are other ways in which free software developers can earn money from their efforts (Mäkinen, 2022) (Cao, Chintagunta, and Li, 2023)) and this is often reflected in the quality of documentation and support (Sowe, Stamelos, and Angelis, 2008). In some cases open source software provides no supporting documentation other than the code itself. This lack of information has led to a confusing, cluttered, landscape of potential software components for a software developer to choose from (Teixeira et al., 2015). As creating and sharing a new software component is so simple, there is usually an overwhelming number of options to address any common problem (Spinellis, 2019). Where documentation does exist, it might be obsolete, incomplete, contain unverified claims, or even conflict with itself (Midha and Palvia, 2011) (Raja and Tretter, 2012).

Even in cases where the documentation of what a component can do, and how to use it, are of better quality, there is an extra layer of information, which can be much harder to find. This "hidden" information includes such things as the performance of the component, any bugs or security problems, and any other components that it uses (Harrison, 2022) (Spinellis, 2019). Most importantly for this research, it turns out to be extremely hard to find and compare details of the energy consumption of a component (Field, Anderson, and Eder, 2014) (Jagroep et al., 2016a). The lack of detailed information about performance, energy usage and other important aspects of a software component is a sharp contrast to components in other fields of engineering. In mechanical engineering, precise tolerances and load abilities are vital, and in electronic engineering the manufacturer of every component provides an extremely detailed data sheet giving the physical and electronic characteristics of every aspect of the component. A software developer facing a deadline and making decisions about whether or not to re-use an existing software component is forced to choose between using a component without knowing its characteristics or spend a lot of time and effort on trying to understand and evaluate many potential components.

3.5.6 Bigger Decisions

There is another context that surrounds and informs all software development choices and decisions - the point of doing it. This may sometimes be codified in advance or it may evolve as an understanding as the problem and solution

domains are explored, but it has to exist in some form in order for any decisions to make sense.

Depending on the structure of any sponsoring organisation, this kind of decision may be outside the control of individual software developers. Such decisions are still vital to the software development process and, in cases where software developers can exert some influence, can have the biggest impact on sustainability. All of these questions eventually lead to a decision on whether to make software at all. Software that does not exist obviously consumes less power and resources than software that needs to be developed and run (Linders, 2023).

In the broadest sense, every decision has costs, benefits, and risks. The same is true of the decision to create or modify some software. Commercial organisations usually concentrate on the financial aspects of such decisions but there are other aspects including the social and environmental (Souza, 2023) (Barbier, Markandya, and Pearce, 1990). If the overall costs outweigh the potential benefits, or if the risk outweighs the understanding, then arguably the work is not worth doing.

In 1966, Abraham Maslow wrote "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." (Maslow, 1966). This attitude is especially prevalent in organisations that specialise in the development of software. The unique flexibility and malleability of software can easily make it seem like the solution to every problem, even when alternative, non-software, solutions might be more appropriate.

3.5.7 Make or Buy?

Not all software is written "in house". Many software applications have already been written, and some of them may be appropriate for the task in hand. The decision on whether to create something new, or alternatively to use an existing product, is a business decision traditionally known as *Make or Buy*. In physical manufacturing this name makes more sense, as the choice is usually between undertaking a costly design and manufacturing process or negotiating with a supplier to source and purchase something equivalent. In software development, however, the decision is different, but just as difficult. Software is infinitely flexible, so determining whether an existing product is equivalent (or at least *capable* of being equivalent) is a lengthy, complex and expensive process in itself. On the other hand, there is a lot of software that is available for free (in monetary terms, at least), so the decision involves not just deciding which option to choose, but also deciding how much time and money to spend on the

decision-making process. In software development, the "buy" aspect of *make or buy* is not a simple monetary transaction. This can cause problems for organisations with a purchasing policy originally conceived as a way to manage costs, as standard approaches such as selecting the lowest "purchase" price are inapplicable.

It might seem obvious that if the "buy" option is monetarily free, then the choice between "make" or "buy" should always be "buy". After all, programmers, and all the other software development roles mentioned above, are expensive, so a free option should be automatically better. Unfortunately, such a view is naïve. While there may be no monetary cost associated with the acquisition of a third-party software component, there are hidden costs that may ultimately outstrip the cost of making such a component in-house. Typical issues faced when attempting to include an external component in a software product include:

- The component does not include all the required features
- The component has faults that would impair the product
- The component documentation is incomplete or misleading
- The component has security problems
- The component depends on other components that conflict with the product or with other components used by the product
- The component does not exhibit the required performance

Resolving these kinds of issues can often cost more than writing the required code from scratch.

If a third-party component were to exist that was entirely suitable for, and easy to integrate with, the product, then the decision to "buy" would be a good one. However, even this is more complicated than it seems on the surface. Most free components, being built by one-person teams as described above, lack the kind of detailed documentation that is needed to make an informed decision as to whether the component is suitable or not. In economics terms this means that although the *purchase cost* of the component is zero, the total *transaction cost* (OECD, 2003) is considerably more than that. The particular difficulty when it comes to choosing between make or "buy", or choosing between options to "buy", is that the actual transaction cost of each option is unknown, and even the process of determining the transaction costs has a transaction cost of its own, and that is also unknown.

In many cases the organisation simply washes its hands of such choices, leaving the decisions to technical staff who are expected to be able to make an appropriate choice. Unfortunately, the process of evaluating potential components takes time away from other work, and the resources to make an optimal choice are rarely available. This evaluation and selection process will be discussed in more detail in Chapter 4.

3.6 What is The Industry Doing About Environmental Issues

An obvious method to reduce energy use is to turn off unused computer systems. This technique has even gained a name: "LightSwitchOps" (RedHat, 2022). Although simple in concept, this has turned out to be a surprisingly difficult problem, with the main issues being identifying which computer systems are not being used (Lechelt, Gorkovenko, and Speed, 2024), and gaining access to them to switch them off (Wiggers, 2023). Distributed systems comprised of multiple services on multiple physical devices usually run continuously on the basis that a user or another system *might* require access to the service at any time. Actual use of such a system might be rare, so determining whether a particular system should be running requires more than just observing traffic over a short period. A system may actually be obsolete and will never be used again, or it may be largely idle until specific circumstances such as the failure of another system or extra traffic from a busy "Black Friday" sale. Some systems are known to be obsolete but remain active because shutting them down requires access credentials or knowledge that are no longer available. Such unused systems and services have gained the description "zombies" because they remain active and consuming resources even though they should be "dead".

On the whole, computing technology has got faster and cheaper every year since its invention. One way organisations aim to reduce energy usage is to replace old systems with newer, more efficient ones. Energy reductions can be achieved in two main ways: by replacing a single system with one that is in some sense "better" (usually known as *upgrading*), or by replacing several systems by one which is powerful enough to do all their tasks (usually known as *consolidating*).

Upgrading has been a popular choice for many years. It is usually relatively simple to implement because it requires no changes to the architecture of the overall system. The new machine can be configured to perform the same roles as the old one, responding to the same addresses and instructions in a way that is indistinguishable to users and other participants. In an ideal situation, the old

machine is then switched off, decommissioned and responsibly reused or recycled. This is not always the case, of course, and missing this step of the process is one of the ways in which machines become "zombies". Upgrading is attractive because it can also provide an increase in capabilities, perhaps storing more data or handling a larger number of requests. However, upgrading does not always reduce energy usage. If the primary purpose of the upgrade is to increase capabilities it is common for the replacement to need at least as much power and cooling as the machine it replaces. This can still be a saving of sorts, if the increase in capabilities were already needed and the alternative was multiple less-efficient machines with a greater total energy requirement than the single new one. This, however, could also be viewed as a form of pre-emptive consolidation.

As well as the operational requirements, energy is also needed to manufacture, transport, and dispose of ICT equipment. Reducing the overall number of computing devices is one way to address rising energy usage. Once zombie machines have been identified and removed, the next step is to find machines that are not being used to their full capacity and consolidate them. Consolidation is a more complex process than simply upgrading old machines. It may require changes to the architecture of the system and the way that elements of distributed systems find and communicate with each other. There may be conflicts with the way the old and new machines are addressed, and so on.

Various technologies have been created to help avoid such problems. The most well established of these is *virtualisation*. This is a technique for running several *guest* virtual machines, complete with operating systems, applications, and data on a single (*host*) computer. None of these guest systems is aware of the others. From a software point of view they are completely separate installations. To make this work requires a *hypervisor*. A hypervisor is a specific form of operating system whose sole job is to manage guest systems and provide them access to the hardware resources without any conflicts. Virtualisation has the advantage that, once the hypervisor is in place, the process of adding a new guest system is usually no more difficult than installing the old system on an upgraded machine of its own. A hypervisor can also provide important management tools such as the ability to take "snapshots" of running systems before making changes, or to migrate a whole virtual machine to a new hardware host without major interruption in service. The main disadvantage of virtualisation is that each guest machine needs a complete software installation including the operating system. This can use a lot of memory and storage resources. Each virtual machine can require many gigabytes of storage and memory, even when the application part of the software is relatively small.

An alternative to virtualisation is *containerisation*. This has many aspects in

common with virtualisation, and provides many of the same benefits in terms of sharing of resources and management tools. In a container system, however, rather than requiring a full installation for each guest machine, several applications can share common facilities such as the operating system and read-only resources. Each containerised application is unaware of the others, and appears to have a whole machine to itself. In most cases the size of a containerised application or system is much smaller than a corresponding virtualised machine image, ideally little larger than the specific resources and code for the application itself. Creating and managing containerised applications is more complex than the equivalent tasks for virtual machines. Each container image must be created using specialist container tools. Although container images are theoretically portable, they can also require specific features of the host system depending on how they were created. Creating, deploying, managing, and updating containerised applications requires very different skills to installing a full system on virtualised or "bare metal" hardware.

When upgrading, consolidating, or simply moving a software system to a different machine, the destination does not usually have to be in the same physical location as the original. There are some cases where issues such as physical security or communication latency mean that devices must be situated together, but most distributed system designs are flexible enough that the components can be anywhere with a network connection. This opens up the possibility of *cloud computing*. Cloud computing is not a precise or rigidly-defined term but is generally taken to include the leasing of computing and storage resources owned and operated by another organisation. Such resources are usually located in large datacenters where economies of scale can reduce the overheads of running and maintaining a large number of computer systems. Although it is possible to lease whole machines (known as *hosting* or just *leasing*), or in some cases place your own equipment in a shared datacenter (known as *co-location*), cloud computing is mostly achieved using a combination of virtualisation and containerisation. Full guest images or containerised applications are uploaded to the cloud provider ready to be deployed to appropriate host systems. Most cloud organisations also provide tools to configure virtual networks between these cloud-hosted systems and to make specific ports and services available to the wider internet.

Simply moving a computer system to a different location in a more efficiently run datacenter can have some sustainability benefits, but this must be balanced against the environmental cost of more network traffic when systems are physically separated. The real benefit of cloud computing is in the ability to add or remove additional copies of machines according to demand. In traditional system designs,

adding new machines is a slow and costly process involving ordering, installing and setting up new hardware, and often changing the software application as well. Because this process is slow and expensive, such systems are usually designed and built with enough capability to handle the largest expected load. In a cloud system, deploying a new virtual machine or container image is quick and simple enough to be automated and performed only when needed. This allows cloud applications to be set up initially to use just enough resources for normal use. If a service suddenly becomes popular, more machine images can be started to cope with the load, and when that demand drops, some of those extra machines can be stopped again, and their resources returned to the pool for use by other systems. This is commonly known as *dynamic scaling*. A system does need to be designed with dynamic scaling in mind, though, to ensure that there are no conflicts when new parts of the application are "spun up" or communication failures when parts are "spun down", and that the operation is generally seamless.

With the increase in awareness of the impact of energy usage, manufacturers have also concentrated on reducing the power consumption of each new generation of technology. A reduction in power consumption is accompanied by the production of less waste heat, which in turn helps to reduce the need for cooling, and thus reduces the power needed for that, too. One of the advantages of designing applications for containerisation is that it becomes relatively easy to rebuild them to take advantage of lower-power hardware as it becomes available. Most cloud providers now offer "greener" hosting options in addition to traditional computer architectures.

Sadly, the continued introduction of new and updated technology has its costs, both in financial terms to buy replacement equipment, but also in the energy and resources used in the production of the new equipment and an increase in "e-waste" when the old equipment is discarded. This problem with obsolescence exists at all levels. Component parts such as disc drives may need to be replaced to provide more storage, even though they still function. Machine clusters and even whole datacenters can become similarly obsolescent. There is a continual trade-off between the impact of continued operation with obsolescent equipment, and the impact of replacing that old equipment with new hardware.

Not all the energy used by the internet is used in server or client machines. A large amount is used in the network itself. The routers, firewalls, transmitters, receivers and signal boosters that make the internet work take energy to operate, and much of this is dependent on the amount of data being transferred. Techniques that reduce the amount of network traffic can also help to reduce energy consumption. In cases where the same data is sent across a network multiple times, such as the content of popular websites or streaming media, local caching can reduce the

number of times the data moves across long distances. This also has the advantage of reducing contention for network resources and can potentially provide faster or more reliable access to the content. The increasing demand for machine learning and artificial intelligence can also increase data traffic when large amounts of input data are sent to central servers for processing. The introduction of specialist neural-processing hardware has meant that some of this processing can now be performed at the "edge" of the network, so that only the results of the analysis need to be transferred.

While these improvements may seem promising, it is against a backdrop of continually increasing demand. Users expect more information, entertainment and commerce to be available online, the rise of remote working places an extra burden on the network, and even many of the attempts to address other aspects of sustainability end up requiring server processing and network traffic. There is no sign of this growth in internet use, or its associated energy usage, slowing down any time soon (Morley, Widdicks, and Hazas, 2018) (Odlyzko, 2016).

In summary, the computer industry is aware of the environmental problems and the energy usage of computer systems, and has made some steps to address these issues. These steps mostly include virtualisation and containerisation, cloud computing, dynamic scaling and upgrading to "greener" hardware where possible. Despite this, computer and software systems keep growing and the overall energy usage keeps increasing. All of the attempts at improving the sustainability of computer systems discussed in this section have one thing in common. They treat it as a *hardware* problem. Despite all the improvements in capability and efficiency, computer hardware is still essentially fixed. What makes the difference between one system and another is the software. Software was designed for the express purpose of being able to change the behaviour of electronic systems without changing the hardware, but, as shown in Section 2.14, relatively little research has been done into improving the development of software to make it more sustainable.

3.7 Context Summary

Most people who work in software development for any length of time come to recognise one central distinction. Even though the hardware, operating systems, programming languages and data formats are objectively logical to the point of pedantry, the skills, practises and processes of software development itself are loose, vague, subjective, incomplete and even contradictory. Many attempts have been made to formalise software development (Glass and Wesley, 2002), but the

sheer number and variety of software products, projects and software development teams defy easy solutions.

Commercial software development is a high-pressure, high-value, business. Time to market is critical and developer time is expensive, so it makes sense to reuse existing code wherever possible. In such situations, developers tend to choose any solution that "gets the job done", often with little consideration of the broader effects of such decisions. Modern software is commonly replicated to many real or virtual machines and will often be executed millions of times per day. Even small differences in resource usage can be magnified hugely (Vercauteren et al., 2007) (Andreolini and Casolari, 2006), requiring more servers in larger data centres, needing more cooling, using more power, costing everyone more, producing more carbon dioxide, and accelerating global warming. The continuing demand for computationally-expensive crypto-currencies, for example, would result in "an unacceptable amount of energy consumed" (Giungato et al., 2017).

This research aims to explore ways to help software developers make smarter choices about the broader, long-term impact of their work, concentrating on comparing and reducing energy usage of large-scale applications through the choice and substitution of software components and libraries.

Chapter 4

Comparing the Performance of Software Components

This chapter investigates the challenges of comparing the performance of ostensibly similar software components. A feasibility study was created to compare the execution speed of a cohort of text template engine components and the results were analysed. Several issues were found with the feasibility study and these were then addressed in a series of further experiments.

4.1 Introduction

In principle, whenever the decision is made to include external libraries, components, or applications in a software product, component sources and candidates should be investigated, evaluated, and compared in order to make a final decision. In practice, things are much more complex than this (Badampudi, Wohlin, and Petersen, 2016). The literature contains many attempts to rationalise and streamline this process, using approaches such as economic models (Milkman, Chugh, and Bazerman, 2009), "decision maps" (Lago, 2019) and even machine learning (Maxville, Armarego, and Lam, 2004).

A typical component selection process consists of several broad phases: *discovery* of the potential candidates, *filtering* by eliminating clearly inappropriate candidates, and *evaluation* of the remainder to determine their suitability. These phases may be sequential, or they may overlap, but in principle each candidate component passes through the process.

There is no single "catalogue" of software to choose from, so even discovering a

list of potential sources or candidates can be time-consuming and will probably be incomplete. Open source software components are available from a wide range of sources including: general code repositories such as *GitHub*[1], *Bitbucket*[2], *Sourceforge*[3], and *Google Code*[4]; organisation-specific repositories such as *The Apache Projects Directory*[5]; and language-specific package repositories such as *Maven Central*[6], *CPAN*[7], and *PyPi*[8]. These repositories are not exclusive - some software components may appear in multiple repositories. Software components are also not entirely distinct. Popular open source software components are often "forked", a process roughly equivalent to copying and modifying, so similar components may appear in multiple repositories with different names and potentially different features, bugs, and non-functional attributes.

Most repositories provide basic programming language and operating system compatibility information, so filtering candidates at that level is relatively simple. Beyond that, filtering relies on the documentation (including the source code, if available) provided for the component. However, documentation for software varies widely in content and structure and is often biased, incomplete, or inaccurate (Bertoa, Troya, and Vallecillo, 2003). Further filtering requires a mixture of investigating the documentation provided with the component and exploring other sources of information such as external reviews, articles, and comparison websites.

Evaluation of software components can be lengthy and expensive, particularly when the requirements for the component are complex or there are large numbers of candidates to choose from (Badampudi, Wohlin, and Petersen, 2016). In many cases components will not be directly interchangeable, but require software to be written or adapted or data to be reformatted in order to use them. This adds time and cost to any kind of in-situ comparisons.

Where time and cost are important, as they almost always are, it is common to resort to heuristics based on factors such as personal experience, project or company history, or what has been read about recently. In many cases the evaluation process will stop as soon as a candidate is encountered that appears functionally suitable. What is rarely apparent is how much candidates may differ in "non-functional" aspects such as performance, maintainability, and

[1] `https://github.com/`
[2] `https://bitbucket.org/`
[3] `https://sourceforge.net/`
[4] `https://code.google.com/`
[5] `https://projects.apache.org/`
[6] `https://repo.maven.apache.org/maven2/`
[7] `https://www.cpan.org/`
[8] `https://pypi.org/`

energy efficiency. This chapter will explore this issue through the lens of one particular type of software component: the *template engine*.

4.1.1 Template Engine Software

There is little consensus in this field. A template engine is a conceptually simple piece of software, sometimes used as a teaching aid (Koskela, 2007), but which has the potential for many different implementations with different features and complexity. A search of the popular open source software website *GitHub* (GitHub, 2023) reveals more than 13,000 software projects with names or descriptions that include the term "template engine" (Figure 4.1.1). A cursory examination of the results indicates that the great majority of these results are separate template engine implementations in some state of development. GitHub is not the only source of software components and libraries. Selecting a template solution for a project can be a major task.

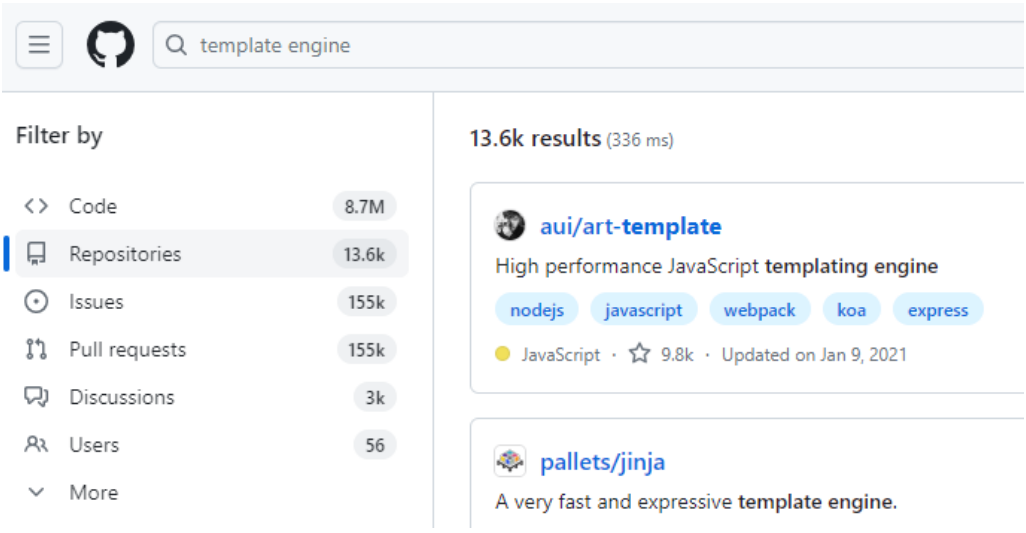


Figure 4.1.1: GitHub search for the term "Template Engine" in October 2023

Project names and descriptions in a software repository such as GitHub are set by project creators and can use a variety of terminology. When searching for academic literature (see Section 2.2), the most distinctive search term was "template engine", and this appears to hold for the GitHub repository. For example a search for the alternative tern "template processor" returned considerably fewer results, and of those most seemed unrelated to the process of generating documents or web pages using templates [Figure 4.1.2].

Comprehensive comparative information about the many differing implementations can be hard to come by. Software documentation for such tools, where present at all, tends to focus on the use of a single implementation

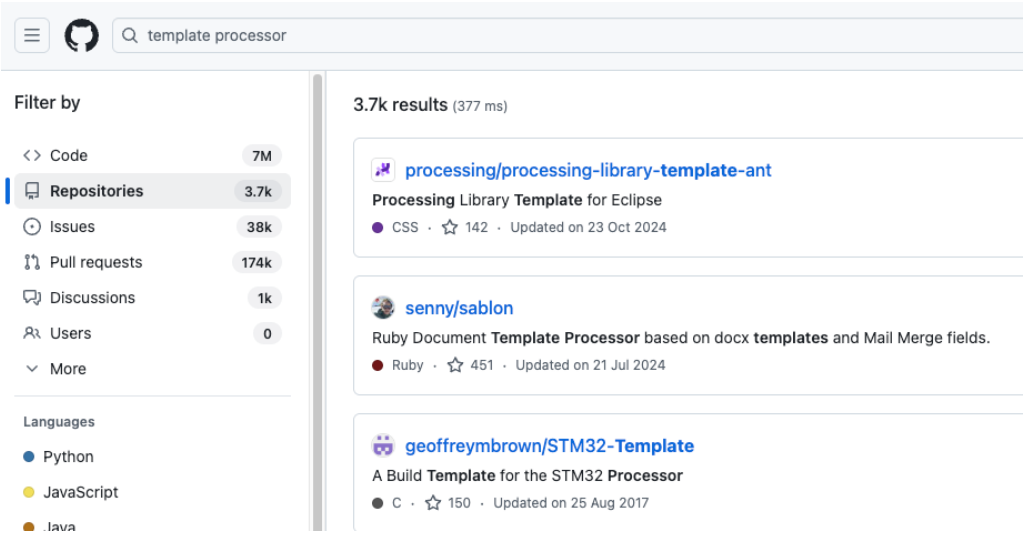


Figure 4.1.2: GitHub search for the alternative term "template processor"

and often reads more like marketing material than a datasheet: emphasising strengths with unverified claims, ignoring weaknesses, and avoiding direct comparison with alternatives.

Template technology is therefore a representative category of software implementation that exhibits the scale, consequences, and lack of easy choices discussed in Section 3.7. The rest of this chapter will dig deeper into template technology, its characteristics, and the variation between a range of implementations.

4.2 Template Engine Implementation Differences

Although all template engines perform essentially the same function, there are a wide variety of ways of achieving the same end result. All the template engines selected in Section 4.3.2 and Section 4.6.3 (described as the *cohort*, below) are implemented in the same programming language but there are still several key areas in which they differ.

4.2.1 Template File Formats

As discussed in Section 2.1, the essential concept of a template is a document containing blocks of boilerplate text interspersed with placeholders and other template instructions which will be transformed during the template expansion process. However, some template engines require other information in addition

to the boilerplate and placeholders, and this is usually provided in the template files. For example, the *JTE* template engine requires the names and types of all applicable context values to be specified at the start of the template as a series of `@param` declarations.

Some template languages support optional directives or configuration blocks within templates that affect the operation of the template engine. An example of this might be a block of *YAML*[9] or *JSON*[10] data providing named values for use in placeholders in addition to the values provided in the context. Such configuration blocks are typically found at the start of template documents. Some template languages also allow specific directives, which affect the processing of a subset of the template, to be included inline in the template text.

In addition to the choice of template language, which determines the format and content of the files, the various template engines apply a range of constraints to the names and locations of the files containing the template data. Some template engines in this cohort are relatively flexible and can be configured to locate and expand templates from arbitrary locations. Others require that templates are stored in a specific file structure or have filenames with a specific suffix in order to be discovered and used by the template engine. Some template engines impose extra constraints on template filenames, such as only allowing certain characters or not recognising file names that start with numbers.

Different template engines also make different assumptions about the encoding of characters used in boilerplate text and context values. Some assume, for example, that template files are encoded with 8-bit ASCII characters while others can deal with unicode encodings such as UTF-8 or UTF-16.

4.2.2 Placeholders and Delimiters

The template engines under consideration have a broadly similar understanding of simple substitutions. A placeholder contains the name of a value to be retrieved from some context provided to the template expansion process. When such a simple substitution placeholder is encountered during template expansion, the value associated with the specified name is retrieved from the context, converted (or *rendered*) to a textual form if needed, and included in the output document in place of the placeholder. Each placeholder is introduced and terminated by distinctive character sequences, ideally sequences that are rare in the boilerplate text. Without a way to identify a placeholder among other text, a template engine

[9] `https://yaml.org/`
[10] `https://json.org/`

will not be able to function.

Arguably the most obvious difference between template languages is the syntax for these identifying sequences. For example, several of the template engines in the comparison use `{{` to start a placeholder and `}}` to end it. Everything between these pairs of characters is *inside* the placeholder, and therefore not part of the text that will be passed to the output document. Instead, the contents of the placeholder serve to instruct the template engine what to do at that point in the text. `{{` and `}}` are not the only way to identify a placeholder, however. Other template engines use a range of differing approaches. A range of character sequences delimiting placeholders are shown in Table 4.1.

Template Engine	**Start delimiter**	**End delimiter**
Freemarker	`${`	`}`
Handlebars	`{{`	`}}`
Hapax	`{{`	`}}`
Jangod	`{{`	`}}`
Jmte	`${`	`}`
Jte	`${`	`}`
Mustachej	`{{`	`}}`
Pebble	`{{`	`}}`
Solomon	`${`	`}`
Stringtemplate	`$`	`$`
Stringtree	`${`	`}`
Thymeleaf	`[(${`	`})]`
Trimou	`{{`	`}}`
Velocity	`${`	`}`

Table 4.1: Placeholder delimiting characters by template engine

Note that the delimiters described in Table 4.1 are for simple substitution placeholders. Most of the template engines in this cohort use the same delimiters for all templating purposes, but some use different syntax. For example, although *Freemarker* uses `${` and `}` for simple substitution placeholders, it uses `<#` and `>` for more complex control structures such as decisions, loops or the inclusion of one template within another. *Thymeleaf* is unusual in that it uses a two-level placeholder delimiter syntax, described more fully in Section 4.2.3.

Although *Stringtemplate* uses a pair of `$` characters as delimiters by default, this template engine can be configured in software to use any pair of single characters. This is useful as, of all the placeholder delimiters in this cohort, a single dollar sign seems the most likely to appear in boilerplate text, particularly in situations where it is used as a currency symbol.

Outside the set of template engines measured in this research, an even wider range of placeholder delimiters are used, including combinations of `<`, `>`, `#`, `$`, `%`, and many other non alphanumeric characters. In some template languages such as *Haml*[11] or *Webmacro* (Hunter and Crawford, 2001), even "whitespace" such as space, newline or tab characters can serve to delimit placeholders.

Beyond such simple substitution cases, template languages vary in more substantial ways

4.2.3 Internal and External Control Structures

A template language can be thought of as a specialised kind of programming language and, as such, many template languages include their own forms of classic programming control structures such as conditions, loops, and subroutines, as well as expressions such as string, mathematical, and comparative operations, array and object member access, and function or method calls. Some template languages have support for other programming language features such as constants, variables, and parameters. In addition to the built-in template language features, some template languages such as *Casper* or *JSP*[12] provide a way to "hand off" complex processing to another programming language. In the case of *Casper*, that language is *JavaScript* (formally known as *ECMAScript*[13]), which is interpreted when each template is expanded. In the case of *JSP*, both the template text and the program elements are transformed into Java source code that is compiled before being used to generate the output documents.

One way to categorise the different approaches to this area is to divide template languages by whether they use *internal* or *external* control structures.

Internal Control Structures In a template language that uses internal control structures, the keywords or symbols that define the control structure behaviour are found *inside* the delimiters for regular placeholders. While such template

[11] `https://haml.info/`
[12] `https://jakarta.ee/specifications/pages/3.0/jakarta-server-pages-spec-3.0`
[13] `https://tc39.es/ecma262/`

languages often default to treating the content of a placeholder as the name of a context item, they also attempt to parse the placeholder contents according to their own language specification in order to distinguish more complex instructions from simple context names. This group further subdivides into those that describe a whole control structure such as a loop or conditional within a single placeholder, and those that require multiple placeholders to describe such structures. In this cohort of template engines, *Stringtree* and *Solomon* are in the single-placeholder subgroup, while *Handlebars*, *JMTE*, *Mustachej*, *Stringtemplate*, and *Trimou* in the multiple-placeholder subgroup.

External Control Structures In a template language with external control structures, these are indicated in the template using a different syntax from regular substitution placeholders. *Freemarker*, as mentioned above, uses both `${` with `}` and `<#` or `</#` with `>`. All the template engines in this cohort with external control structures require multiple directives to describe the structures. There is no equivalent of the single-placeholder control structures used by *Stringtree* and *Solomon*. In this cohort, *Freemarker*, *Jangod*, *JTE*, *Pebble*, and *velocity* have external control structures.

Alternative Approaches The outlier in this classification is *Thymeleaf*, which can potentially be viewed as having either internal *or* external control structures, depending on which character sequence is considered as defining a placeholder. In *Thymeleaf*, all substitution placeholders and control structures are contained within `[(` and `)]`. If those are considered as the placeholder delimiters, then *Thymeleaf* sits in the internal group. However, this does not have the same characteristics as the substitution placeholders in other template languages. *Thymeleaf* will not accept the name of a context value between these symbols, instead, such a name needs to be enclosed within a further set of `${` and `}` delimiters. It is possible to use *Thymeleaf* as if the substitution placeholder delimiters are actually `[(${` and `})]`, in which case control directives such as a loops or conditionals would count as external.

4.2.4 Template Language Grammar

As well as the differences described above, template languages differ in grammar. This difference is apparent in template languages with both internal and external control structures. Some template languages attempt a grammar similar to a programming language, some prefer to align with markup languages such as HTML, while others take a more mathematical approach. Most template

languages are a mixture of these approaches. Consider a simple conditional scenario in which a context variable named "stock" has a value of true or false, representing whether an item is in stock. In the output document this status is to be represented by the word "Available" or the word "Unavailable".

Freemarker uses a control structure grammar, largely inspired by HTML, that emphasises that this is a conditional operation by using the keywords "if" and "else", this template fragment might be represented as shown in Listing 4.1.

```
<#if stock>Available<#else>Unavailable</#if>
```

Listing 4.1: Freemarker conditional syntax

In *Handlebars* there is no equivalent to the "if" and "else" keywords, but instead it repeats the name of the context variable at the start and end of each block and uses symbols `#` and `^` to imply *if-true* and *if-false*, so the same template fragment might look like the code in Listing 4.2

```
{{#stock}}Available{{/stock}}{{^stock}}Unavailable{{/stock}}
```

Listing 4.2: Handlebars conditional syntax

Jangod uses an "if"/"else" grammar that is in some aspects similar to *Freemarker*, but with a syntax further from HTML, which introduces an "endif" keyword. See Listing 4.3.

```
{% if stock %}Available{% else %}Unavailable{% endif %}
```

Listing 4.3: Jangod conditional syntax

For comparison, *Solomon*, a template language with conditional control structures based partly on the *ternary operator* (Oracle, 2021) found in the Java programming language, might represent the same scenario using the code in Listing 4.4.

```
${stock?'Available':'Unavailable'}
```

Listing 4.4: Solomon conditional syntax

4.2.5 Single or Multiple Files

In a similar manner to the distinction between internal and external control structures, both template language designers and template authors often face a choice between representing a template in a single file, or split into multiple files. Most template languages support some way for a template author to "include" one template in another, perhaps to re-use existing templates, or to separate

repeated sections to enhance readability and maintainability. In addition to this, some template languages require such splitting for the implementation of control structures.

Consider, for example, a template used to generate a page of a catalogue from a list of product details as shown in Figure 4.2.1. Some aspects of this template, such as a page header and footer, might be used only once. However, the section of the template that generates the description for each product will be used as many times as there are products in the list. This requires either foreknowledge of the contents of the list when creating the template or, in the general case, the use of some form of "loop" or "iteration" control structure that evaluates a sub-template with the details of each product in turn. A symbol sequence for such a loop structure might look like Figure 4.2.2.

In this cohort of template engines, two (*Stringtree* and *Solomon*) have a template language that requires that such sub-templates are represented as separate files. All the other template languages support some form of in-line sub-templates. As can be seen in Section 4.2.3, the two template engines which require separate files are also the template languages that use a single placeholder for such control structures. The use of internal control structures with single placeholders does not inherently imply a multiple file representation, as can be seen in template languages which allow the inclusion of general purpose programming code such as *Casper* and *JSP*. However, doing so can greatly simplify the parsing of placeholders due to the constrained syntax rules for filenames compared with arbitrary template or program code. A requirement that such sub-templates be represented in separate files can also help mitigate the problems of nested control structures (see Section 4.2.6). Disadvantages of requiring that sub-templates are stored in separate files include reduced top-to-bottom readability of templates and the creation of a profusion of files each needing distinct, while preferably also descriptive, names.

4.2.6 Nested Control Structures

Returning to the catalogue example used in Section 4.2.5, there is potentially a further problem. Within the sub-template used to describe each product, there may be the need for further control structures. For example, imagine that each product in the catalogue is available in several colours that need to be listed alongside the other product information as shown in Figure 4.2.3. A natural solution to this would be to use a similar loop or iteration control structure to the one that was used to generate each product entry, but with its own sub-template within it. A symbol sequence for such an approach is shown in Figure 4.2.4.

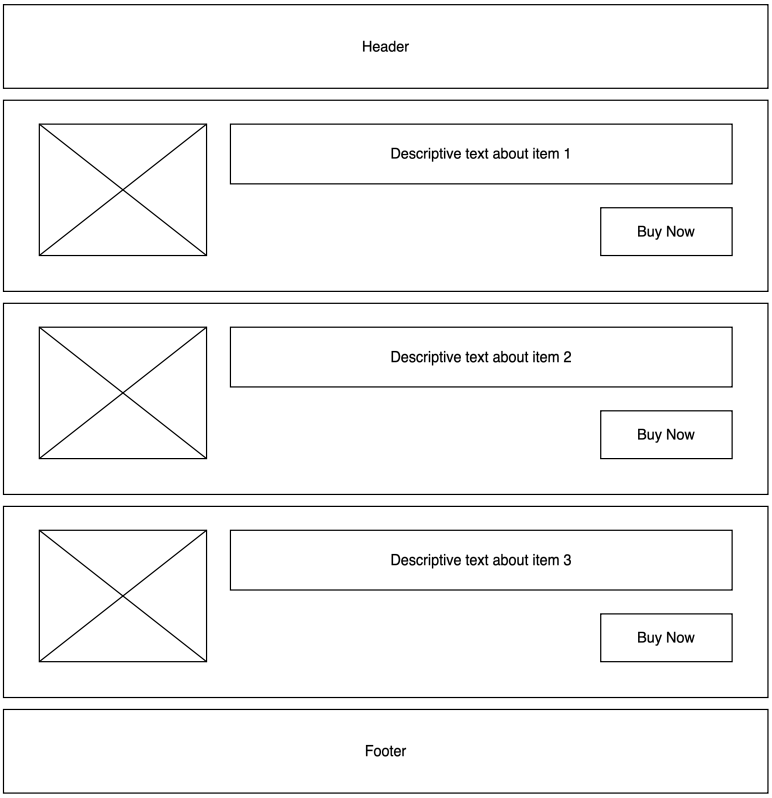


Figure 4.2.1: Wireframe for an example of a simplified catalogue page showing repeated sub-templates for a list of items

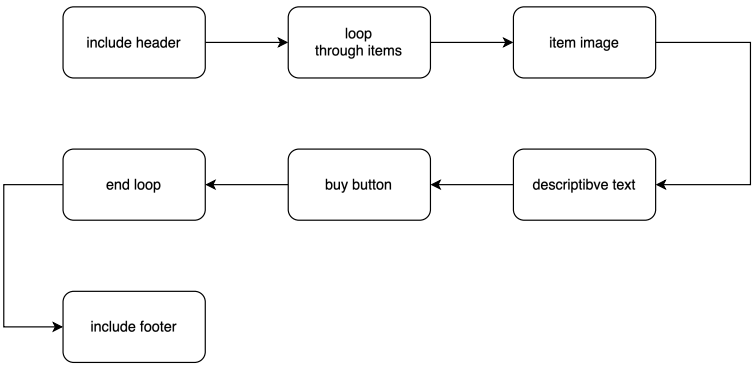


Figure 4.2.2: Symbol sequence representing a loop template for the catalogue page shown in Figure 4.2.1

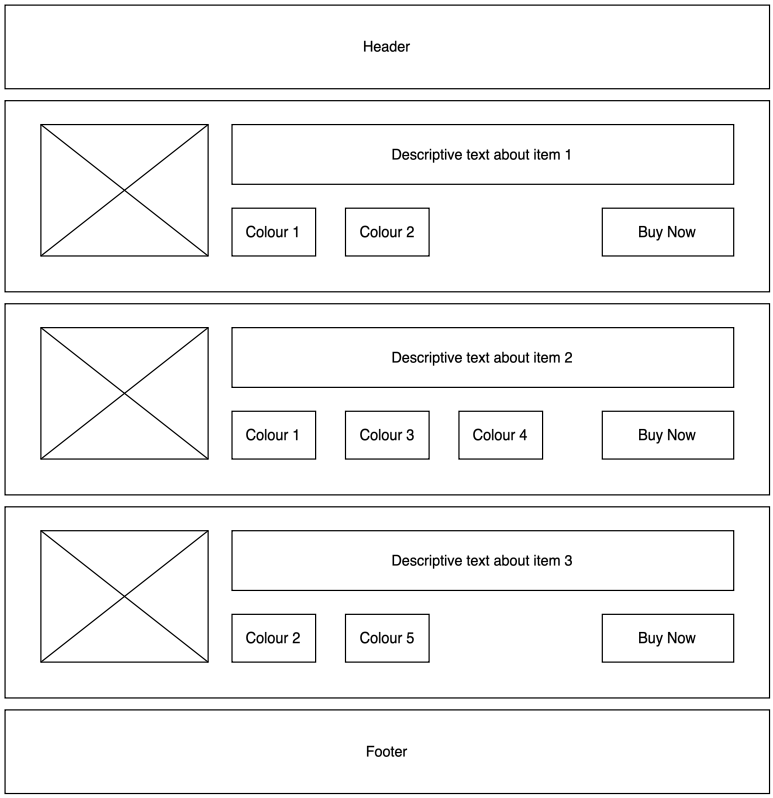


Figure 4.2.3: Wireframe for a catalogue page with multiple colours for each item

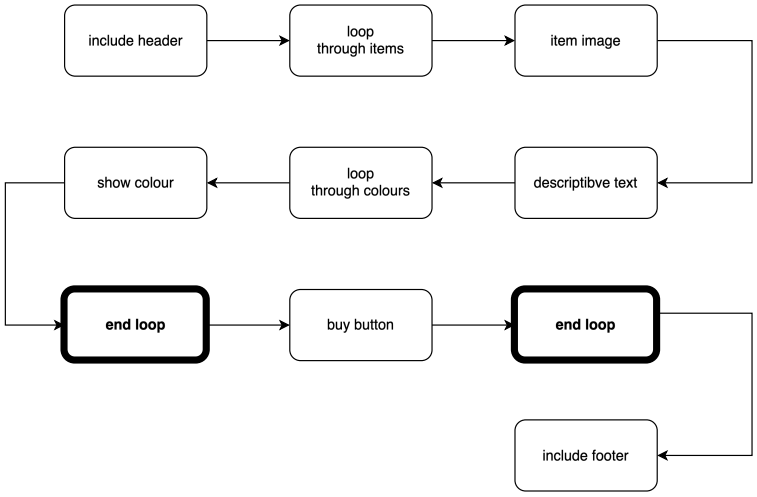


Figure 4.2.4: Symbol sequence representing a nested loop template for the page in Figure 4.2.3 highlighting symbols with context-dependent meanings

However, this nesting of sub-templates raises a problem for template language and template engine designers. The simplest approach to processing a sequence of symbols is a stateless one in which each symbol is dealt with as it is encountered. For most symbols this is unambiguous. These symbols either generate output or indicate a control structure. However, when a control structure spans multiple symbols and *contains* an in-line sub-template, which may in turn contain further control structures and sub-templates, the symbols representing the *ends* of such control structures can be ambiguous as to which structure is ended by which symbol. In Figure 4.2.4, the ambiguous symbols are shown in bold. To correctly resolve such ambiguous symbols requires extra information beyond the symbol itself. This ambiguity can allow for erroneous symbol sequences that appear valid until much later.

This problem is not unique to template languages - general programming language compilers and interpreters include various techniques to address this issue (Aho et al., 2007). One solution is to maintain a stack of current scopes. Another solution is to require each end symbol to contain some form of identifier for its associated start symbol.

Although most template languages in this study allow such nesting of sub-templates, a few take an alternative approach. The *Stringtree* and *Solomon* template languages, as well as the *GILT* template language discussed in Appendix E, represent each control structure by a single symbol with no need for an in-line sub-template and its associated *end* symbol. *Stringtree* and *Solomon* require all sub-templates to exist in separate files. In these languages, every control structure is a single symbol which includes a reference to a sub-template file. In such an approach, the start and end of each sub-template is implicitly indicated by the start and end of the sub-template file itself. The start and end of a file are external constructs not represented as characters, so can never appear within the file. The ability to process template language symbols in a stateless manner is one contributor to the impressive speed of *Solomon*.

The *GILT* template language also represents each control structure by a single symbol with a reference to a named sub-template, but allows the sub-template to be either a separate file, or included in the main template as a separate, named, non-nested, block outside the scope of the main template. More details of the implementation of the *GILT* template language can be found in Appendix E.

4.2.7 Placeholder Value Expressions

All the template languages in this cohort, and arguably template languages in general, support simple value placeholders that replace the placeholder with a single named value from a provided context. Some template engines also support a variety of forms of placeholder value expressions in a manner analogous to expressions found in programming languages. Both the kinds of expressions supported and the syntax used to specify them varies widely. Some template languages, such as *Mustachej*, support no expressions at all, requiring all values to be pre-calculated and stored as named values in the context. Others, such as *Casper*, hand over all placeholder value expressions to a general-purpose programming language and support all the possibilities that offers. In the general case, each template engine supports whichever kinds of expressions the template engine developers considered worth the effort to implement.

All the template engines in this cohort are based on the Java programming language and expect Java values to be provided in the template context. Java is an Object-Oriented programming language (Adenowo and Adenowo, 2020) and all values that will be available in a template context are therefore "objects"[14]. Java objects provide access to both data (in the form of "fields") and program code (in the form of "methods"). Each object also has a mandatory *toString()* method that renders the object in a textual form. This *toString()* method is typically called by a template engine to obtain the representation of the object value to include in the final document in place of the placeholder. Some Java objects, such as Strings and numeric values have a hard-coded implementation of the *toString()* method, but in general, Java programmers are free to implement *toString()* in any manner that seems appropriate to the contents of the object.

Note that with every kind of placeholder expression, there is the possibility of a name conflict if a value is placed into the template context with the same "name" as the complete (or partial) expression. The precedence and behaviour in this situation is not usually defined in template engine documentation and varies between implementations, in some cases even changing between versions of the same template engine. General guidance seems to be to avoid this situation wherever possible.

Typical placeholder expressions that might be supported include:

[14] Java does support some forms of non-object "primitive" values but these cannot be placed in general-purpose collections, so will not be available in a template context.

Field Access Access to a data field of a context object needs both the name of the object and the name of the data field. In the Java programming language, and many others, fields are almost always accessed by writing these two names separated by a '.' character, and this convention is carried forward into template languages. In this cohort, template languages that support field access all do it using this syntax. For example, in *Jangod*, which does support field access, rendering the "name" field of a "customer" object would look like Listing 4.5.

```
Welcome {{customer.name}}
```

Listing 4.5: True object field access in *Jangod*

```
Welcome {{customer.name}}
```

Listing 4.6: Syntactically identical but semantically different "fake" field access in *Mustachej*

The two templates shown in Listing 4.5 and Listing 4.6 have the same syntactic representation, but very different semantics. The use use of context value names that conflict with template expressions is generally discouraged, but is still a relatively common practice as a way of "faking" some kinds of expression in template languages that do not support them (Mittapalli and Arthur, 2021).

In *Mustachej*, which does not support field access, rendering a field of an object requires an extra step of pre-fetching the object field and placing it into a new context value and referring to that new value in the template. If this new context value were to be named "customer.name", the resulting "fake" field access template might look like Listing 4.6. This appears identical to the "real" field access template given for *Jangod*, above. However, this technique can be confusing for template authors, who might reasonably assume that other fields may be accessed the same way.

Method or Function Call Some context objects may be more complex than just a collection of fields, and also contain associated code. Code associated with an object is commonly known as a *method*, and some template engines support the calling of methods during the expansion of a template. When supported, this can be a way to include behaviour and expressions either beyond the capabilities of the template language or that would require complex template control structures.

Array or List Element Access In addition to named fields and methods (and, in the case of the template engines in this cohort, JavaBeans[15]), some context

[15] https://docs.oracle.com/javase/8/docs/technotes/guides/beans/index.html

objects represent more complex data structures. In particular, it is common for context objects to include collections of other objects. The key distinction is that the objects in such collections are not individually addressable by name, but only as members of a collection. Java supports a variety of different collection-style types including Arrays, Sets, Lists, and Maps. Each of these types provides a different selection of methods to access the individual elements. Arrays and Lists, for example, are "ordered" and provide methods to access elements in order or an element at a specific position. Sets and Maps are "unordered", so do not provide those features. Sets provide a way to enquire if a specific element is "in" the Set, and Maps provide a way to retrieve an element using a key.

Just as with the complexity and variety of method calls, it is not common for template languages to directly support all of these modes of access. Those template engines that do provide such access do so by handing off the expression to a more general-purpose programming language. What is more common, however, is to treat all such collections as being simply a series of values that can be iterated. While this ignores the specific capabilities of each of the different types of collection, and many Set and Map types cannot guarantee any particular ordering, it does at least allow the values to be included in a target document. Where the order of values is important, a typical solution is to copy the values into an ordered collection before placing that into the context.

Accessing values from a collection as a sequence typically requires some form of loop. In all of the template engines in this cohort except *Stringtree* and *Solomon*, this is done using using an external control structure (see Section 4.2.3) rather than a placeholder value expression.

In *Stringtree* and *Solomon*, which largely share the same syntax, rendering values from a collection as a sequence uses the `*` operator. The essential function of this operator is to fetch each item from the collection in turn and render it using a named template. As an example, to render the values from a collection stored in the context with the name "prices" using a template named "price" for each one would look like Listing 4.7.

```
${prices*price}
```

Listing 4.7: Stringtree loop example

Each element in turn will be added to the context using the special context value name "this" and the specified template will be rendered. *Stringtree* always requires a template to be specified even if all that is required is to emit the textual value of each element with no adornment. In such cases, a template must be used containing just a single placeholder, as shown in Listing 4.9.

```
${this}
```

Listing 4.8: Stringtree loop variable access

Solomon provides this as an implicit internal template that is used if no template name is specified.

```
${prices*}
```

Listing 4.9: Solomon loop with implicit subtemplate

In many natural language situations a separator will be needed between elements. For example, if the above "prices" collection contains three values "9.99" "12.99" "15.99", the desired output might be as shown in Listing 4.10.

```
9.99, 12.99, 15.99
```

Listing 4.10: Collection elements separated by commas

In this case, the separating commas and spaces cannot simply be included in the template used to render each element, or there would be an extra one at the start or end of the list. For this scenario, both *Stringtree* and *Solomon* allow the specification of an additional template that will be rendered between the items in the collection. In this case the main template would look like Listing 4.11.

```
${prices*price/comma}
```

Listing 4.11: Solomon loop separator syntax

and the "comma" template would look like Listing 4.12.

```
,
```

Listing 4.12: Solomon loop separator template

Although *Stringtree* and *Solomon* use largely similar template languages, *Solomon* adds support for a literal string value as a separator rather than always requiring an extra template inclusion. An example is shown in Listing 4.13.

```
${prices*price/','}
```

Listing 4.13: Solomon literal separator syntax

Note that neither *Stringtree* nor *Solomon* are capable of more advanced list rendering behaviour, such as adding "and" before the final element or breaking and indenting long lines. In such cases, it would be necessary to generate the desired output in the host programming language either as a separate context value, field, or method as discussed above.

Template languages that do not include direct support for such list separators usually require the use of if-then operations (see below) within the subtemplate. Some template languages provide specialist expressions for determining if an element is the first or last of an iterable collection. For example, to generate the same output as above, *Stringtemplate* uses a complex syntax reminiscent of classic list operations involving treating the "first" and "rest" of the iteration separately as shown in Listing 4.14.

```
$first(prices):{p|[$p$]}$$rest(prices):{p|,[$p$]}$!
```

Listing 4.14: Stringtemplate separator syntax

If-Then Operations While many values available from the context when expanding a template are intended to be rendered into the final document, some instead serve as "flags" to indicate what should be rendered, or not. Just as with list rendering, most template engines in this cohort rely on external control structures for this purpose while *Stringtree* and *Solomon* use an internal control structure in the form of a placeholder expression. In each case, the context value is tested for some notion of "truthiness" allowing different output for the "true" and "false" cases. Individual template engines differ on what counts as "true" and what counts as "false". All the template engines in this cohort agree on the meaning of system Boolean values but differ on whether, for example, null values count as a false or an error, or if any numeric (e.g. 0 and 1) or textual values (e.g. "true" and "false", or "True" and "False") are also treated as Boolean values.

Stringtree and *Solomon* allow the specification of either quoted literal text or the name of a template to be rendered for the "true" and "false" cases. Examples for various template languages are given in the discussion of control structures in Section 4.2.3.

Arithmetic and Comparative Operations Arithmetic and comparative operations such as `+`, `-`, `=`, and `<` are very common in general-purpose programming languages but relatively rare in template languages. As with many such advanced features, the template engines which do support these kinds of expressions usually so by handing off parsing and evaluation to a general-purpose programming language. Other template engines usually require such expressions to be pre-calculated and stored in the context as separate values.

Implicit Element Access In some circumstances, such as when iterating through a collection, a template language needs a way to refer to the "current"

element. Without this, whatever text or template values are rendered for each element would all be the same. This could be considered as a special case of context value access, but with the need to access an implicit, and thus unnamed context value. Template languages vary widely in how they address this issue. Some avoid it by simply not implementing the kinds of control structures that might need it; some use specialist control structures, or placeholders; while others create some kind of pseudo context value with a predefined name. *Handlebars*, for example, uses a special placeholder `{{.}}` for this purpose, while *Stringtree* and *Solomon* follow the lead of the underlying Java language and create a pseudo context value with the reserved name "this". *Freemarker* also creates such a pseudo context value, but requires that this value is given a name using an "as" keyword as part of the iteration directive as shown in Listing 4.15.

```
<#list prices as sku>${sku} </#list>
```

Listing 4.15: Freemarker loop syntax

4.2.8 Quoting and Escaping

Most template language designers have considered the possibility of clashes between document boilerplate content and the characters chosen to delimit placeholders and control structures and included some mechanism to "escape" any such characters that appear in the document content and prevent them being processed by the template language parser. The most common way to escape problematic characters is to prefix them with a different "escape character" which is recognised by the template language parser. A popular escape character is \, which happens to be the character used for the same purpose in many programming languages including *C* and *Java*.

There are also situations that may occur within template placeholders and control structures where a character within some literal text may confuse the template language parser. Examples include attempting to "include" a template file or attempting to evaluate the value of a placeholder with significant characters in its name. In such cases template languages vary in their behaviour. Some simply ignore the problem, requiring the user to only use names and literal text that are "template safe"; some make use of the same escaping mechanism used for problematic characters in boilerplate text; and others require some form of "quoting" to distinguish names and literal text from template instructions.

In common with typical programming languages, quoting of names and literal text is usually done with pairs of ' or pairs of " characters depending on the particular template language. This, of course, raises the issue of what to do if the

quoted text contains one of these quoting characters. Template languages vary on their approach to this, too, and may either ignore the problem, use an escape character, or take the same approach as English and allow ’ within text quoted with " and " within text quoted with ’.

Some template engines require quoting of literal text or template names in placeholders and control directives even if they do not contain problematic characters, while others quoting is optional and only applied when needed.

4.2.9 Ease of Reading and Editing

Template languages vary in readability, just as general-purpose programming languages do. However, template languages have the additional issue that template constructs such as substitution placeholders, processing directives, and control structures have to co-exist with arbitrary boilerplate text. The choice of delimiting characters can make some difference to readability, but arguably it is the representation of processing directives and control structures that has the largest effect on the readability and therefore the ability of designers and copywriters to understand and work with the templates. In the cohort of template engines for this study, most of the languages use mixture of single and multi-character symbols and descriptive words such as “if”, “else”, “list” or “each”. Two template languages in this cohort, *Stringtree* and *Solomon* are entirely symbolic, using single characters to indicate directives and control structures.

All the descriptive words used by template languages in this cohort are in English. Just as with a programming language, any attempts at descriptive wording ties the template language to the vocabulary and grammar of a particular natural language. Less commonly considered, however, is the impact of the choice of symbolic characters and delimiters on the ability of users to create and edit templates. Erz (2023) notes that the { and } characters, which appear frequently in the template languages in this study, are difficult to type on a German keyboard, for example.

4.2.10 Fragility

When creating anything, in this case a template to be processed by a template engine, there is always the chance of making a mistake. Template engines and their associated template languages also differ in their tolerance for such mistakes. The larger the number and variety of such potential accidents indicates the *fragility* of the template language. The more fragile a template language, the more work

a template engine has to do to detect and report errors, and the more work a template author has to do to ensure that everything works correctly. The number and variety of cases where the template engine does not identify and report such an error, or does not report it in a way that helps a template author to solve the problem, indicates the fragility of the template engine itself. Selecting a less-fragile template language will usually make it easier to code a less-fragile template engine.

The *Handlebars* implementation of the conditional example introduced in Section 4.2.3 requires that the name of the context variable ("stock") is repeated four times with subtly different syntax and meaning. If whoever is creating the template accidentally introduces a typo into one of those names, the whole control structure will be invalid, and the results of expanding the template will be very far from what is expected. In *Handlebars*, in common with several other template engines in this cohort, a missing or null context value is treated as "false". If the template author accidentally types or copies an initial `#` into the section that should contain `^stock`, then the template will generate "AvailableUnavailable" for in-stock items, and nothing for ones that are out of stock.

JTE is unusual among the template languages in this cohort in that it requires all context values used in the template to be defined and allocated a type in a preamble at the start of the template. For anything other than basic types, this preamble also has to contain an "@import" directive for the class that defines the type. This introduces a range of possible additional failure modes, from simply forgetting to update the preamble when the template is edited to mistakes in the name or type of the list of value names and types, to unexpected behaviour if a code change elsewhere in the project affects the type of a context value. For this reason, *JTE* is one of the most fragile template languages in this cohort.

4.2.11 The Challenges of Compound Values

In many cases the representation of a value in a template may seem simple, but contain hidden complexity. For example, consider a context object representing a customer, but which contains more details than just a "name" field. This customer object contains separate fields for "firstName", "lastName", and an optional "middleName". A simple approach to including the full customer name in a templated document might look (in *Jangod* syntax) something like Listing 4.16.

```
Welcome {{customer.firstName}} {{customer.middleName}} {{customer.
    lastName}}
```

Listing 4.16: Jangod customer name example

Although this is largely correct, in the case where a customer does not have a middle name it will produce incorrect output. Depending on the error behaviour of the particular template engine this might produce no output for the middle name, leaving a gap of two spaces between the first and last name, or it might generate an error or warning message, or it might refuse to render the document at all. While it might be possible to enclose the middle name placeholder and its associated whitespace within some kind of conditional control structure, this becomes increasingly complex. Extending this approach to deal with people with more than one middle name gets more complex still.

There are several potential programming solutions to this kind of problem.

One solution ignores that the name is conceptually a field of the customer object and creates a new independent context value for the full name. This requires an addition to the code that populates the context before expanding the template. In this approach, the conditional execution and string manipulation capabilities of the general-purpose programming language are used to construct a new "fullName" value that is placed into the context before expanding the template. This approach has the advantage that it works with the greatest variety of template engines, as it only needs the ability to expand a simple value placeholder in order to include the full name in the target document. It has the disadvantage that the code to construct the full name will be evaluated for every document that uses that context, even if that document does not make use of the full name.

In the resulting template this might look (in *Solomon* syntax) something like Listing 4.17.

```
Welcome ${customerFullName}
```

Listing 4.17: Solomon customer name example 1

A second solution is to add an extra field to the customer object to contain the combined full name. In many ways this is similar to the first solution, above, in that the full name must be generated before any template is expanded, but it has some key differences. The main difference is that this approach only works with template engines that support field access. This approach has the advantage that no change is needed to the code that builds the template context. Depending on the lifecycle of the customer object, this approach may also be error-prone,

as the full name field needs to be kept up to date when any of its components changes. This approach usually requires modification to the class that defines the fields and behaviour of the customer object. In many situations, such objects will be provided by other parts of the system and cannot be easily modified. There are approaches to address the issue of "adding" a field to an object of an immutable class, involving techniques such as inheritance or delegation (Connolly Bree and Cinnéide, 2020) but these are considerably more complex to implement than simply creating a new context value as described above.

In the resulting template this might look (in *Solomon* syntax) something like Listing 4.18.

```
Welcome ${customer.fullName}
```

Listing 4.18: Solomon customer name example 2

Another solution is to add a method to the class that defines the behaviour of the customer object. This method performs similar conditional and string processing to the first option but within the context of the customer object. This approach also has the advantage that no change is needed to the code that builds the template context. The code to construct the full name will never be executed if the full name is never used. The disadvantages are that it requires support for method calls in the template engine (which not all template engines provide) and the issue with potentially immutable classes as described above. If the full name is required multiple times in the same document, then this approach also has the disadvantage that the code to construct it will be executed multiple times.

In the resulting template this might look (in *Solomon* syntax) something like Listing 4.19.

```
Welcome ${customer.fullName()}
```

Listing 4.19: Solomon customer name example 3

Although in common programming languages, method calls may take parameters, this is very rarely supported in template engines. Those which do support this usually achieve it by handing off the parsing and processing of the whole expression to a general purpose programming language such as Java or JavaScript.

In the specific context of the Java programming language, which is used for all the template engines in this cohort, there is an alternative to including a direct method call in a placeholder expression. *JavaBeans* is a technology introduced in version 1.1 of Java that provides a way to treat certain methods as if they are object fields. Although making use of JavaBeans requires some code support, it is typically less than that required to support arbitrary method calls, and is thus

more commonly implemented. In JavaBeans any object method matching one of a small set of specific patterns can be treated largely as if it is a field access. For example, if an object provides a public method named `getFullName()`, the JavaBeans protocol will treat this as if it is a read-only field named `fullName`.

In the resulting template would look very similar to the approach of adding an extra field. In *Solomon* syntax, this might be something like Listing 4.20.

```
Welcome ${customer.fullName}
```

Listing 4.20: Solomon customer name example 4

In the relatively simple case of constructing a full name from parts, none of these advantages and disadvantages make much difference to overall system performance, and the deciding factors will be which features the template engine provides, and whether the context value classes are mutable. However, if the code in question requires considerably more processing, such as fetching values from a database or a remote system, then the impact of executing the code zero, one, or many times is also considerably more, which should affect the choice of approach.

4.3 A Feasibility Study

Even though there are several implementations of template engines mentioned in the literature, there are also many more available. Rather than attempt to gather detailed information about every possible software candidate before starting, the intention was to proceed incrementally, building a collection of data on a growing subset, using preliminary analysis and reflection to inform further research.

Before embarking on a potentially lengthy experimental process, it seemed prudent to attempt to determine whether such research would be likely to provide interesting results. The overall research goal was to help software developers make informed choices to minimise the environmental impact of their work. If measurable differences could be found even between a small selection of software components, then component selection or substitution could be a valid environmental strategy.

With thousands of template engines to choose from, the scope of any initial feasibility study needed to be constrained. Options included limiting the study by popularity, by application domain, by the preferences of specific software developers, by programming language, by template language, by price, or many other factors. Each of these options had theoretical advantages and

disadvantages. However, an initial feasibility study needed to be practical and provide results as soon as possible. Following the approach used by Laakso and Niemi (2008) and Zoio (2005) a study was planned that would select a small number of template components and construct a software test harness to run an identical set of tests on each of the components. There was no attempt to be exhaustive with the selection of components to test.

The aims of the feasibility study were twofold:

- To ascertain if this would be a reasonable and useful approach for identifying and comparing template components.
- To establish the scale of the variability in features and performance between a sample of available components.

Throughout the construction of the study, efforts were made to avoid or constrain "scope creep" (Heinze, 2014).

4.3.1 Selection of a Programming Language

To obtain results that could more easily be compared, it was decided to limit the feasibility study to include only template systems in a single programming language. The language *Java* (Oracle, 2018a) was chosen at the start of the research in 2018. At the time it was at the top of the Tiobe list of the most popular programming languages for new software in the world (TIOBE, 2018). While Java has slipped slightly in the Tiobe ranking since then (TIOBE, 2024), its continued presence over many years has resulted in a large body of Java software still in operation that needs to be maintained and improved. Java is freely available and runs on a wide range of hardware platforms. Java development tools are also available and familiar to the researcher, reducing both the need for training before commencing the study and the risk of mistakes or misunderstandings.

4.3.2 Selection of Template Engines

The next stage was to select an initial set of template engines for the feasibility study.

Template Engine	Reasons for Selection
StringTemplate[16]	Cited several times in academic literature
Velocity[17] *FreeMarker*[18]	Popular in lists of Java template engines on the web
Mustache[19]	Probably the most widely implemented style of template engine, with implementations in at least 45 programming Languages
Jangod[20]	An alternative implementation with syntax somewhat similar to *Mustache*
Hapax[21]	A template engine implementation with syntax partially similar to Mustache but with internal differences
Casper[22]	Processes placeholders using an embedded JavaScript interpreter
JMTE[23]	Pre-compiles templates into Java class files for execution
Stringtree[24] *Solomon*[25]	Implementations resulting from previous personal research in this area

Table 4.2: Reasons for selection of template engines

4.3.3 Construction of a Feasibility study

Keeping the feasibility study aims in mind, the following eight simplified "real world" evaluation scenarios were constructed:

Scenario S0: No substitutions ("control" case)

The job of a template engine is to read and combine a template with some variable data by replacing placeholders. However, not all template documents contain placeholders. Some are just documents to be served unchanged. The processing within the template engine will likely add some overhead. This scenario gives some indication of the processing overhead in each implementation.

Scenario S1: A single textual substitution

The simplest active case for a template engine is the identification of a placeholder and its replacement with a single constant textual value. The purpose of this scenario is to evaluate the performance of this common activity.

Scenario S2: A collection of textual values

A step up from a single value is the substitution of a collection (such as an array, a list, or some other iterable object) of textual values. This scenario requires the template engine to determine in some way that the value to be substituted is a collection, and to make some attempt at rendering the contents in order.

Scenario S3: A collection separated by commas

It is usual in English when presenting a list to separate the items with commas. Given the list (*ham eggs chips*) it might be typical to present it as `ham, eggs, chips` (or even `ham, eggs, and chips`, but that is less common in dynamically generated documents, and beyond the scope of this feasibility study). This scenario is an interesting challenge for a template engine, which is why it is a separate test from simply rendering the contents

[16] https://www.stringtemplate.org/
[17] https://velocity.apache.org/
[18] https://freemarker.apache.org/index.html
[19] https://mustache.github.io/
[20] https://code.google.com/archive/p/hapax/
[21] https://code.google.com/archive/p/hapax/
[22] https://code.google.com/archive/p/casper/
[23] https://github.com/HubSpot/jmte
[24] https://github.com/efficacy/stringtree
[25] https://bitbucket.org/efficacy-misc/emo/src/master/src/main/java/org/stringtree/solomon/

of a collection. Some template engines make it difficult to avoid placing an excess comma after the final item, for example.

Scenario S4: Include another template

Structural decomposition (colloquially known as *divide and conquer*) is a key practice of software engineering, and also applies to the design and construction of documents such as web pages from component parts. For this to work, a template system must be able to include within its output the result of processing another template. This scenario evaluates the effectiveness of a template engine at constructing compound documents.

Scenario S5 and S6: Show a value if a boolean value is *true* or *false*

Conditional execution is another key aspect of software. In the case of template processing it is a common requirement to show a true/false value from the data as text. From a single letter or a glyph such as an emoji or checkmark (✓) to a swathe of boilerplate or a whole extra template, the process is the same. If the value is true, show the specified text. Some template languages make a big issue of the difference between showing some text if true, but nothing at all if false, and showing alternative text such as ✓for true and × for false. These scenarios evaluate the effectiveness of a template system at converting boolean values.

Scenario S7: Call some code

Although Parr considers this a "violation" of the principle of separation of logic from display, many template systems support the execution of arbitrary code during the rendering of templates. Techniques for achieving this vary widely, but with the Java ecosystem there is a standard way of invoking code from other contexts: the JavaBeans API (Oracle, 2018b). Use of this API allows systems such as template engines to execute an object method by treating it as accessing a data field. Typical uses for this facility in a template is to include data that is expensive to pre-calculate or may not be known in advance. This scenario evaluates the effectiveness of a template system at using the JavaBeans API.

4.3.4 Implementation of a Feasibility Study

One of the characteristics that separates template systems is the variation in their template languages. An implication of this is that templates cannot usually be re-used to evaluate different template engines. These distinctions, and some approaches to minimising their impact, are explored in more detail in Appendix E. Likewise, template engines provide a variety of software interfaces.

For best compatibility of test results tests should be as similar as possible, yet the nature of the different template engines requires many differences in implementation. To resolve these tensions, an architecture was chosen that used the strategy pattern (Gamma et al., 1994) to enable isolation of specific differences between the code for the various template engines. This pattern is illustrated with a UML class diagram [Figure 4.3.1], shown during development with three `TemplateSystem` implementations (*Stringtree*, *Solomon*, and *Casper*).

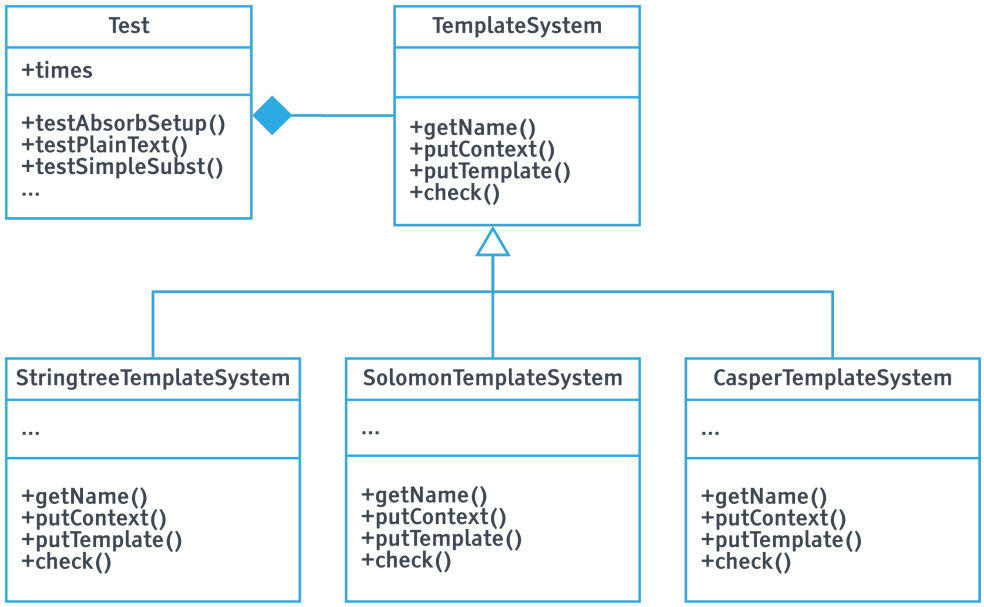


Figure 4.3.1: Class diagram illustrating use of the strategy pattern

Each experiment run was driven by instantiating an object of the `Test` class that in turn created instances of each of the specific driver classes, which it refers to through the `TemplateSystem` abstraction. Once all the Template `TemplateSystem` objects had been created, the controlling `Test` object ran the series of template scenarios against each `TemplateSystem` implementation. This process is illustrated with a simplified UML sequence diagram [Figure 4.3.2], shown with two `TemplateSystem` implementations (*Stringtree* and *Casper*) and two test scenarios ("plain" and "single"). The full performance test suite included `TemplateSystem` instances for all of the template engines and the full set of test scenarios.

As each concrete strategy class was coded and tested, it was integrated with a test harness written using the JUnit test library[26]. When all implementations were complete, code was added to compare generated output with expected output from each test for each test harness, and to time multiple runs of each such test. Tests were timed using the Java `System.currentTimeMillis()` method, which

[26] `https://junit.org/`

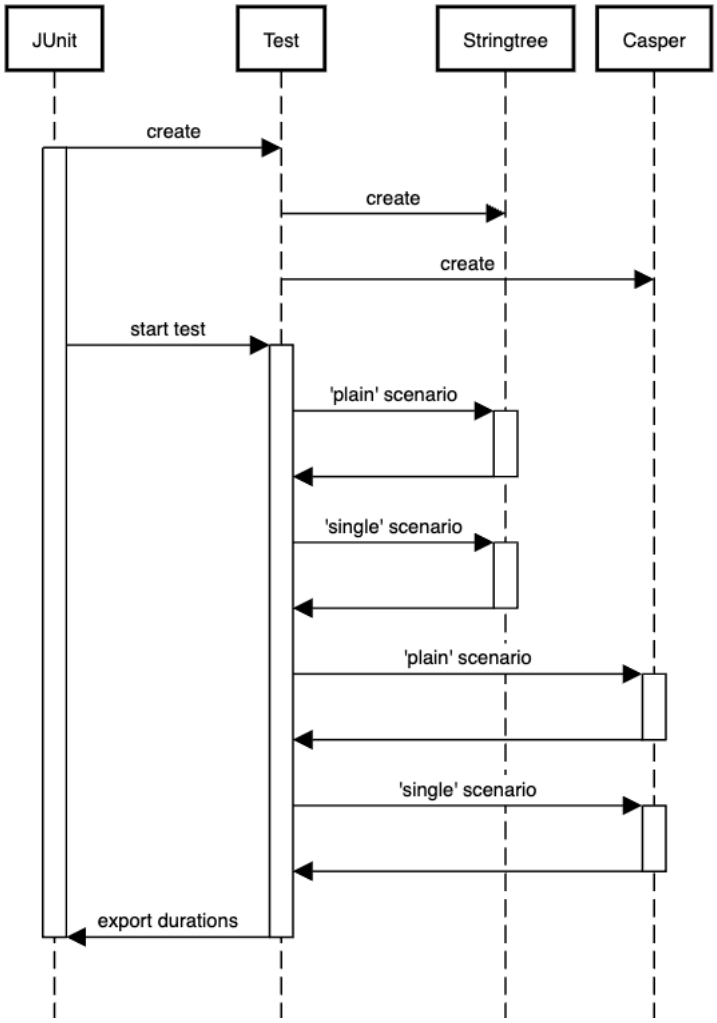


Figure 4.3.2: Sequence diagram illustrating the performance test process

uses the system "wall clock" rather than attempting to measure only time used by specific processes or threads.

The code for the test harness underwent several changes as the template implementations were added. Most commonly this was due to differing assumptions between implementations. For example, many template engines expect to fetch templates from a local file system but some prefer other means, so the abstract definition of the strategy implementations had to change to support a wider variety of such sources. As much as possible, however, such changes were kept in the common part of the code rather than being duplicated in template-specific classes.

Despite copious reading of documentation, source code and web articles, it was not possible to coerce all template engines to generate fully correct output for all test scenarios. Such failures are noted in the results, with the NOTMATCHED

annotation or with a duration in brackets, but it was decided not to exclude these test scenarios from the timed tests.

As testing progressed it became apparent that performance of the differing template engines varied widely, so the timed part of the experiment was tuned to give all the template engines a chance to complete in a reasonable time. Eventually a value of 10,000 repetitions was settled on as an effective number. This gave largely repeatable results for each template engine.

4.4 Feasibility Study Results and Analysis

The success of the test scenarios is given in Table 4.3, and the timings for 10,000 runs of each scenario are given in Table 4.4.

The mean test durations for each template engine are shown as a bar graph in Figure 4.4.1. Problems with this visualisation are discussed in Section 4.4.1 and a potentially more useful visualisation (without the outlier *Casper*) is shown in Figure 4.4.2.

	S0 No Subst	S1 Single Text	S2 Collection	S3 Separated	S4 Include	S5 Bool True	S6 Bool False	S7 Call Code
Solomon	✓	✓	✓	✓	✓	✓	✓	✓
Stringtree	✓	✓	✓	✓	✓	✓	✓	✓
JMTE	✓	✓	✓	✓		✓	✓	✓
Mustache	✓	✓			✓	✓	✓	✓
Stringtemplate	✓	✓	✓	✓	✓	✓	✓	
FreeMarker	✓	✓	✓	✓	✓	✓	✓	✓
Velocity	✓	✓	✓		✓	✓	✓	✓
Hapax	✓	✓	✓					
Casper	✓	✓	✓	✓	✓	✓	✓	✓
Jangod	✓	✓	✓			✓	✓	

Table 4.3: Test successes by template engine and scenario

It is reassuring that all the template systems in the study were capable of processing both plain text (S0) and simple substitution (S1) cases. However, these were the only scenarios that all the template engines passed. For all the other test scenarios there was at least one template engine that could not

	S0 No Subst	S1 Single Text	S2 Collection	S3 Separated	S4 Include	S5 Bool True	S6 Bool False	S7 Call Code	*Mean*	*SD*
Solomon	4	4	17	2	8	7	2	20	**8**	**6.87**
Stringtree	133	62	119	77	109	83	56	53	**86.5**	**30.43**
JMTE	9	13	46	3	*(30)*	10	9	11	**17.4**	**14.30**
Mustache	396	275	*(631)*	*(461)*	453	361	377	396	**418.8**	**103.36**
Stringtemplate	239	243	322	150	350	158	92	*(40)*	**196.6**	**108.42**
FreeMarker	25	17	29	34	40	22	37	28	**29**	**7.75**
Velocity	252	168	282	*(250)*	205	178	124	114	**196.6**	**61.58**
Hapax	64	79	124	*(58)*	*(117)*	*(43)*	*(49)*	*(55)*	**73.6**	**30.88**
Casper	19311	15878	23783	18226	27997	21292	21572	21048	**21138.3**	**3,659.57**
Jangod	145	158	187	*(270)*	*(126)*	127	117	*(313)*	**180.5**	**72.94**
Range	**19307**	**15874**	**23766**	**18224**	**27989**	**21285**	**21570**	**21037**		
Relative Diff.	**4827.75**	**3969.50**	**1399.00**	**9113.00**	**3499.63**	**3041.71**	**10786.00**	**1913.45**		
SD	**6063.46**	**4986.19**	**7461.39**	**5719.58**	**8804.06**	**6699.23**	**6792.19**	**6621.10**		
SD (exc. *Casper*)	**133.51**	**101.89**	**195.61**	**154.78**	**151.65**	**113.90**	**113.96**	**140.82**		

Table 4.4: Total time (ms) taken to complete 10,000 runs by template engine and scenario

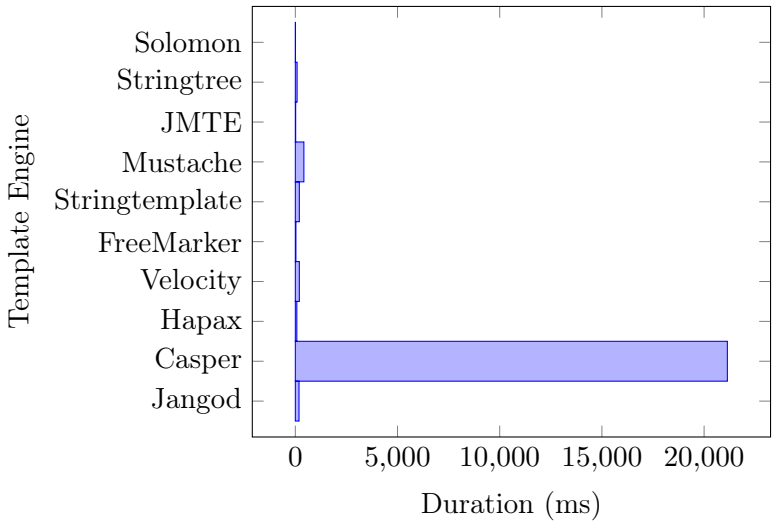


Figure 4.4.1: Mean test duration by template engine

manage it. There were only four template engines in the sample that correctly rendered every scenario: *Solomon*, *Stringtree*, *Freemarker*, and *Casper*. Given that these scenarios were chosen as representative of common situations in professional software development, it is surprising that there was so much variability in features. The documentation for all the implementations makes some kind of claim of being "powerful", "flexible" or "suitable for a wide range of projects".

Consider, for example, the *Hapax* template system that failed five of the eight tests. Its documentation states:

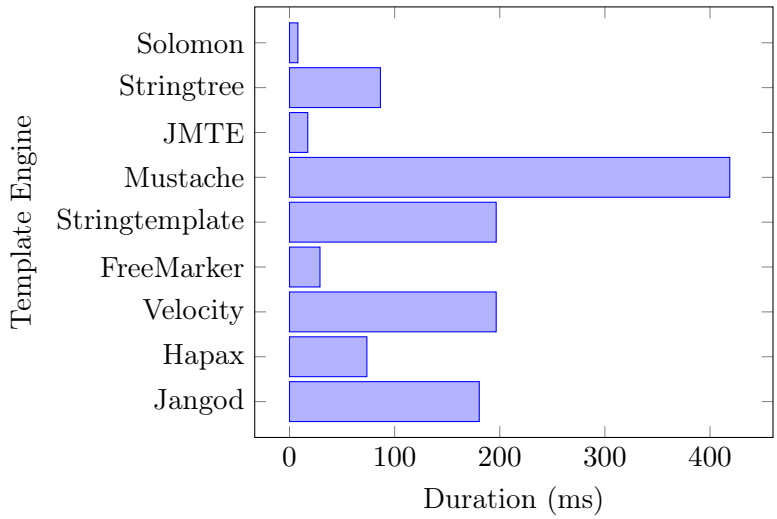


Figure 4.4.2: Mean test duration excluding *Casper*

> Hapax is a simple but powerful text templating library for Java. Hapax is suitable for constructing text output from Java code. The syntax is similar to Google's ctemplate library, and emphasizes the separation of logic from presentation. Hapax was designed to be easy to use and have minimal dependencies. Hapax does not depend on any existing web framework, and is suitable for use in servlets, scripting languages (Scala, Groovy, etc), and server-side applications.

Nowhere in this does it mention any limitations, making comparing implementations on documentation alone a tricky proposition. Attempts have been made to compile comparative lists of such software by features (Wikipedia, 2018) but interpretation of the exact meaning of each feature can be loose, and there would potentially be great benefit in defining a comprehensive set of such benchmark scenarios that could be used to compare and evaluate even software that is not found in such lists. The production of such benchmark scenarios would be difficult, however, because of the wide differences between template languages. There is currently no single "meta language" in which scenarios can be expressed for multiple template engines. This problem is explored further in Appendix E.

In many cases the limitations of the template engines seem closely tied to the choice of template language. The minimalist language common to many *Mustache* implementations, for example, deliberately avoids constructs such as loops, in favour of performing such logic in application code before invoking the template engine. While this can be a workable strategy, and satisfies Parr's (2004) desire for separation of concerns between view and model, it also has some potential drawbacks. Preparing values requires extra processing in the application code,

which partially disguises the processing cost of generating pages compared to more competent template engines. It also requires that every possible configuration of data for the page is pre-calculated, including "expanding" collections. While expanding lists of textual or numeric values is relatively simple, this approach becomes much more complex in the relatively common case where each entry in the list needs to be processed using a secondary template in order to generate the desired result. This approach also denies the template engine any possibility of automatically re-using intermediate values or using "lazy evaluation" strategies to avoid unused calculations, for example when Boolean conditions remove the need for a value to be rendered. Such choices have considerably more impact when the same application code is used to evaluate a wide variety of templates, each with different data requirements. In common with many software development habits, which prioritise development effort over execution time, it can be easier for developers to produce general-purpose code that pre-calculates everything required by every possible template, even though many of those values will not be required for any given rendered page. This can add extra, unneeded, processing cost to every page.

There is also an anomaly in the timings. In Table 4.4, it appears that several of the template engines take longer to process a plain text file than a simple substitution. While this is possible, it seems somewhat unlikely, so the experiment was temporarily adjusted to run the tests in a different order. Whichever test runs first seems to take longer. It was hypothesised that this was due either to interaction between the different template engines, or to some form of "warm up" behaviour. Similar issues were encountered by Carvalho, Duarte, and Gouesse (2020). These issues cast doubt on the accuracy of the feasibility study model for performance comparisons, so an improved performance study was constructed to eliminate these kinds of influences. This improved performance study is described in Section 4.6.

Despite these limitations, the feasibility study serves its purpose of determining whether the performance differences between different template engines were worth investigating further.

The software "test bench" for these measurements was constructed using the strategy pattern (as shown in Figure 4.3.1) with all the template engine libraries loaded into memory at the same time. This made running the experiments much simpler, as they could be performed by a simple loop through a collection of *TemplateSystem* objects with no need to stop or start the code, or to add and remove external code blocks. However, this approach also required some careful coding to deal with potential clashes and incompatibilities between the separate template libraries and their transitive dependencies. The version of Java used

for these experiments did not provide any form of separation between libraries beyond class and package names. Any problems with any of the template engine implementations (such as a syntax error in a template specification) that caused the template engine to "crash" or raise an exception, would abort the whole test run. It is not clear whether there was any more subtle interference between the separate libraries that may have affected the results.

The approach of treating this code as a "throw away" experiment and writing only the minimum code necessary to perform the experiments had an impact on the usability of the software. Result values were simply printed out as soon as they were calculated, without performing any analysis or data formatting. This required manual copying into analysis software to determine derived results such as the mean duration. Configuration parameters such as the collection of template engines to be tested and the number of times to invoke them were hard-coded into the software, which meant that any change required editing and re-compliation of the experiment code.

Finally, it is important to note that this feasibility study only measured two things: the ability to correctly render some specific scenarios, and the time taken to perform them. While some *guesses* may be made about what that might mean for overall energy usage, this study does not say anything definitive about the energy usage of the different template engines.

Later sections in this chapter and later chapters in this thesis address these issues. Improvements to the construction and operation of the test framework and more detailed comparison of component performance are discussed in Section 4.6. An apparatus for comparing the energy usage of software is developed in Chapter 5 and the energy usage of template engines is explored in Chapter 6.

4.4.1 Analysis of Individual Template Engines

Casper takes so long to perform the experiments that it makes Figure 4.4.1 unusable for comparing the performance of the other template engines. By removing Casper from the results (see Figure 4.4.2) a more nuanced view of their relative performance can be seen.

In Figure 4.4.2, template engine performance can be seen to cluster into a few groups. *Mustache* is in a group of its own, with a mean duration of 418.8ms, taking roughly twice the time as the engines in the next group. The next group contains three template engines: *Stringtemplate*, *Velocity*, and *Jangod* all of which take around 200ms A third group includes *Hapax* and *Stringtree*, which both take

a bit less than 100ms to complete the run. The final, fastest, group includes *Solomon*, *JMTE*, and *Freemarker*, all taking 30ms or less.

It is clear from these results that the performance of different template engines varies widely. Examining the code and templates of the different template engines shows some interesting characteristics.

- *Casper* has obviously been designed to be as flexible as possible, going so far as to start a complete JavaScript interpreter for every page generated, and discarding it when the page is completed. This imposes a relatively huge burden on generating regular web pages but does potentially provide some facilities unavailable in most other systems. It is important to note that this feasibility study was designed based on common web page scenarios, and thus does not include any cases where Casper would be the only applicable solution.

- *Solomon*, on the other hand has been coded with an emphasis on speed. It uses a relatively non-standard template language (see Section 4.2), optimised for simple and fast processing rather than easy readability and the internal design minimises time consuming operations such as parser-backtracking, buffering and data copying.

- *Stringtree* uses a similar template language to Solomon but did not achieve the same performance benefits. When Stringtree was coded, performance was not the primary goal.

- Two other template engines (*JMTE* and *FreeMarker*) achieve speeds almost as fast as *Solomon*. *FreeMarker* was also one of the few template engines that also managed to produce correct output for all the scenarios, which indicates that it is capable as well as fast. *JMTE* and *FreeMarker* have apparently also been coded with speed in mind, although their template languages are more complex to parse than the language used by *Solomon*, which may contribute to their slower performance.

- Simplicity of a template language is no guarantee of performance, however. *Mustache* has one of the simpler template languages but performed the worst of the non-*Casper* template engines. *Hapax* and *Jangod* have template languages that are similar in many ways to *Mustache*, but exhibit different performance characteristics.

- The remaining two templates in this study (*Stringtemplate* and *Velocity*) show similar performance to each other even though they have different template languages and design goals.

The difference in template processing speed between the template engines in this study indicates that even when the platform and the task (generating specific output text) are fixed, choices during the design of template languages and template engine implementations can significantly affect performance.

If the above experiments were used to select a template engine for a project based on performance, then the three template engines in the fastest group would form the short-list. Selecting between these three would then depend on further factors. *Solomon* is clearly the fastest. It is also able to correctly perform all the scenarios, so is the obvious choice. However, Solomon uses a slightly unusual template language (see Section 4.2) and this could require training that might hinder adoption. JMTE is the second fastest, but could not correctly perform all the scenarios. In development situations where the missing features are not important, however, then JMTE could be a good compromise. *Freemarker*, although it is up to 18 times slower than Solomon for some cases, was the only popular template engine to correctly perform all the scenarios. *FreeMarker* also has the advantage of a well-documented template language and wider adoption than the other two, so would also make a good choice for a more conservative development team. Based on their performance it would seem a poor choice to select any of the others.

4.5 Feasibility Study Discussion

A key finding to arise from the feasibility study is the surprising variety in capabilities and performance between purportedly similar software libraries. The largest range in performance for a single scenario was for the Boolean False scenario between *Casper*, which took 21572ms to complete the 10,000 template expansions, and *Solomon*, which took 2ms. This is a difference of 10786 times. An application using *Solomon* for this scenario would be able to process over 10,000 documents in the time it takes *Casper* to process one. To handle a similar load, the machine running *Casper* would need to be over 10,000 times as powerful. If that was not possible, multiple machines would be needed.

Even averaged across the range of different scenarios, the mean time taken for *Casper* to perform the suite of scenarios (21138.3ms) is still over 2,500 times more than the mean time taken by *Solomon* (8ms).

Selecting a template engine for use in a software project is a difficult task. Typical open source software documentation has no quantitative specifications. All the components evaluated in this study are free software, so the traditional

approach of minimising the purchase cost does not apply. All the components are available with licences that permit inclusion in commercial software, so considering licence terms is also no use in selecting between them. Despite their free availability, and with no direct financial incentive to increase "sales", they often provide documentation that reads like optimistic and unsubstantiated advertising copy. Such documentation as is provided tends to concentrate on listing and describing specific available features, covering everything else with blanket subjective statements such as "fast", "powerful" or "flexible". This paucity of information makes it hard for developers to make informed choices.

The fact that one template engine, *Solomon*, was able to correctly perform all the scenarios in a much faster time than any of the others when they are all written in the same programming language and running on the same platform, shows that the choices made in the design and programming of the different template engines in this study have had a huge impact on the performance and capabilities of these supposedly equivalent components. In this specific case, one reason could be that *Solomon* was written to minimise multiple handling of template characters. Whenever possible, input characters are passed directly to the output document without buffering. The characteristics of the *Solomon* template language (discussed in more detail in Section 4.2) help reduce the need for buffering.

In conclusion, the results of this feasibility study show that there are surprisingly large differences in capabilities and performance between superficially similar solutions to a specific class of problem. The study provides data unavailable from the on-line documentation for the components being considered. This in turn suggests that a change in software development behaviour, both when writing software and when selecting components or libraries, could potentially make a noticeable difference in performance, and by implication the processing requirements and resource usage of large software systems such as the world wide web.

In particular, the following specific conclusions will be taken forward from this feasibility study:

- The differences between this sample of components are large enough to be worth proceeding with the research.
- Choices made during design and development of software, and when selecting components, can have a large impact on the performance of the final product.
- The methodology of constructing a range of representative scenarios and

using them to compare the different components is fundamentally sound, however, improvements are needed to the implementation of the tests

- The template languages used by the different template engines vary widely enough that further research is needed on ways to simplify the creation of equivalent templates for a range of template engines.
- Further research is needed to determine what impact this wide variety in performance might have on energy usage.

4.6 An Improved Performance Study

The following sections address the shortcomings of the feasibility study to enable a deeper exploration of the performance characteristics of template engine components.

- The problems with the initial feasibility study are summarised in Section 4.6.1.
- The changes to the component landscape in the time since the design of the feasibility study are considered in Section 4.6.2.
- The process for selecting template engines to include in an improved performance study is discussed in Section 4.6.3.
- Specific improvements to address the problems with the software used for the feasibility study are described in Section 4.7 and Section 4.8.

4.6.1 Recap of Problems With The Feasibility Study

The feasibility study was conducted to gain an overview of the scale of the differences between a selection of components. While the results were clear, the individual performance measurements were neither rigorous nor comprehensive. In particular, the following issues were discovered:

- **Problem 1** All the template engines were loaded into memory at the same time, which could cause influence between different components.

- **Problem 2** Clashes, both between class and package names of components and where components required different versions of the same library.
- **Problem 3** Configurations were all hard-coded, requiring editing and re-compilation of the code
- **Problem 4** An anomaly in the timings where whichever scenario was run first took extra time.
- **Problem 5** Scenarios were always run the same number of times, and therefore showing only how the components compare at that load.
- **Problem 6** Results output was simplistic and difficult to process.

4.6.2 Changes to the Component Landscape

All of the template engine components evaluated as part of the feasibility study were open source and free of charge and were chosen to be broadly representative of the available software at the time. However, in the several-year period between the feasibility study and the later measurements of the components (see the research timeline in Section 1.3), the landscape of software in this niche had changed and developed. This is a natural process in open source software (Sonatype, 2023) (Xie, Chen, and Neamtiu, 2009). While the template engines evaluated in the feasibility study were all still available, many had not been updated or were no longer in popular use. Several other template engines had appeared or increased in popularity in the intervening time. Research was needed to determine an appropriate set of current template engines to evaluate in more detail.

4.6.3 Selecting Template Engines to Compare

For this stage of the research a more systematic approach was needed. This comprised the following initial steps to build a list of candidate template engines:

1. Limit the search to the *GitHub*[27] code repository
2. Perform a search for the term "template engine".
3. Limit the results to code in the Java language.
4. Ignore repositories that are not textual template engines in their own right such as adaptors, plugins, examples, game engine templates etc.
5. Ignore special-purpose template engines with specific output formats such as PDF, DOCX, and SQL.

[27] `https://github.com/`

6. Ignore template engines that are obviously based on the "Template Engine Kata" from Koskela (2007).
7. Ignore template engines that are "forks" of other template engines in the list
8. Ignore template engines with documentation in languages other than English.

This process resulted in a list of 132 template engines. The full list with names and GitHub URLs can be found in Appendix A. Note that, as mentioned above, the open source software landscape is continually changing, so following the same process at a later date would produce a different, and probably larger, set of results.

Even having eliminated the obvious cases where template engines appeared to have been created purely as a learning aid, there were still many in the list that were either incomplete, unusable, or would be unlikely to be used by anyone other than the original developer. To get perspective on which template engines to consider further, some gauge of popularity was needed.

Unfortunately, there is no single resource that indicates the popularity of such software components. Commercial products do not usually reveal their choice of components. Some code repositories provide indications of the number of times a particular component has been downloaded, or how many times it is used by other projects in the same repository. While such statistics can give clues as to the popularity of components, they are not reliable. For example, download counts are easily skewed if some organisation has a policy of re-downloading every component for every build, which may happen many times per day. Statistics on the usage of components by other components are arguably more useful, but are limited to dependencies that the repository is aware of, and do not include code that is not stored in a public component repository.

On the wider internet, there are a range of personal and collaborative websites containing recommendations and comparisons. These websites can provide an indication of popularity. Individually, each such website is of limited use. The information provided may not be objective and has probably never been reviewed. The site may be out of date, presenting information on obsolete versions of components. The authors may have misunderstood the capabilities of individual components, or be unaware of the existence of some, or lack the skills or experience to fully evaluate the options. In some cases the information may be deliberately or accidentally biased.

When a range of such resources are considered as a whole, however, they can provide a kind of *zeitgeist*, but there are still potential problems. While outright

collusion is less likely when a broad enough group of websites is included, there is still the possibility of an "echo chamber" effect (Cinelli et al., 2021), in which the creator of each website gains their information mainly from other similar websites. Such effects tend to reinforce the popularity of well-known choices and exclude newer or less-familiar opinions. The main reason to include this source of information in this research is because it is the main public source of information available to software developers. Individual developers will often have other sources, such as personal experience, the opinions of colleagues or friends, or instructions from an employer or client, but such sources are largely private and were unavailable for this research.

Public information sources on the web can be grouped into categories, with the elements in each category having different characteristics.

Data from component repositories As discussed above, the details provided by component repositories vary and may not be fully representative, but within their scope they are authoritative.

Open Forums and Discussion sites These kinds of sources typically consist of questions and answers, although there are often multiple answers to a question and the responses may conflict with each other or digress from the original topic. Typical examples include sites aimed at software developers such as *Stack Overflow*[28] and *The Java Ranch*[29] as well as more general question-and-answer sites such as *Quora*[30] and *reddit*[31]. Some such sites have mechanisms intended to emphasise credible answers, or at least credible participants. Others make no such judgement.

Crowd-sourced Comparisons In crowd-sourced websites, multiple users contribute to collecting and organising information and provide some notion of consensus. The most well-known website of this nature is *Wikipedia*[32] but this category also includes other "wiki" style websites as well as review aggregators such as *Capterra*[33].

[28] `https://stackoverflow.com/`
[29] `https://javaranch.com/`
[30] `https://www.quora.com/`
[31] `https://www.reddit.com/`
[32] `https://www.wikipedia.org/`
[33] `https://www.capterra.com/`

Individual Opinions This category includes websites, blogs, videos, podcasts, articles, and social media posts predominantly created and managed by a single individual. The form of these opinions varies widely as does the scope. This category also includes articles or chapters in non peer-reviewed publications or collaborative websites, as long as there is an identifiable author. The key point with the sources in this category is that they originate from one individual and reflect that person's opinions, or at least what they would like to present as their opinions. In some cases there may be opportunities to engage with the author through comments or direct messages, which can help to justify or clarify their opinions.

Provider opinions The final category includes all information in which the creator, provider or vendor of a particular product has a stake of some sort. The most common example of this is documentation websites provided for users of their product or products, but this category also includes forums, discussion sites, and blogs owned or managed by a stakeholder in a particular product. Such sources are usually the most authoritative in regard to their own products, but have an inherent bias and cannot be relied on for information about competing products.

This research included sources from each of the above categories. A full list of sources consulted is given in Appendix C. In addition to the template engines in the original feasibility study, several potential new candidates were identified. Not all were technically suitable to be included in the comparison, but this was not discovered until an attempt was made to include them in the experiments. Details are given in Section 4.7.3.

4.7 Improvements to the Feasibility Study

The original feasibility study exhibited the six problems described in Section 4.6.1. Problems 1, 2, and 4 are all associated with loading all the template engines into a single memory space at the same time. To address this, the individual template engines were extracted into separately-compiled "dynamic plugins", loadable at runtime without recompilation. When running performance measurements, the improved performance measurement framework loads only a single plugin at any one time. the design, implementation, and use of these plugins is described in Section 4.7.1 and the following subsections of Section 4.7.

Problem 3 was the requirement to edit and recompile code in order to change even common configurations such as the number of template expansions to run,

or the template engine to measure. The improved performance measurement framework allows these configurations to be specified as command-line parameters which can be specified without changes to the code. The extraction of the number of template expansions to a command-line parameter allowed scripts to be written to run multiple performance measurements at different numbers of expansions, thus addressing problem 5. The design, implementation, and use of the improved performance measurement framework is described in Section 4.8.1 and the following subsections of Section 4.8.

The issues with the original output format of problem 6 were addressed by standardising output into a single CSV format for all measurements. This allowed results to be parsed and managed easily, supporting both combining and filtering of output files to suit the needs of analysis and results presentation. The improvements to, and the use of, the output data format are described in Section 4.9.1.

4.7.1 Extracting Template Engines into Separate "Plugins"

The testing and evaluation of software components requires a different approach to the normal process of component-based software development. When developing a software product that uses third-party components, the main aim is usually to integrate those components closely with the rest of the code, to form a single application code base. When evaluating a range of components, on the other hand, the aim is to keep them as separate as possible from each other to minimise unintended influence on the evaluation. The feasibility study described in Section 4.3 was developed using a traditional integration approach and raised several issues that needed to be addressed in order to reliably measure and compare the components.

To ensure complete separation between the different template engine implementations, it was decided to re-code the measurement framework to treat the template engines as "plugins" to be loaded into the framework at run-time and tested individually. The Java language provides support for dynamic swapping of code, subject to the following constraints:

- The code to be loaded must be in the form of compiled Java class files.
- The classes must have been compiled with a compatible version of the Java compiler.
- The plugins must all implement the same java interface.

- The classes to be loaded must be visible on the Java classpath.

All the selected template engines were written in Java and the source code was available. The availability of source code meant that they could be compiled to Java class files using the same version of the Java compiler as the test framework. This addressed the first two restrictions.

Even though all the template engines under consideration were written in Java, they all had different code with no interfaces or class names in common. This meant that they could not be called in the same way by the same code. To resolve this restriction, a "driver" was created for each template engine. Each driver presents the same interface to the comparison framework but understands how to setup and invoke a particular template engine.

The cohort of template engines considered for this study all use one of two ways of initiating the code for the template engine. The traditional approach in Java is to use the Java `new` operator to create an object of a named class. An alternative approach is to directly call a `static` method on a named class that itself creates an appropriate object. In each case, the result is an object with methods that may be called for typical template engine operations such as setting context values or expanding a template. The specific details of the methods available vary between template engine implementations, with some requiring further setup and configuration while others are immediately ready to be used to process documents. Similarly, the specific sequence of methods and parameters to use when creating a context and expanding a template to produce a destination document also varies between template engines.

To ensure that the API for each engine was called in the correct manner, each plugin driver implements a Java `interface` with two core methods: `init()` to perform whatever initiation and preparation is required by the template engine before it can be used; and `expand()` that can be used to combine a named template with a provided context to produce an output document.

Further details of the plugin driver implementation are given in section B.2.

4.7.2 Dynamic Loading of Plugins

In order to compare the behaviour of each template engine without overhead or interference from other template engine code or data, it is vital that each template engine driver is loaded independently. It is also highly desirable to be able to include new template engines without requiring changes to the code

of the comparison framework. In order to meet this requirement, the code for template engine drivers must be separate software, with the template comparison framework treated as a "black box". In this approach, knowledge should be strictly one-way. It is acceptable for drivers to make use of classes from the framework code, such as the `TemplateEngine` interface that all drivers must implement. However, it is not acceptable for the comparison framework to require knowledge of any classes from individual drivers as this would require those drivers to be present whenever the comparison framework is compiled.

The aim is that a template engine evaluation can be run with a template engine plugin specified as a command-line parameter. The comparison framework will then prepare and evaluate each comparison scenario and generate an appropriate destination document using the driver specified in the plugin. Unfortunately, in Java this process is not quite as simple as that would seem. In order to call methods on a driver, the driver must be instantiated as a Java object, which in turn requires that the class which specifies that object must be loaded first.

The Java language and virtual machine supports dynamic loading of classes, but with a catch. In Java there are essentially two ways to load a class. By far the most common is to refer to it by its fully-qualified name in the calling code. This approach, however, requires that the calling code know the full details of the package and name of the class to be loaded, which would violate the one-way knowledge rule, above. The alternative way to load a class involves techniques known as "reflection" and "introspection" that allow Java code to examine and use compiled Java objects without needing to know about their classes in advance. Reflection and introspection are relatively cumbersome to use, are known to be slower in use, and have a much wider range of failure modes than the regular way of using classes and objects.

There is, however, a way to gain the benefits of invoking a named class without knowledge of the details of the plugin. This requires the creation of a single class in every plugin that has the same fully-qualified name, in this case `plugin.EngineFactory`. When the comparison framework code is compiled, it is provided with a "dummy" implementation of this class to avoid compiler errors about a missing class. When the comparison framework is run to evaluate a particular template engine, the dummy class is removed from the classpath, and the identically-named class from the supplied plugin is used instead. The intention is that this impostor class is as minimal as possible, leaving all the real work to appropriately-named classes within the plugin.

Further details of the plugin driver factory implementation are given in Appendix B.3.

The key method in this class is `create`, which is responsible for creating a template-engine-specific driver object that implements the `TemplateEngine` interface. Once that object is created, it can then be used by the template comparison framework, which only knows about the methods provided by the `TemplateEngine` interface. The other method is not strictly necessary to generate templates, but aids observability of the comparison process and can be used in logging and error messages to indicate which plugin was in use when something happened. Note that as the shared interface has the same "leaf" name as the class being defined, it needs to be specified as a full-qualified name `shared.EngineFactory` to avoid conflicts during compilation.

The `create` method requires one parameter, a `File` object representing a folder containing stored templates. Most of the template engines in this cohort require their templates to be stored on a file system, sometimes with specific requirements for file structure and naming. Arguably, template engines would probably perform faster and use less energy if template storage was in memory, without the overhead of locating and loading a template from slower file storage. However, while some of the template engines under consideration do support other forms of template storage, the only template source supported by all the template engines is templates stored on a file system, so for a more equable comparison of template engine performance, all comparisons were done with templates stored in files.

While this approach of defining an identically-named class in every plugin addresses the problem of using plugin code without foreknowledge of the details of the plugin, it can cause some issues during development. Some Java development tools and integrated development environments (IDEs) prefer to load all the code for a project at once in order to build a search index and check for potential conflicts. Such tools do not work well with this style of coding. Happily, the Eclipse IDE[34] used for the development of this code allows development in a "workspace" that consists of several separate "projects", each of which contains its own independent namespace. Using Eclipse, the comparison framework and all the plugins could be worked on at once without the need to open, close, or switch projects.

The same approach to plugin loading and development was also used for the dynamic loading of template language drivers that define the characteristics of different template languages for use when generating test templates from an intermediate template. Templates for each of the cohort of template engines were generated from a single intermediate template specified in the intermediate language (see Appendix E).

[34] `https://www.eclipse.org/ide/`

4.7.3 Conformance Testing of Individual Plugins

Each plugin was developed and tested individually for conformance to the plugin specification described in Section 4.7.1. A conformance test framework was created using the *JUnit*[35] testing tool to load and exercise each method of a specified plugin to ensure that it operated correctly. Example code to test a template engine plugin is given in Appendix B.1. As will be seen later, the template languages used by the various template engines varied considerably, so the tests could not be identical but each one had to be adjusted to use the correct template language for the plugin being tested. A potential solution to this problem is discussed in Appendix E.

4.8 Replicating the Original Tests Using the New Plugins

Once a selection of plugins supporting a range of template engines had been created and tested, the next step was to replicate the feasibility study experiments using the new framework, to see how the results compared with the original. This involved re-working the original code to use the plugin API described above rather than the custom code that had been written for the different template engines in the original study.

4.8.1 The Test Runner

One of the key aims for the reworking of the template engine comparisons was to ensure complete isolation between the different template engines. The original performance comparison described in Section 4.3 had loaded the code for all the template engines into a single application and run a complete suite of tests for all loaded template engines in a single run of the code. The redesigned comparison framework not only separated out the template engine code into dynamically loadable plugins but also separated the evaluation of individual scenarios into separate test runs. This was to make sure that there were no lingering side-effects from one scenario affecting the next. For compatibility with the original set of measurements, the same selection of scenarios were used. The new test runner is illustrated with a UML sequence diagram [Figure 4.8.1], which may be contrasted with the similar diagram from the feasibility study [Figure 4.3.2].

When analysing the results of the original feasibility study it became clear that

[35] `https://junit.org/junit5/`

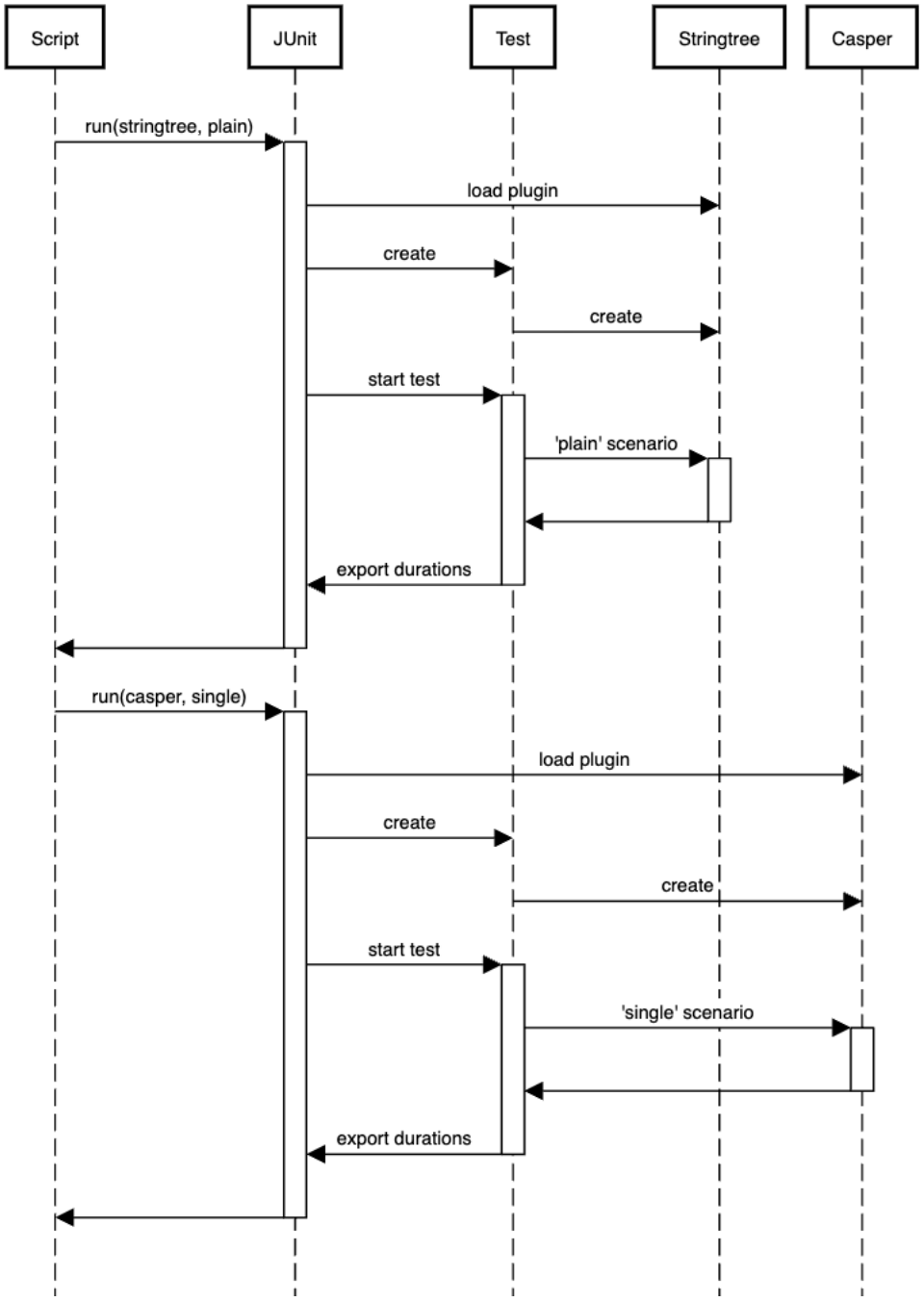


Figure 4.8.1: Sequence diagram for the improved performance test process

the experimental design was limited in several aspects. The plugin model enabled a way to evaluate individual template engines without interference from any of the others. The original experiments were also hard-coded to a specific number of expansions of each template. While implementing and testing the plugins it appeared that template engines have different performance characteristics as well as different rendering speeds for individual templates. To determine how these characteristics affect overall performance, a further series of experiments were

designed that would measure the time taken to render different quantities of each template.

The test runner main routine reads command-line parameters to obtain the parameters for the test run, then loads and initialises the specified template engine, loads the details of a specific scenario and the expected outcome, populates a context with the specified values, and then calls the template engine to expand the provided template the specified number of times. Results were collected for each triplet of (template engine, scenario, number of expansions) and are analysed in Section 4.10.

An extra command-line option is also supported.`-v` (for "verbose") enables extra diagnostic messaging for use when tracking down problems when evaluating a template engine. The verbose option was only used while setting up the comparisons. All timed runs had "verbose" disabled.

In Java, the entry point to an application is a `main` method, and the `main` method for the `Run` class, which executes a particular test scenario is shown in Appendix B.4.

Each test scenario is stored in a folder and consists of two files. One file is the output document that is expected to be produced when the scenario is run. The contents of this file will be checked against the output that is actually produced by each template engine. The other file is a "properties" file[36] that contains the specification of the scenario. A properties file consists of a set of key/value pairs. Each pair is positioned on a separate line of the file and takes the form *key*=*value* or *key*: *value*. The properties file used to specify a test scenario has one mandatory entry with the key `~template`. The value associated with this key is the name of the template to be expanded. When the scenario is run, this template name will be used to locate the template within a folder of templates specific to the template engine being evaluated. The properties file does not require other entries, but any other entries in the properties file will be used to populate the template context before expanding the template.

In a basic properties file, all keys and values are text strings. For some of the scenarios, however, the template context needs to be populated with values that are not simple strings. To enable this, all the context values in the properties files are processed using a filter that detects and converts values to different types. Each property key is examined for the presence of certain indicator characters at the end. If no indicator characters are present, then the key is used as it is and the value is treated as a text string. The presence of any of the indicator characters

[36] `https://docs.oracle.com/cd/E23095_01/Platform.93/ATGProgGuide/html/s0204propertiesfileformat01.html`

causes the value to be converted to the associated type. After conversion any indicator characters are removed from the key before adding the value to the template context. As an example, the properties file in Appendix 4.21 will result in the use of a template named "cond" with a template context containing a single entry named "yes" with a value of boolean *false*.

```
~template: cond
yes?: false
```

Listing 4.21: Example 'properties' file

The initial set of indicator characters are listed in Appendix 4.5

Character	Resulting Type	Value
`?`	`java.lang.Boolean`	*true* if it starts with `T` or `t`
`!`	a new object of a specified type	fully qualified class name
`[`	`java.lang.Array`	comma-separated list of strings
`>`	a reference to another value	the key of the other value

Table 4.5: Indicator characters for key/value conversion

4.8.2 Experimental Process

The aim of this experiment was to address the problems with the original study, as described in Section 4.6.1. Problems 1 (all template engines in memory at one time) and 2 (class and package name clashes) were addressed by the introduction of the template engine driver model and the dynamic loading of a single driver for each test run. Problem 3 (hard-coded configurations) was addressed by the use of enhanced properties files for scenario specification. Problem 4 (A timing anomaly with the test order) is no longer relevant, because in the re-coded architecture each test is run in isolation.

Problems 5 (the use of a fixed number of template expansions) and 6 (non-machine-readable results output) remained to be addressed by the design of the experimental process.

To address problem 5, the number of template expansions was added as a run-time parameter to an experiment run. This allowed test runs to be controlled by a script that could measure the time to run a specific scenario using a specific template engine for a range of different quantities of expansions. Each scenario for each template engine was initially tested by hand with a small range of quantities to ensure that the test runs and the time measurement worked correctly and

produced repeatable results. Once all the scenarios for each template engine were considered fit for further experiments, a script was developed to run each combination for a broader range of repetitions.

To address problem 6, the output data format was changed. The changes are discussed in Section 4.9.1.

To enable evaluation of the performance of everything from a single expansion of a single scenario using a single template engine to a full set of tests for every scenario for every template engine at a range of numbers of repetitions, a series of scripts were developed. The more complex and long-running scripts were built to use the more specific and quicker tests. This also ensured that there was no accidental differences in the way experiments were performed between single specific runs and a full set. Where possible, scripts were coded to accept optional arguments, but apply reasonable default values if the arguments are not supplied.

The basic script, named `run.sh` is responsible for running a single scenario using a single template engine a specified number of times. All the arguments are optional, but the template engine name and the scenario name will usually need to be supplied in practice. If not supplied, the template engine defaults to the "dummy" template engine used when compiling the experiment framework (see Section 4.7.1) and the scenario defaults to the "plain" scenario that contains only boilerplate text and no template language features. Calling this script with these defaults can be used to ensure that the script is working correctly, but is of little use for performance measurement. If no value is supplied for the number of repetitions, then the specified scenario will be evaluated a single time. This default value was frequently used when investigating issues and potential solutions to template engine or template language problems (see Section 4.10.4).

The Java language was not initially designed as a scripting languages that can be easily run from a command line. Java is a compiled language, which means that a compilation step is required before the code can be run. Java files are generally compiled using the `javac` tool, although some development environments skip the tool in favour of calling the lower-level APIs used by that tool. The result of this compilation step is one or more "class files" (files whose names have a `.class` suffix), which can then be executed by the Java Virtual Machine (JVM). Java class files are executed using the `java` tool. When the result of the compilation step is a single class file that uses no external libraries or other components, executing it can be as simple as something like `java HelloWorld.class`. However, Java projects that result in a single independent class file are relatively rare.

In the case of evaluating the performance of template engines, to execute a

performance test requires access not only to the class files that comprise the comparison framework, but also to the class files that form the template engine being evaluated. To include multiple class files when executing some compiled Java code, the JVM provides the notion of a *classpath*. A classpath is formed from a list of class files, directories, and "jar" (java archive) files. When the code is executed, all the class files specified in the classpath, and all the class files in the specified folders, and all the class files contained in the specified jar files are available for the code to use.

The `run.sh` script constructs a classpath from the following:

- The classes that comprise the specified template engine driver, compiled into a directory named "bin" (for binary) in a directory named for the template engine.
- The classes and libraries that comprise the specified template engine itself, placed into a directory named "lib" (for libraries) in a directory named for the template engine.
- The general-purpose classes used by the comparison framework and the template engine drivers, placed into a directory named "shared/bin"
- The classes that comprise the comparison framework itself, placed into a directory named "bin"

Using the classpath constructed as described above, the `run.sh` script calls the main method of the entry point of the comparison framework, the `runner.Run` class. The name of the template engine, the name of the scenario, and the number of repetitions, as well as any extra command-line arguments from the script are passed as arguments to the Java code, which then executes the evaluation. Additional command-line arguments are always optional but include, for example, a `-v` (for "verbose") option to enable extra diagnostic output during evaluations.

All the files and directories that are constructed into the classpath are relative to the current working directory. For this script to function correctly, it must be started from the base directory of this project. If it is run from elsewhere, the required classes and data files will not be available, and the script will not be able to run.

The code for the `run.sh` script is given in Appendix B.4.1.

Building on the basic `run.sh` script, a second script, `one.sh` was developed to run the full suite of scenarios using a specified template engine. In this script there

is no scenario parameter needed, as it implicitly processes all scenarios. As with the `run.sh` script, the template engine name is optional, defaulting to "dummy", and the number of repetitions is optional, defaulting to 1.

The list of scenarios to evaluate is not hard-coded in the script, but derived by listing the files in a "scenarios" directory relative to the current directory. This script scans the available scenarios, and calls the `run.sh` script with the name of each scenario as well as the specified template engine name and number of repetitions. Just as with `run.sh`, this script must be run from the base directory of this project so that it can find all the files it needs.

The code for the `one.sh` script is given in Appendix B.4.2.

The next step beyond the `one.sh` script is a script (`all.sh`) that runs all the available scenarios for all the available template engines. As with the `one.sh` script, there is no need to specify the scenario name or the template engine name, as this script scans directories to find both the list of scenarios and the list of template engines to evaluate. The number of repetitions is still optional and defaults to 1, as in the previous two scripts.

This script, while broadly similar in structure to the `one.sh` script, contains some extra processing to determine which directories contain a valid template engine driver. The main technique used for this is to search each directory for a file named `EngineFactory.java`. This file is the entry point for a dynamically loaded template engine driver, as discussed in Section 4.7.1. Any directory that does not contain this file cannot contain a template engine driver. In addition to looking for the template engine driver entry point, the `all.sh` script also applies a further rule. Any matching directory that also contains a file named "SKIP" denotes an inactive template engine driver that should not be evaluated. This feature was initially implemented to exclude the "dummy" template engine driver from the evaluation process, but was later found to be useful when comparing a reduced set of template engines for other purposes.

Once the list of valid and active template engine driver directories has been collected, each one in turn is passed to the `one.sh` script to evaluate that engine in the full suite of scenarios for the specified number of repetitions.

The code for the `all.sh` script is given in Appendix B.4.3.

4.9 Measurement Sets

The performance comparison of the cohort of template engines were performed as a series of measurement sets. Each measurement set provided a different view on the performance of the selected template engines. Feedback from each measurement set was used to suggest improvements or alternative approaches for further sets.

Each measurement set was performed by running a `fulldata` script. Each `fulldata` script made use of the `all.sh` script described in Section 4.8.2 with a different pattern of repetitions to collect a set of measurement data.

The measurement sets performed during this experimentation are as follows:

Set 1 examined the performance of a single template engine (*Solomon*) at a range of loads in detail.

The Set 1 version of the `fulldata` script (`fulldata1.sh`) used a simple linear shell "for" loop to start with one repetition and increase one by one until the feasibility study maximum of 10,000. It became apparent that for some of the template engines this process would take a prohibitively long time, so this approach was abandoned.

The code for the Set 1 `fulldata1.sh` script is given in Appendix B.4.4 and the results of Set 1 are explored in Section 4.10.1.

Set 2 reduced the level of detail in order to compare all of the selected template engines against all the scenarios at a range of loads.

To gain a quick insight into the major differences between the template engines, the Set 1 script was copied to `fulldata2.sh` and re-coded to instead perform a simple logarithmic experiment, executing each combination of scenario and template engine 1, 10, 100, 1000, and 10000 times. While the logarithmic script was successful in providing an overview of the relative performance characteristics of the different template engines, the results contained obvious artefacts related to the large jumps in the number of repetitions, and the attempts to interpolate between them.

The code for the Set 2 `fulldata2.sh` script is given in Appendix B.4.5 and the results of Set 2 are explored in Section 4.10.2.

Set 3 introduced the *Handlebars* template engine and excluded *Hapax*, which overshadowed the other results. Set 3 also introduced averaging of multiple runs to decrease the impact of "noise" on the measurements.

A further version of the `fulldata` script was then written to gain more information about the performance of the template engines between the logarithmic steps. This approach attempted to improve the performance of the linear approach by using two slopes. To provide detail in the low end of the performance graph, this script increased the number of repetitions in increments of 10, rather than 1 for tests up to 100 repetitions. After this, the number of repetitions was stepped in increments of 100 up to the maximum of 10,000 repetitions. This took a long time, particularly for the slowest of the template engines, but the resulting data contained much more detail than the logarithmic script.

The code for the Set 3 `fulldata3.sh` script is given in Appendix B.4.6 and the results of Set 3 are explored in Section 4.10.3.

Set 4 re-introduced *Hapax* after a change to the *Hapax* driver to reduce internal warning messages and thereby improve performance. A problem was also rectified in some of the *Handlebars* templates that had been causing slow behaviour in the "iter" scenario. The inability of the *Stringtemplate* template engine to correctly include other templates was also addressed with a driver change.

This final version of the test script brought the test repetitions into a hybrid approach combining aspects of both the linear script and the logarithmic script. This approach increased the number of repetitions in steps of 10 up to 100, then steps of 100 up to 1000, then steps of 1000 up to 10000. While not exhaustive, this script was considered to be a reasonable compromise of providing enough information to characterise template engine performance behaviour in a time that allowed for multiple runs of each set for subsequent averaging.

The code for the Set 4 `fulldata4.sh` script is given in Appendix B.4.7 and the results of Set 4 are explored in Section 4.10.4.

4.9.1 Data Collection

To address problem 6 from the original measurements, The Java test runner was coded so that each execution of the test runner application resulted in a single CSV row of data. To gain a bigger picture of the relative performance characteristics of the different template engines required multiple runs of the test runner, so the experimental scripts were coded to append the data from each of the multiple

runs with different parameters into a single CSV output file for analysis. The CSV output included columns for the date and time of the test run, the template engine name, the scenario name, the number of repetitions, the time taken (in milliseconds) and an indication of whether the result of expanding the template produced correct output (indicated by **OK**) or incorrect output (indicated by **NOTMATCHED**). An example section of the CSV data might look as shown in Listing 4.22.

```
2022-06-30T11:57:32,stringtree,iter,10,24,NOTMATCHED
2022-06-30T11:57:32,stringtree,plain,10,18,OK
2022-06-30T11:57:33,stringtree,separate,10,26,OK
2022-06-30T11:57:33,stringtree,single,10,19,OK
2022-06-30T11:57:33,jte,bean,10,1090,OK
2022-06-30T11:57:34,jte,cond-false,10,1043,OK
2022-06-30T11:57:36,jte,cond-true,10,1038,OK
2022-06-30T11:57:37,jte,include,10,1090,OK
```

Listing 4.22: Example of CSV output

One advantage of this format is that the content of the generated CSV results files are independent of the order of the entries. Each row contains enough information to uniquely identify it and its contribution to the results. This became especially important during analysis when it was desired to look at the average of several runs through the complete test suite. In principle, merging the data could be done by simply concatenating the data from the output files into a new single file that could then be processed by the analysis and graphing software. In practice, each CSV file starts with a header line, so simply concatenating the files would result in several such header lines interspersed with the data. As the header lines contain textual descriptions of the columns in the CSV data, they do not contain valid test data and would prevent correct functioning of any analytic or graphing software. A short script was coded in Python to accept an arbitrary list of these CSV output files and correctly concatenate the data, keeping only a single header line at the start.

In several of the sets of testing, the `fulldata` script was run more than once in an attempt to detect the effects of events and forces outside the behaviour of the template engines being evaluated. The measurements were performed on a Linux virtual machine running on a Windows PC, so there were two levels of underlying system, either of which could spontaneously use system resources on tasks unrelated to the template engine performance experiments. While, as can be seen in the results in Section 4.10, the overall character of the performance of each template engine remained broadly the same, there was evidence of such external factors.

In an attempt to mitigate the effects of external interfering factors on the

measurements, an additional script was also coded in python to read the output from multiple `fulldata` runs and produce a new output file containing the arithmetical mean of the time taken for each set of similar measurements from the provided data. The results of this attempt at smoothing the data can also be seen in Section 4.10.

It is important to note that while care was taken to minimise changes in external factors such as hardware and software versions and other applications and processes running on the test platform during multiple runs for the same set, this could not practically be maintained between sets. Each set of testing therefore represents a comparison between the performance characteristics of the cohort of template engines but not an absolute measurement. The aim was that the overall relative characteristics would be repeatable if performed on other hardware and software systems, while the individual measurements would probably be different.

4.10 Results

As discussed in Section 4.9, the results from the performance comparison of template engines are gathered into a series of measurement sets. Each measurement set provides a different view on the performance of the selected template engines. Feedback from each measurement set is used to suggest improvements or alternative approaches for further sets. The details of each measurement set are described in the sections below, and overall discussion and conclusions are explored in Section 4.11 and Section 4.12. Where appropriate, results are illustrated by both large and small scale graphs. The small scale graphs fit a full set of 8 scenarios on a single page and provide an overview for direct visual comparison between the different scenarios. The large scale graphs provide clearer indication of the performance of individual template engines in the context of a single scenario.

4.10.1 Set 1

As mentioned in Section 4.8.2, the first set of performance comparisons was intended to measure the performance of each template engine running each scenario at every step of repetitions from 1 to 10,000. This initially seemed plausible, with some of the faster template engines being able to complete their tests in just a few hours on the available computing hardware. A run using a slightly modified `fulldata1.sh` script (see Appendix B.4.8) that called the

`one.sh` script using only the *Solomon* template engine, one of the fastest from the original study, completed in 306 minutes (over 5 hours). However, some of the slower template engines took tens or even hundreds of times longer to process the test scenarios in the original study, and would therefore possibly require many days to complete.

Examination of the results of running the modified `fulldata1.sh` script (shown in Figure 4.10.1) showed no evidence of unusual performance characteristics at specific numbers of repetitions. What was clear, however, was the presence of "noise" in the results. The working hypothesis was that this was due to other software and hardware factors outside the software being tested. To eliminate such "noise" would require one of two techniques:

- Curve fitting or smoothing could assist in providing a cleaner curve for the data, but would not necessarily be representative, as it would include the "noise" in its calculations.
- Averaging multiple runs would potentially provide a better solution, because any variation in results which is present in only a single data set would contribute proportionally less to the final result.

Averaging multiple runs was chosen as the more appropriate technique, and was used for measurement sets 3 (Section 4.10.3) and 4 (Section 4.10.4).

The combination of the apparent lack of need for this degree of detail, and the need to perform multiple runs for averaging, led to this approach being abandoned in favour of taking larger steps to achieve more results faster in further sets.

4.10.2 Set 2

The second set of measurements used a logarithmic progression of repetitions, measuring the time taken by each template engine to process each scenario 1, 10, 100, 1000, and 10000 times. Each run of this script completed much more quickly than the version in Set 1, which enabled a full set of all scenarios for all active template engines. The initial results are shown in Figure 4.10.2 and Figure 4.10.3.

Although the graphs in Figure 4.10.2 and Figure 4.10.3 go some way to indicating the relative performance characteristics of the different template engines, they are not very useful. The first six graphs in Figure 4.10.2 have been rendered using a completely different y-axis range (0-1300ms) compared to the remaining two graphs in Figure 4.10.3 (0-20000ms) which makes it difficult to get a visual

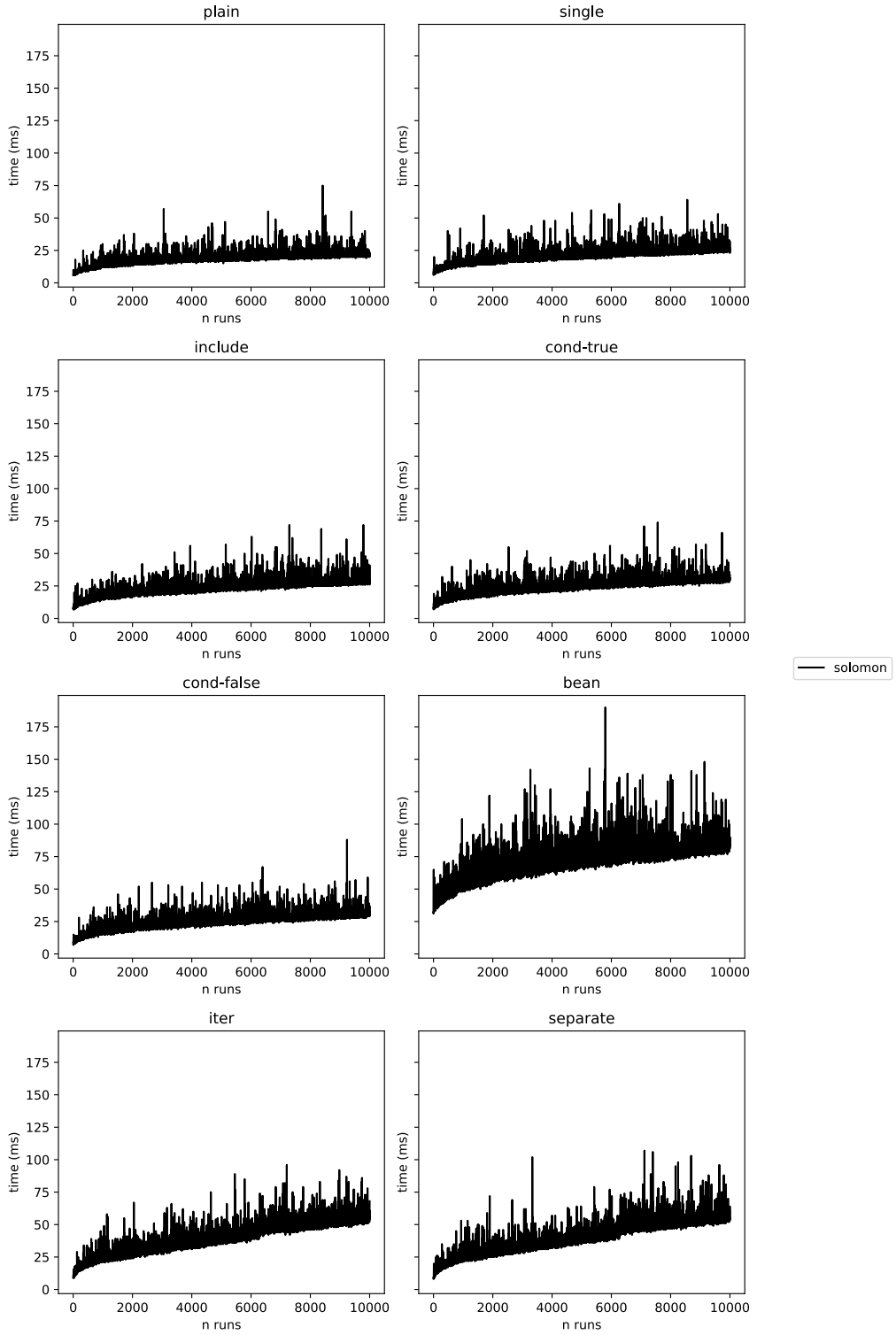


Figure 4.10.1: Performance comparison set 1 overview illustrating the results of testing at every increment and the impact of noise on the readings (*Solomon* only)

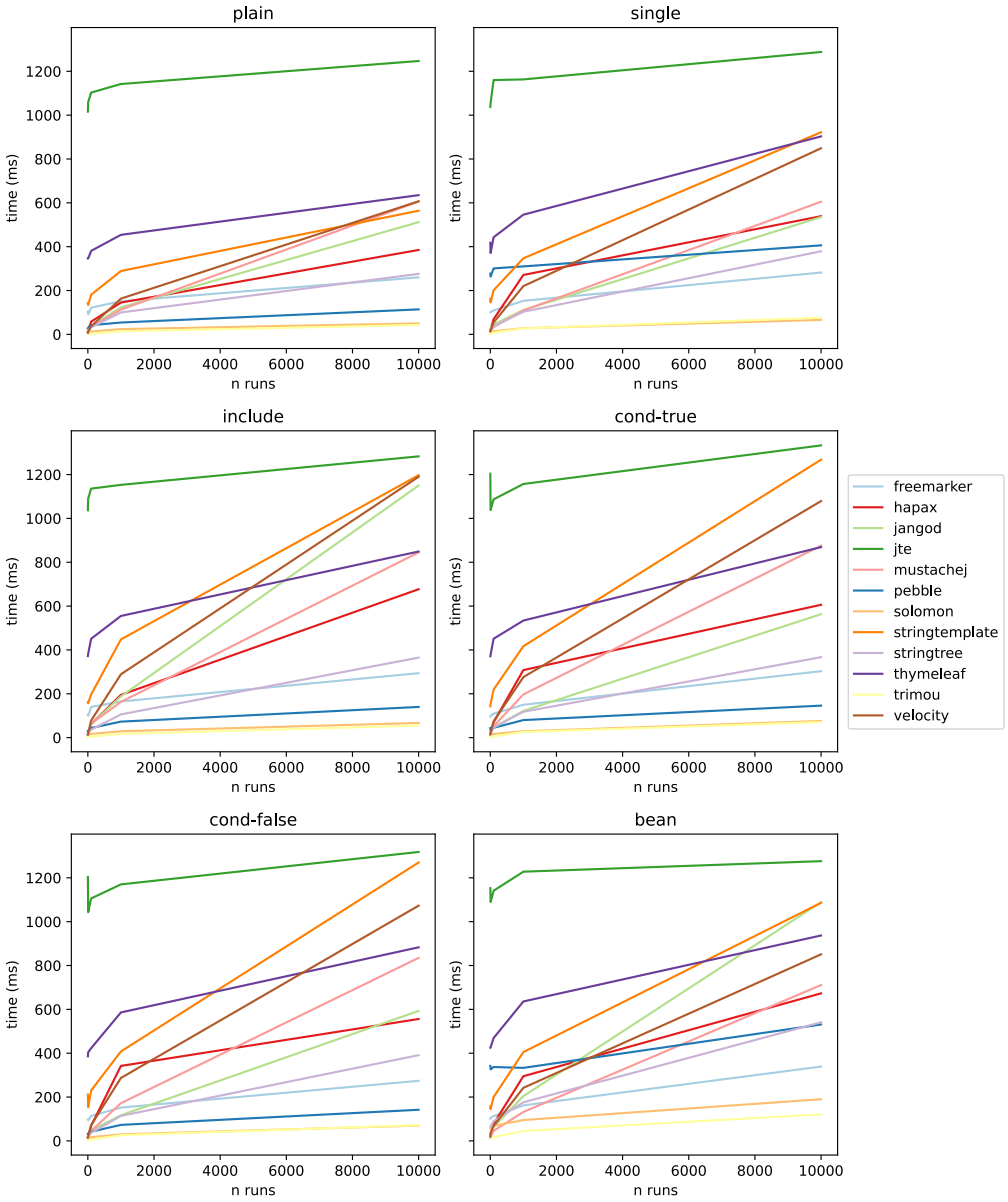


Figure 4.10.2: Performance comparison set 2 overview for the six scenarios (*plain*, *single*, *include*, *cond-true*, *cond-false* and *bean*) in which the performance of *Hapax* was comparable to the other template engines

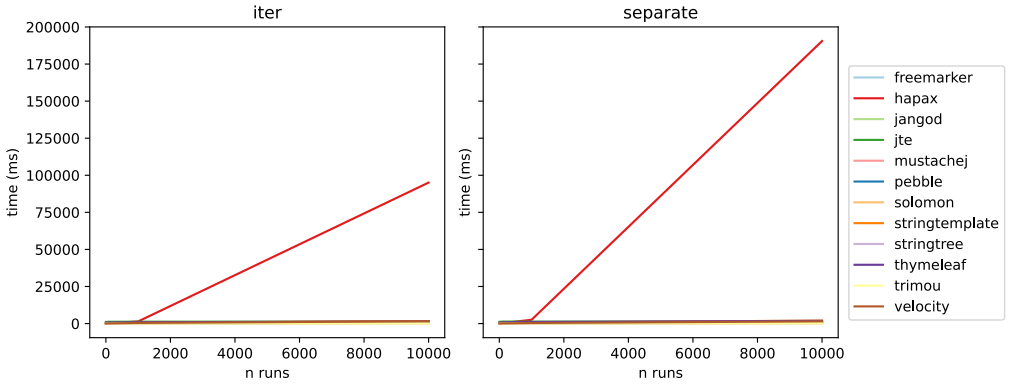


Figure 4.10.3: Performance comparison set 2 overview showing the very poor performance of *Hapax* in the *iter* and *separate* scenarios and the way it overshadows other template engines

understanding of the relative performance of the template engines in the full range of scenarios. In Figure 4.10.3 The time taken by all the other template engines is completely overshadowed by the much worse performance of the loop scenarios (*iter* and *separate*) when evaluated using the *Hapax* template engine.

Considering the full cohort of template engines on a single graph can make it difficult to clearly see differences between individual template engines. The same results from Figure 4.10.2 and Figure 4.10.3, but with the *Hapax* template engine omitted from the graphs, are shown in more detail in Figure 4.10.4 to Figure 4.10.11.

The graphs in Figure 4.10.4 to Figure 4.10.11 illustrate much more clearly the range of differences between the various approaches used by the different template engines. On the whole, all the template engines show a progressively increasing duration as the number of repetitions is increased, but they differ in three key aspects: the minimum duration, the slope of the increase, and the variation between scenarios. The minimum duration is indicative of some kind of "setup cost" incurred by the template engine before it is able to expand a template. This contributes a larger proportion at low numbers of repetitions, and can make a template engine with a high minimum duration seem as much as a thousand times worse than other template engines for low-volume use. The slope of the increase is indicative of the additional time take to process each repetition of the same template. This factor is most important at high numbers of repetitions. A template engine with a steep slope can soon require much more time to process the same templates as a template engine with a high minimum duration but a shallow slope of increase. These factors are not typically constant for each template engine, however, but vary depending on the content of each template being expanded. The graphs for the different scenarios show considerable differences in slope and to a lesser degree, minimum duration between scenarios.

In the data from Set 2, *JTE* has a high minimum duration of around 1000ms but a relatively shallow slope of increase. This implies that in many cases, potentially at very high volumes of template expansions, it will eventually take less time to process all the templates than other template engines. However, this is not always the case. There are some template engines, such as *Trimou* and *Solomon*, which have both a low minimum duration *and* a shallow slope.

The performance results for each scenario are discussed below in a series of figures and tables that highlight the differences in low-volume (single run) and higher-volume (10,000 run) performance between each of the template engines for that scenario.

"plain" (no placeholders, just boilerplate text) The results of the "plain" scenario are shown in Figure 4.10.4 and Table 4.6.

In the "plain" scenario, both these template engines have very low minimum duration and, even at 10,000 repetitions, have only increased by 41ms (*Trimou*) and 37ms (*Solomon*) whereas *JTE* has increased by 231ms (see Table 4.6). Extrapolating from these results, it seems unlikely that either of these two template engines would ever perform worse than *JTE* in this scenario, regardless of the quantity of template expansions.

In this study, the "plain" scenario largely serves to measure the underlying performance of the template engine including "setup costs" and the time to process a document. Without the presence of any placeholders, it does not provide information about the performance characteristics of each template engine when processing different placeholders and directives. The remaining scenarios are investigated below.

"single" (a single replacement placeholder) The results of the "single" scenario are shown in Figure 4.10.5 and Table 4.7.

The performance curves for each template engine in this scenario are broadly similar to the ones for the "plain" scenario, although most template engines exhibit a steeper slope of increase, indicating that they are doing more work for each template expansion. The main exception to this is *JTE*, which has an almost identical curve to the "plain" scenario.

Although *JTE* still took more time than any of the other template engines, even at 10,000 repetitions, some of the ones with steeper slopes (such as *Stringtemplate*, *Velocity*, and *Thymeleaf*) were approaching the *JTE* minimum duration.

The performance profile for the *Pebble* template engine warranted further investigation. In this scenario, as well as in the "bean", "iter" , and "separate" scenarios, the *Pebble* template engine exhibits a relatively large minimum duration compared to the "plain", "include", "cond-true" and "cond-false" scenarios. The code for *Pebble*[37] makes use of lazy evaluation, a technique in which the calculation or processing of data values is deferred until needed. It is significant that the "single", "bean", "iter", and "separate" scenarios all require context values to be rendered. It appears that the apparent speed of *Pebble* in scenarios without value rendering is a benefit of the lazy evaluation process. In practice, templates that do not include rendered values are very rare, so the relative speed of *Pebble* in real use is likely to be closer to the performance in

[37] `https://github.com/PebbleTemplates/pebble`

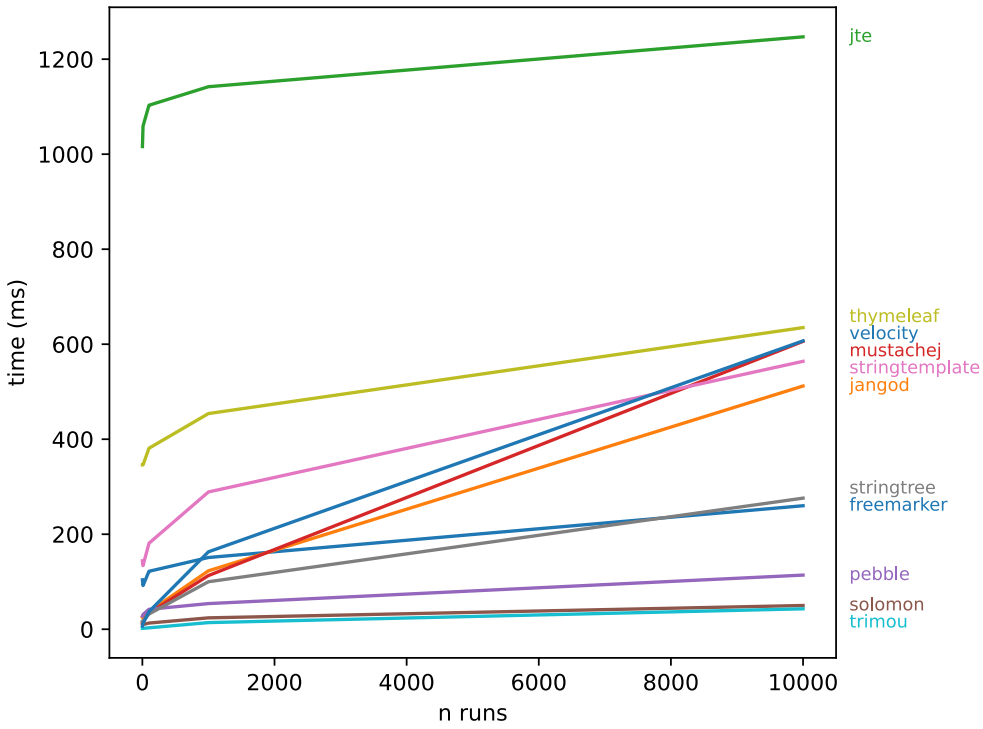


Figure 4.10.4: Set 2 performance comparison for the *plain* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	2	43
solomon	13	50
freemarker	104	260
stringtemplate	144	564
stringtree	17	276
jte	1016	1247
jangod	25	512
velocity	7	607
pebble	28	114
thymeleaf	346	635
mustachej	12	606

Table 4.6: Set 2 durations (ms) for the “plain” scenario

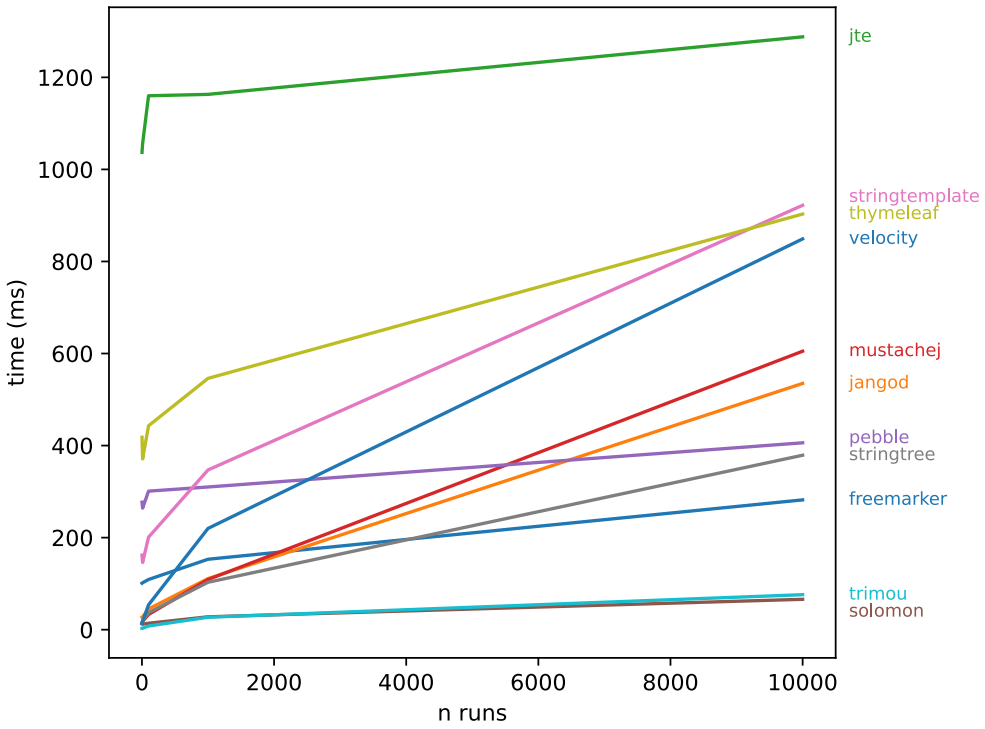


Figure 4.10.5: Set 2 performance comparison for the *single* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	3	76
solomon	13	66
freemarker	101	262
stringtemplate	162	922
stringtree	16	378
jte	1037	1288
jangod	27	535
velocity	14	849
pebble	277	406
thymeleaf	418	903
mustachej	13	605

Table 4.7: Set 2 durations (ms) for the “single” scenario

the "single" scenario.

"include" (include a second template) The results of the "include" scenario are shown in Figure 4.10.6 and Table 4.8.

This scenario followed the progression from the previous two. *JTE* was still the worst performing template engine over the range of 1 to 10,000 repetitions, but several of the other template engines (*Stringtemplate*, *Jangod*, *velocity*) were in the same area by the end of the range. *Trimou* and *Solomon* remain the best performing template engines in this scenario, both once again showing a shallower slope than *JTE*, so likely to remain a more performant choice regardless of template volume.

The poor performance of the *Stringtemplate* and *Jangod* template engines in this scenario may be partly explained by their lack of full support for this operation. It appears that the ability to include templates is not included in *Jangod* by design, so a failure in this scenario is to be expected. The documentation for *Stringtemplate* claims that template inclusion is possible, however, but at this point in the experimentation, template inclusion in *Stringtemplate* was not working. Template inclusion in *Stringtemplate* was addressed in Set 4 of the measurements (see Section 4.10.4).

"cond-true" and "cond-false" The results of the "cond-true" scenario are shown in Figure 4.10.7 and Table 4.9. The results of the "cond-false" scenario are shown in Figure 4.10.8 and Table 4.10.

These two scenarios produced very similar results. This is not surprising, as they use identical templates, so the processing needed to parse and/or compile the templates should be the same. By the end of the range there are some small differences in the time taken by each of the template engines, but based on the evidence of the presence of external "noise" shown in Figure 4.10.1, this could easily be the result of that. These differences were explored in more detail in further sets of measurements.

Once again, *JTE*, *Stringtemplate*, and *Velocity* performed the worst at high volumes, but *Jangod*, which was among the worst in the "include" scenario found itself in the middle of the pack for these scenarios.

"bean" (implicit method call) The results of the "bean" scenario are shown in Figure 4.10.9 and Table 4.11.

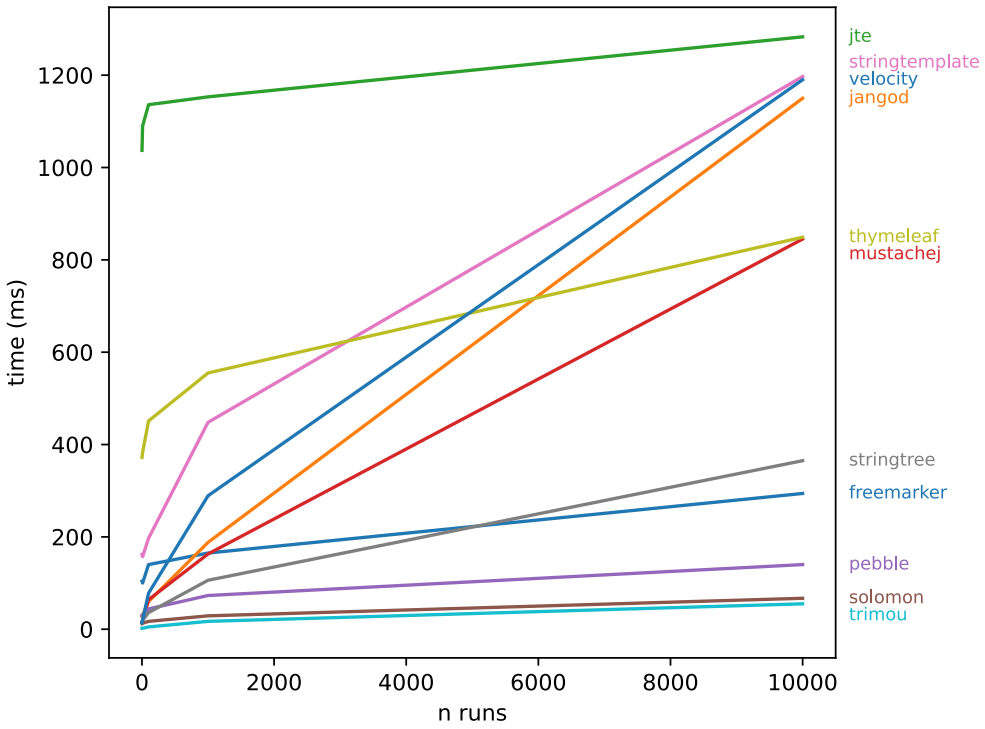


Figure 4.10.6: Set 2 performance comparison for the *include* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	2	55
solomon	12	67
freemarker	104	294
stringtemplate (NOTMATCHED)	162	1197
stringtree	16	365
jte	1037	1283
jangod (NOTMATCHED)	28	1150
velocity	14	1190
pebble	30	140
thymeleaf	372	849
mustachej	16	845

Table 4.8: Set 2 durations (ms) for the “include” scenario

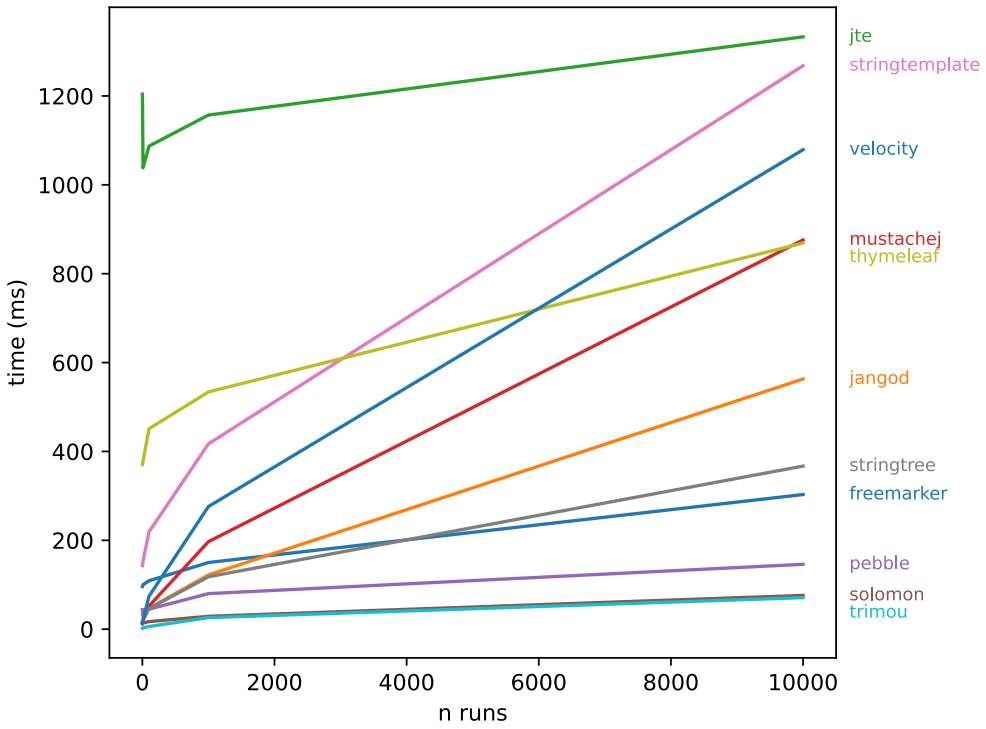


Figure 4.10.7: Set 2 performance comparison for the *cond-true* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	2	71
solomon	12	76
freemarker	96	303
stringtemplate	143	1268
stringtree	17	367
jte	1204	1333
jangod	32	563
velocity	15	1079
pebble	44	145
thymeleaf	371	869
mustachej	16	876

Table 4.9: Set 2 durations (ms) for the “cond-true” scenario

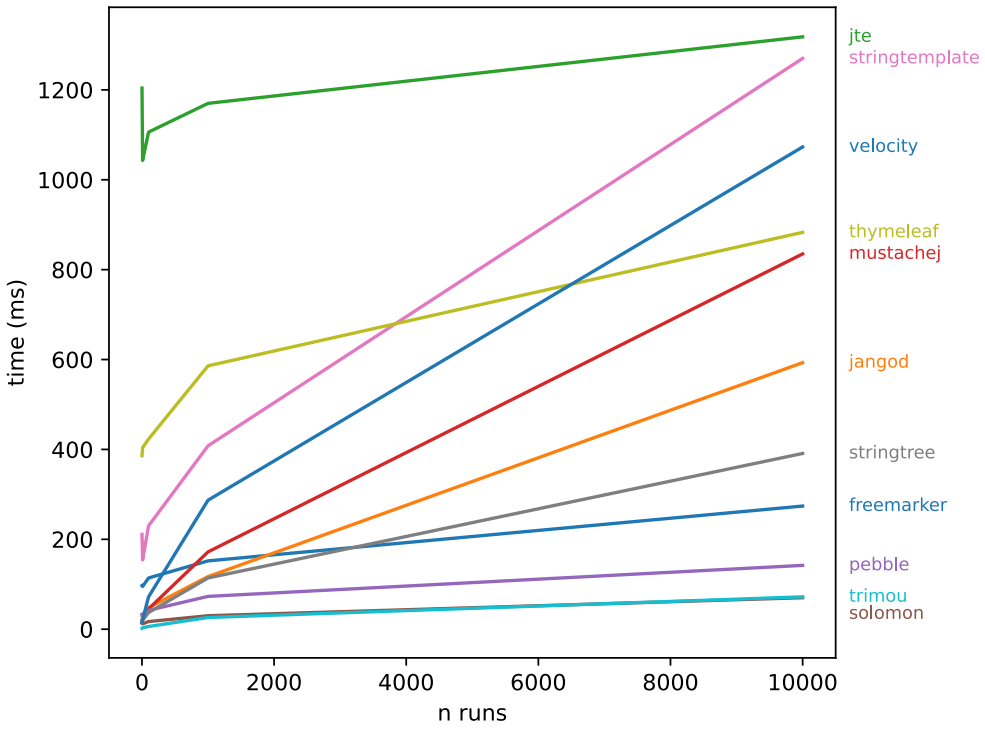


Figure 4.10.8: Set 2 performance comparison for the *cond-false* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	2	72
solomon	13	70
freemarker	97	274
stringtemplate	211	1270
stringtree	21	391
jte	1204	1318
jangod	30	593
velocity	15	1073
pebble	33	142
thymeleaf	386	883
mustachej	17	885

Table 4.10: Set 2 durations (ms) for the “cond-false” scenario

Most template engines in this cohort seem to have roughly equivalent performance characteristics for this scenario as for the "single" scenario, reflecting what should be a broadly similar parsing and compilation overhead for essentially a slightly specialised form of a value substitution placeholder. The usual poor-performers *Stringtemplate* and *Jangod* perform somewhat worse in this scenario than the "single" scenario. As mentioned in the discussion of the "include" scenario in Table 4.8, the poor performance of *Jangod* in this scenario may be due to the lack of support for this feature.

"iter" and "separate" (present all items in a collection) The results of the "iter" scenario are shown in Figure 4.10.10 and Table 4.12. The results of the "separate" scenario are shown in Figure 4.10.11 and Table 4.13.

The "iter" and "separate" scenarios are arguably the most complex of this suite of evaluation scenarios. Template languages represent these scenarios in very different ways, which leads to distinct differences in processing speeds. These scenarios are the only ones in the suite in which *JTE* is not the worst performer. In the "iter" scenario, *Stringtemplate*, *Velocity*, *Thymeleaf*, and *Mustachej* all take longer to process 10,000 repetitions than the pre-compiled *JTE*. The steep slopes exhibited by these template engines in this scenario may indicate that they are doing extra work, such as re-parsing the sub-template used to present each list item, for every element of the collection. In the "separate" scenario, both *Stringtemplate* and *Thymeleaf* show considerably worse performance than *JTE* even at relatively low volumes.

Most of the template engines were able to successfully process the "iter" scenario. Although *Solomon* and *Stringtree* are shown as "NOTMATCHED" in Table 4.12, this was discovered to be a template error, corrected in later sets. Only *Mustachej* was unable to successfully generate the required output. The "separate" scenario requires separating the elements of a collection with commas, but without an extra comma at the end of the items, and proved to be more challenging. *Jangod*, *Velocity*, and *Mustachej* were unable to successfully generate the required output.

Overall considerations Overall, the results from this set show differences between the performance profiles of the different template engines in this cohort when used for different scenarios and therefore go some way to validating the experimental approach. The distinct difference in performance characteristics between scenarios shows why a single "benchmark" that tests one template at a fixed number of repetitions or for a fixed duration (Hasselbring, 2021) does not tell the whole story.

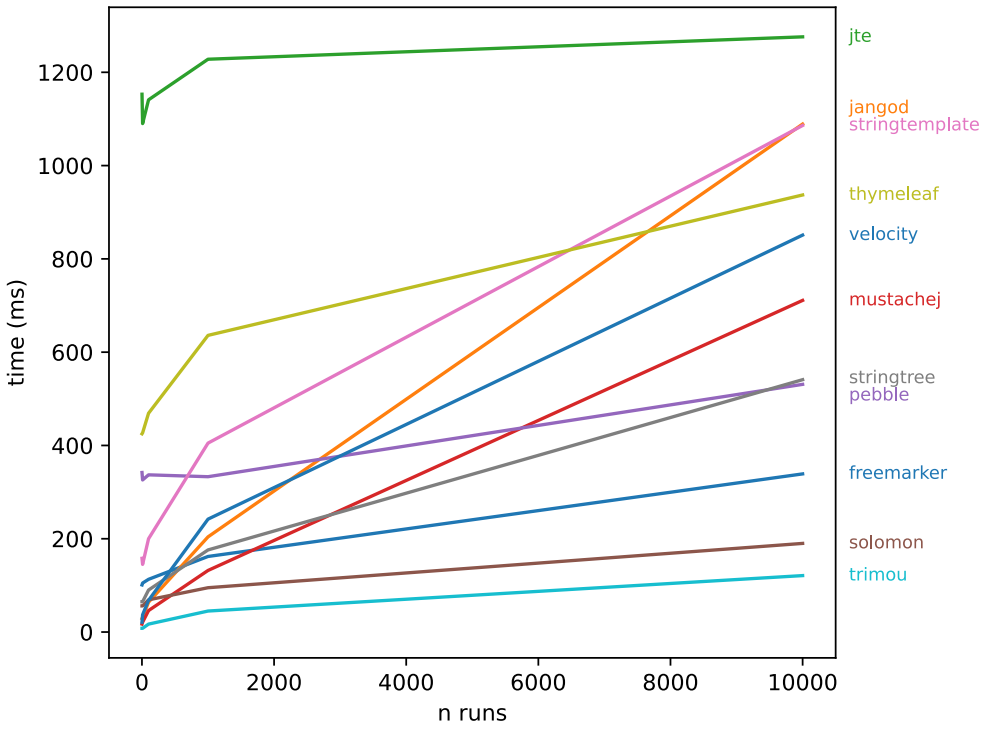


Figure 4.10.9: Set 2 performance comparison for the *bean* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	8	121
solomon	56	190
freemarker	101	339
stringtemplate	158	1086
stringtree	65	541
jte	1153	1276
jangod (NOTMATCHED)	29	1089
velocity	21	851
pebble	342	531
thymeleaf	425	937
mustachej	17	711

Table 4.11: Set 2 durations (ms) for the “bean” scenario

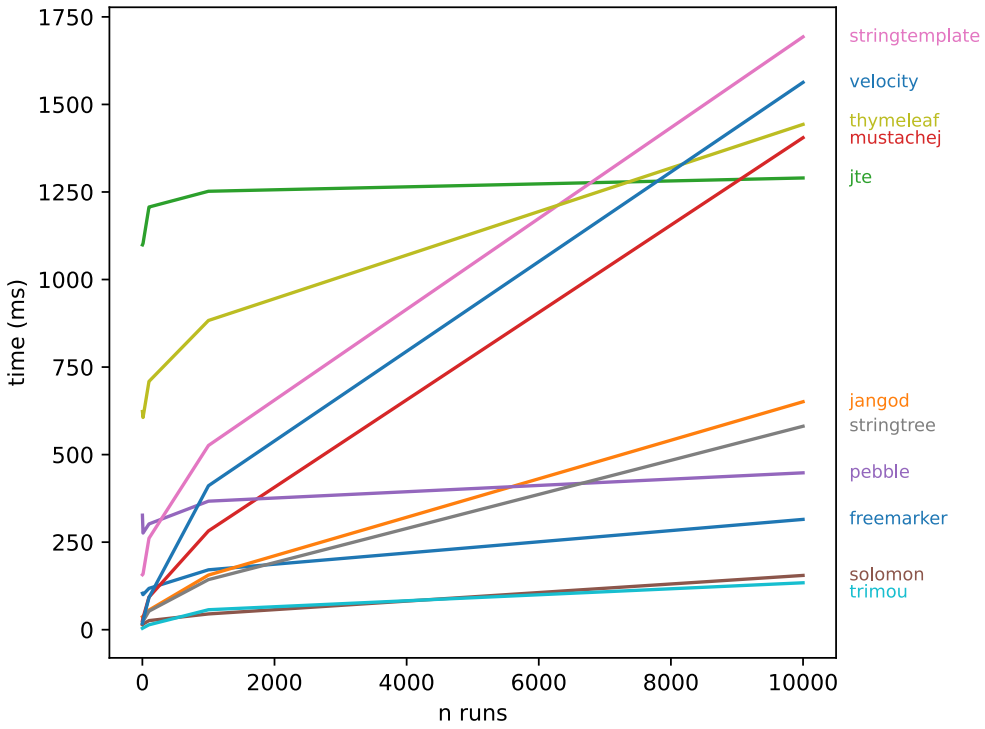


Figure 4.10.10: Set 2 performance comparison for the *iter* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	4	134
solomon (NOTMATCHED)	14	155
freemarker	104	315
stringtemplate	157	1693
stringtree (NOTMATCHED)	20	581
jte	1099	1290
jangod	37	651
velocity	18	1563
pebble	327	448
thymeleaf	623	1443
mustachej (NOTMATCHED)	26	1405

Table 4.12: Set 2 durations for the “iter” scenario

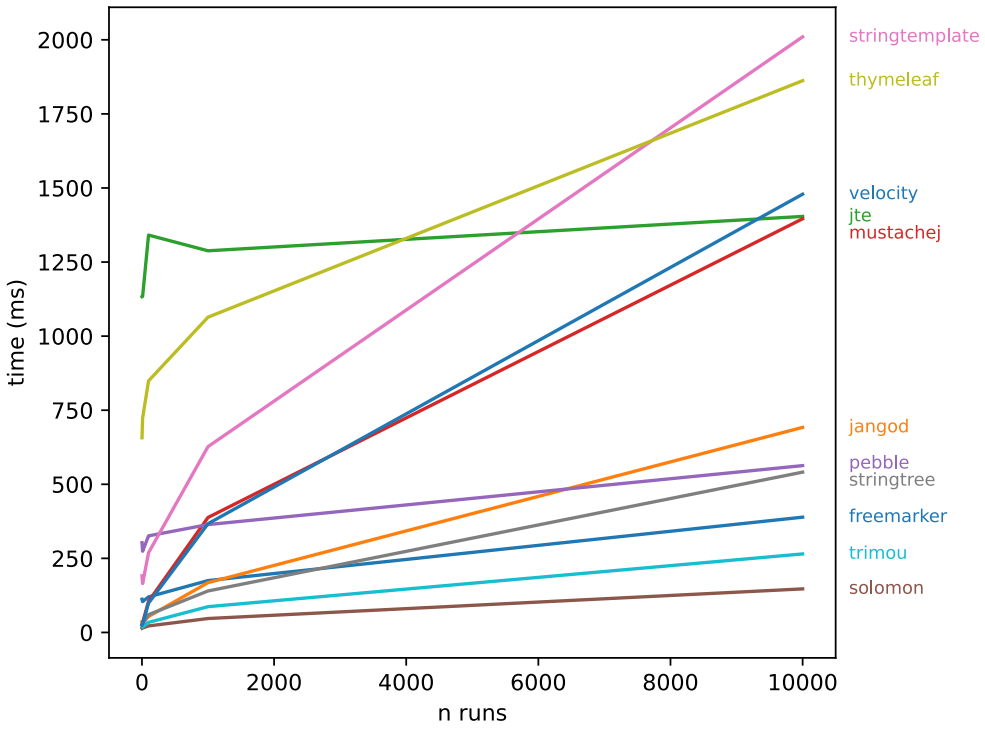


Figure 4.10.11: Set 2 performance comparison for the *separate* scenario, excluding *Hapax*

Engine	**Single Run**	**10000 Runs**
trimou	28	265
solomon	14	147
freemarker	112	389
stringtemplate	191	2010
stringtree	19	541
jte	1133	1404
jangod (NOTMATCHED)	36	692
velocity (NOTMATCHED)	25	1479
pebble	303	563
thymeleaf	657	1682
mustachej (NOTMATCHED)	28	1396

Table 4.13: Set 2 durations for the “separate” scenario

However, there are some anomalies in the results, such as the variable "setup cost" of the *Pebble* template engine and the spikes in duration taken by *JTE* in the "cond-true", "cond-false", and "separate" scenarios. It is unclear from these results whether these anomalies are inherent in the performance characteristics of the template engines concerned, or are due to external "noise" distorting the readings and corrupting the data.

Later sets of measurements addressed these issues with a combination of more measurements per set, and re-running each set multiple times in the hope of decreasing the impact of external "noise". As a reminder, the graphs in Figure 4.10.4 to Figure 4.10.11 do not include the results from *Hapax*. Reconsidered results following changes to the *Hapax* template engine driver are examined in Set 4 (see Section 4.10.4).

4.10.3 Set 3

As discussed in Section 4.8.2, the third set of performance evaluation measured the time taken to expand templates in a larger number of steps than the second set. This set also introduced the *Handlebars* template engine. The template language for *Handlebars* is similar to the other "Mustache" style template languages in this cohort (*Mustachej* and *Trimou*) but, as can be seen in the detailed graphs Figure 4.10.14 to Figure 4.10.21, they each have different performance characteristics. The results considered in this set also exclude the measurements of the *Hapax* template engine, as the time taken to process the "iter" and "separate" scenarios overshadowed the other results.

Including more measurements than Set 2 gave more data but, as can be seen in Figure 4.10.12, exhibited similar "noise" to the results of the first set. In an attempt to mitigate the effects of such external interference, the Set 3 script was run 8 times and the results averaged. The results of this process are shown in Figure 4.10.13 and explored further in the detailed graphs Figure 4.10.14 to Figure 4.10.21.

Comparing the graphs in Figure 4.10.14 to Figure 4.10.21 with similar graphs from previous sets shows an overall increase in performance of all the template engines, with *JTE* typically taking around 600ms, compared to around 1000ms in previous sets. This is due to an update to the underlying hardware, as discussed in Section 4.8.2. While the overall timings for each template engine may have changed, the shape of the curves is largely similar, which implies that the factors used to evaluate the template engines remain constant relative to each other. This acts to validate the overall repeatability of the experimental approach.

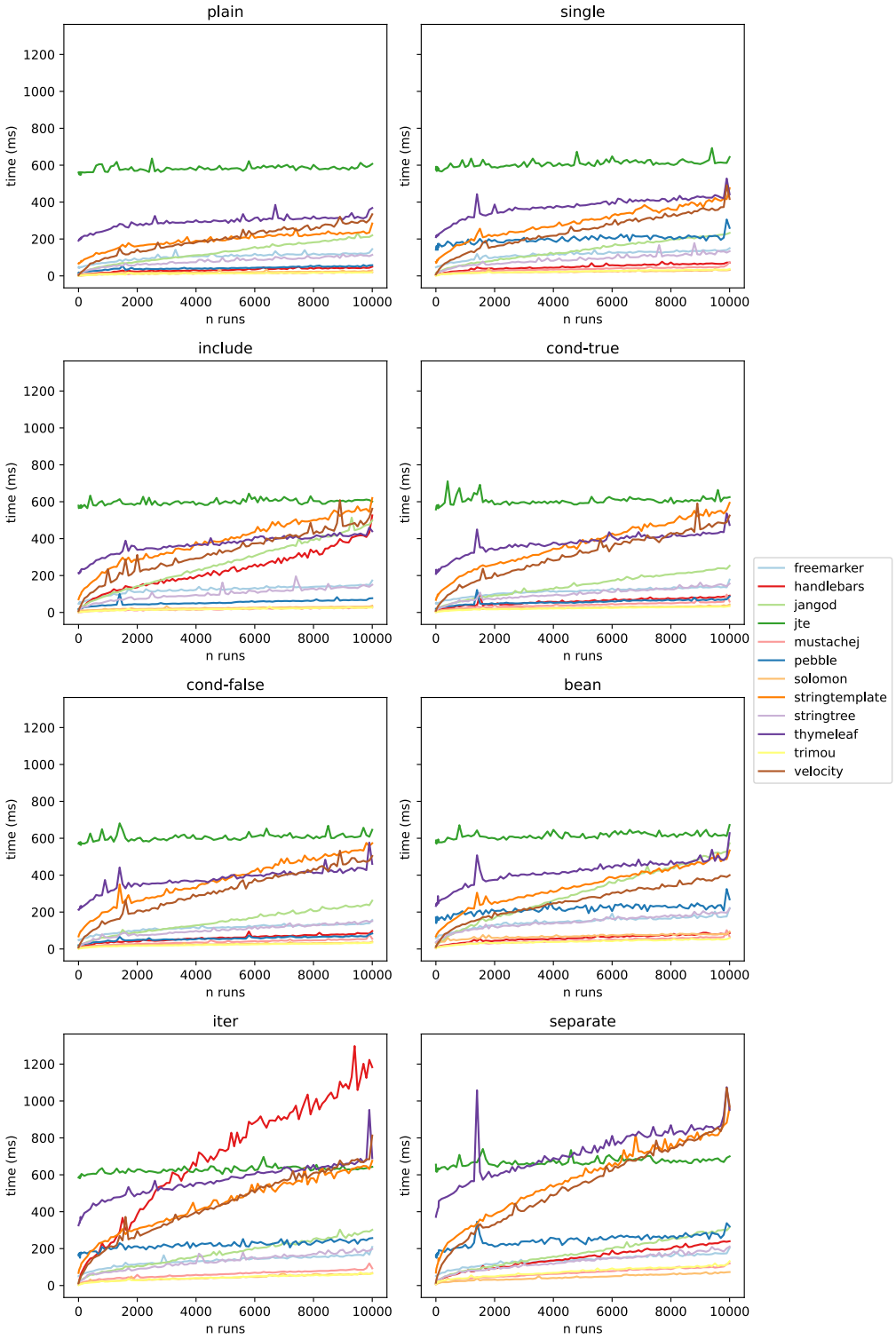


Figure 4.10.12: Performance comparison set 3 overview with an increased number of samples showing the impact of *noise* unrelated to the experiment

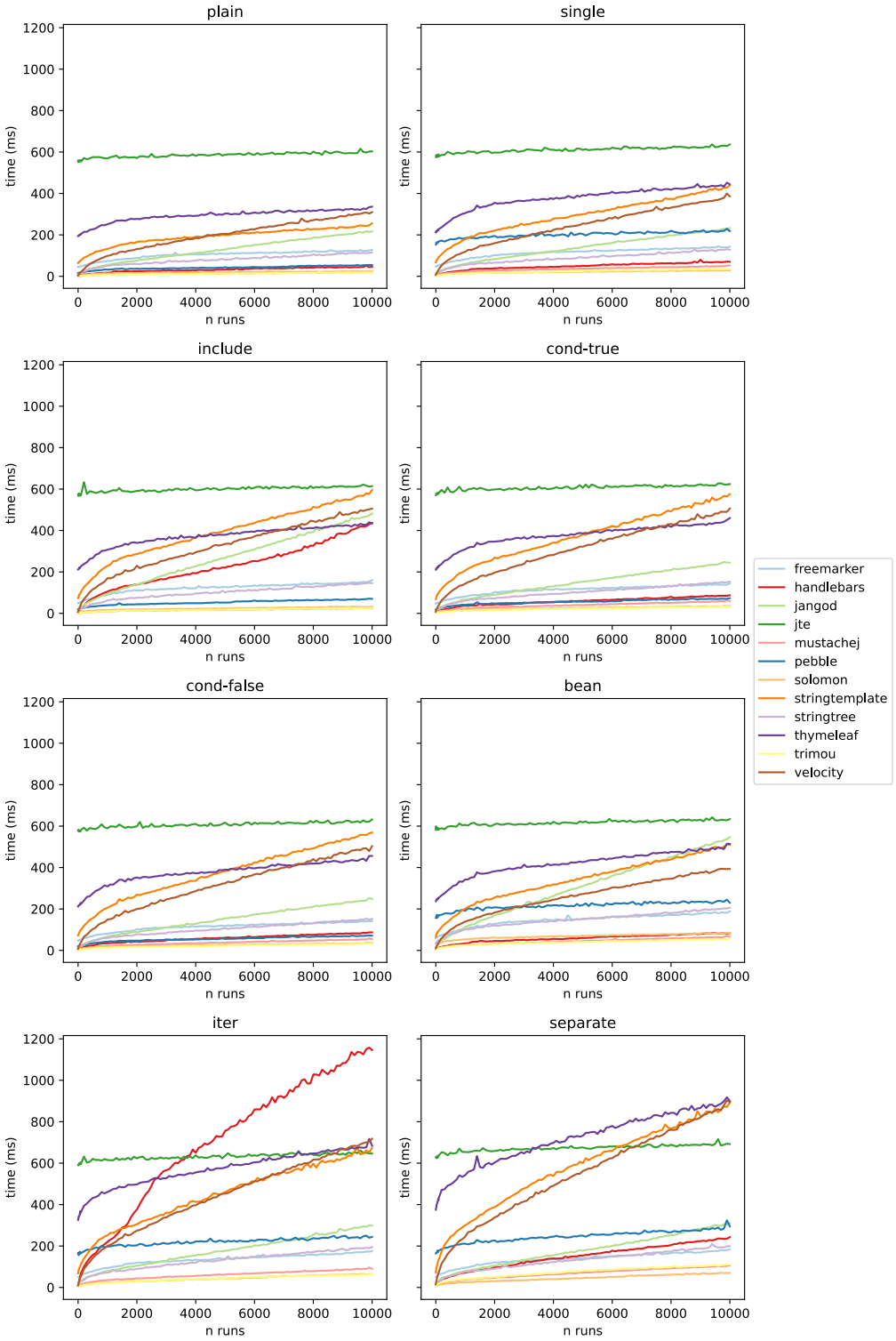


Figure 4.10.13: Performance comparison set 3 overview by running the script eight times and taking an average to mitigate the effect of unrelated noise

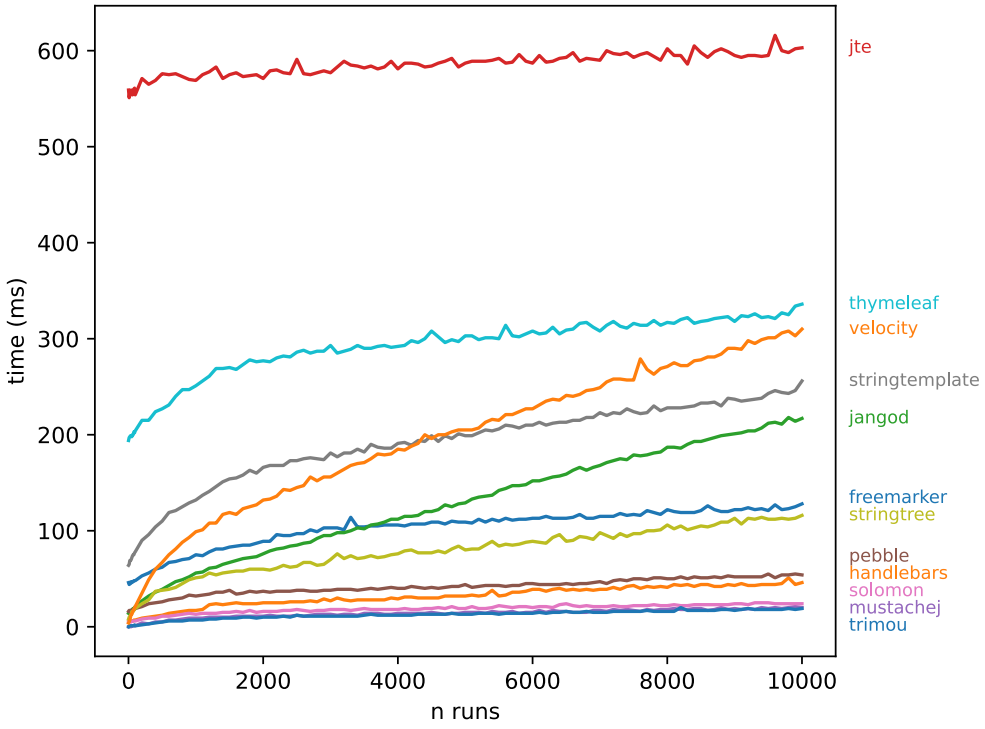


Figure 4.10.14: Set 3 performance comparison for the *plain* scenario, averaged over 8 runs

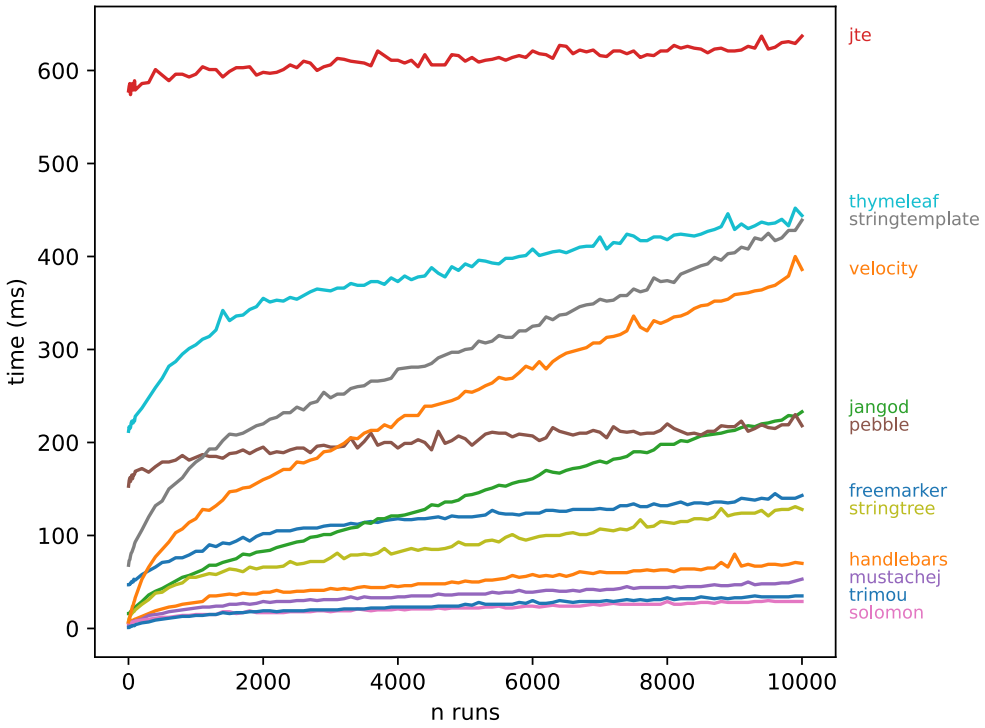


Figure 4.10.15: Set 3 performance comparison for the *single* scenario, averaged over 8 runs

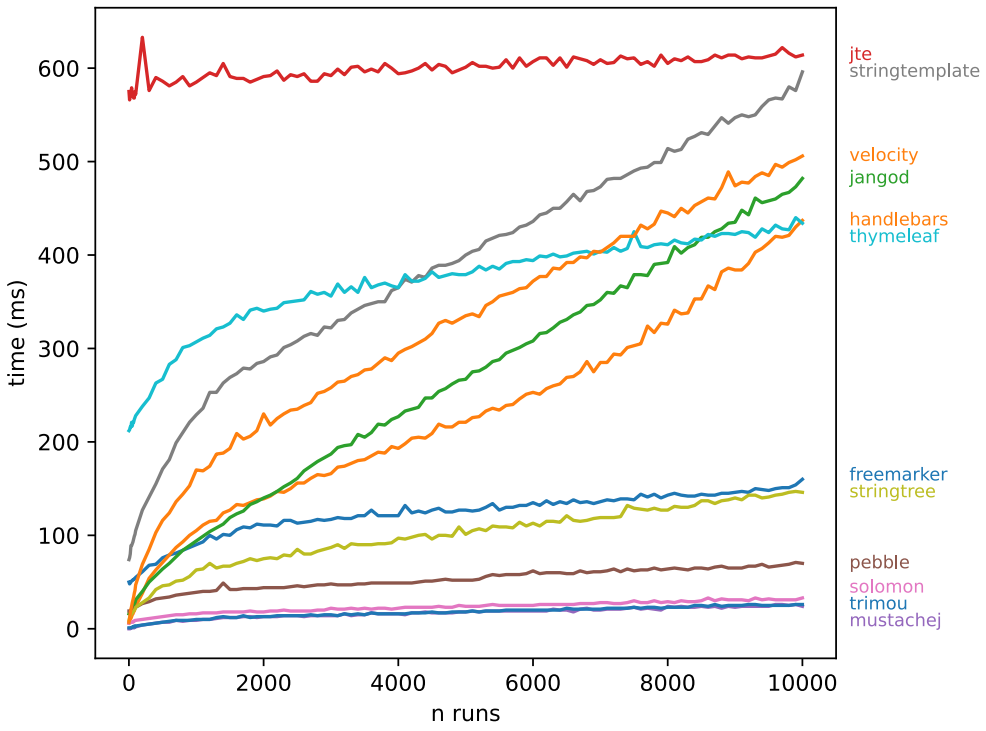


Figure 4.10.16: Set 3 performance comparison for the *include* scenario, averaged over 8 runs

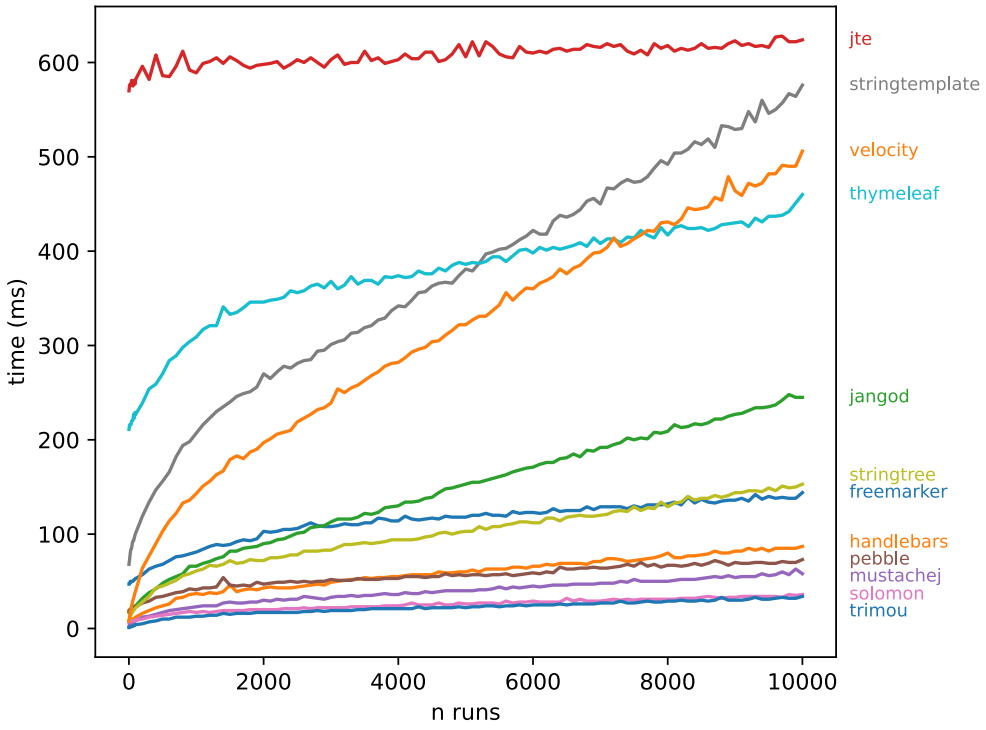


Figure 4.10.17: Set 3 performance comparison for the *cond-true* scenario, averaged over 8 runs

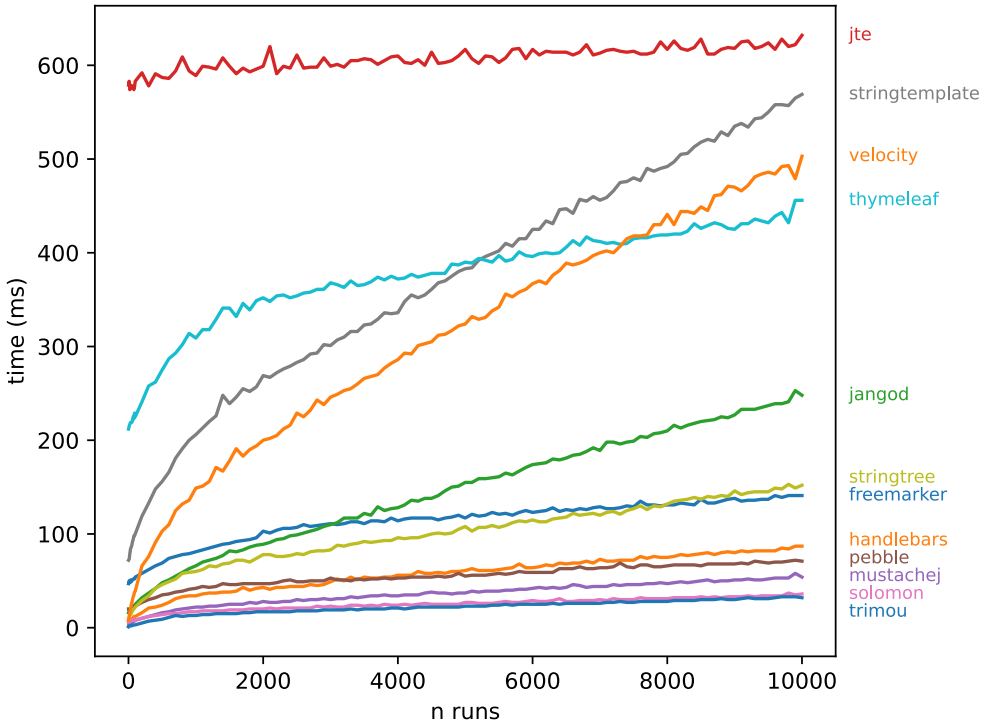


Figure 4.10.18: Set 3 performance comparison for the *cond-false* scenario, averaged over 8 runs

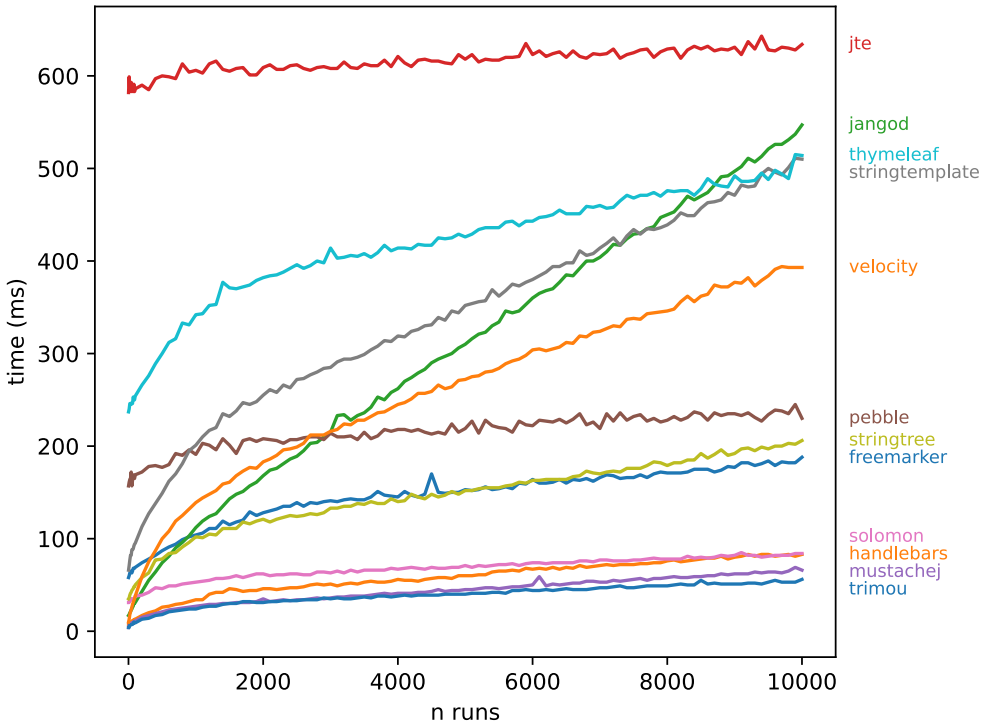


Figure 4.10.19: Set 3 performance comparison for the *bean* scenario, averaged over 8 runs

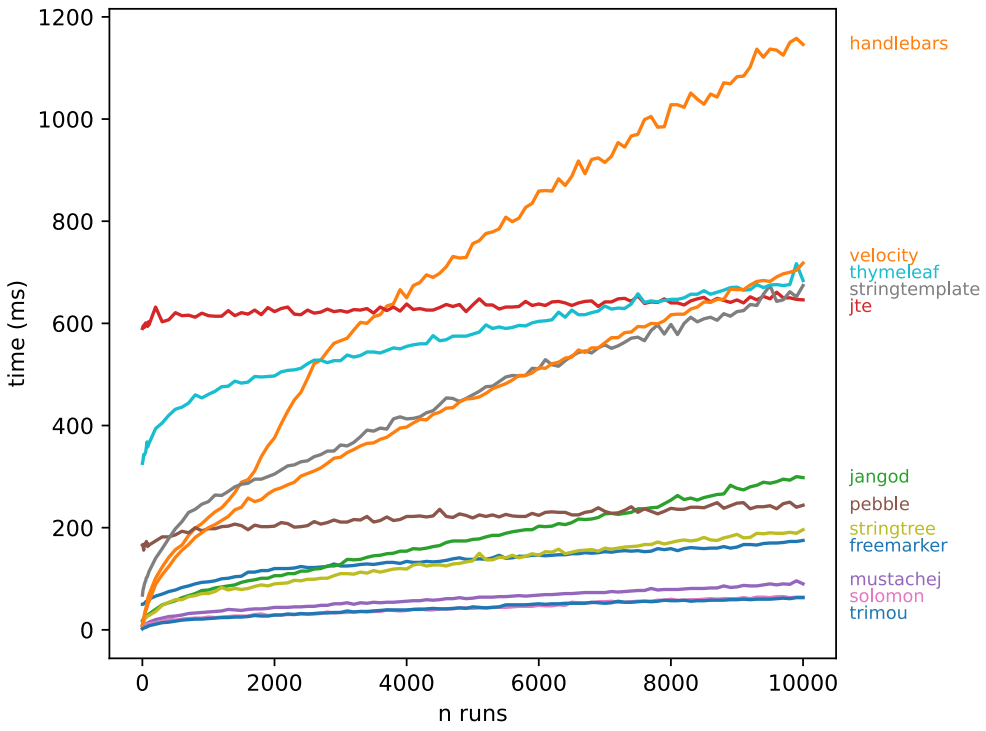


Figure 4.10.20: Set 3 performance comparison for the *iter* scenario, averaged over 8 runs

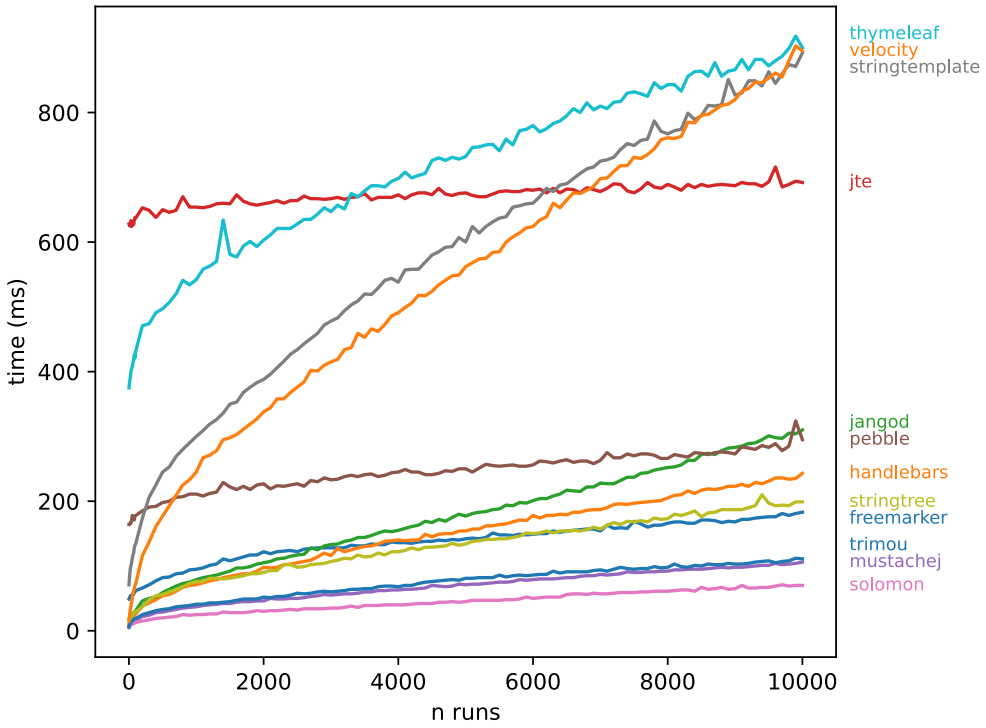


Figure 4.10.21: Set 3 performance comparison for the *separate* scenario, averaged over 8 runs

The addition of the *Handlebars* template engine to this set highlights its unusual performance characteristics in the "iter" scenario. While a template engine taking more time to process this scenario is not unusual, all the other template engines that exhibit this behaviour also show such increased time for the adjacent "separate" scenario. *Handlebars*, however, only takes this extra time for the "iter" scenario, returning to the lower cluster of template engines for the "separate" scenario. This issue is explored, and some potential solutions evaluated, in Set 4.

The increased number of data points in the Set 3 results shows more clearly the curved, rather than linear, nature of the performance characteristics of several of the template engines. It was initially thought that this could be an artefact of the script used to generate the results, which increases in steps of 10 until 100, then in steps of 100 from then on. Examination of the performance curves, however, shows that they do not have a clear turning point, but rather a smooth transition, with most template engine curves eventually tending towards a more linear slope.

Although the *Casper* template engine did not meet the inclusion criteria for this cohort (see Section 4.6.3), it stood out as the poorest performer during the original study, overshadowing some other results in a similar way to *Hapax* from this cohort. During Set 3, a separate experiment was conducted to compare the relative performance characteristics of *Casper* and *Hapax*, and the results are shown in Figure 4.10.22. These graphs illustrate the consistently poor performance of *Casper*, showing a largely linear increase in time with volume regardless of scenario. In comparison, *Hapax* performs much better than *Casper* except in the "iter" and "separate" scenarios in which it shows a progressively steepening slope and rapidly overtakes *Casper* as the worst performing template engine studied.

Further examination of the behaviour of *Hapax* showed a large number of messages being generated on the error stream during processing of these scenarios. To address this, the template engine driver for *Hapax* was reworked to address these errors (see Section 4.10.4). This brought the performance figures for *Hapax* into a similar range to the other template engines, so results for *Hapax* were re-introduced in Set 4.

4.10.4 Set 4

The measurements from Set 3 had shown that, when the "noise" was minimised by averaging several runs, the performance results of the template engines in this cohort were fairly stable, exhibiting the same characteristics across multiple runs

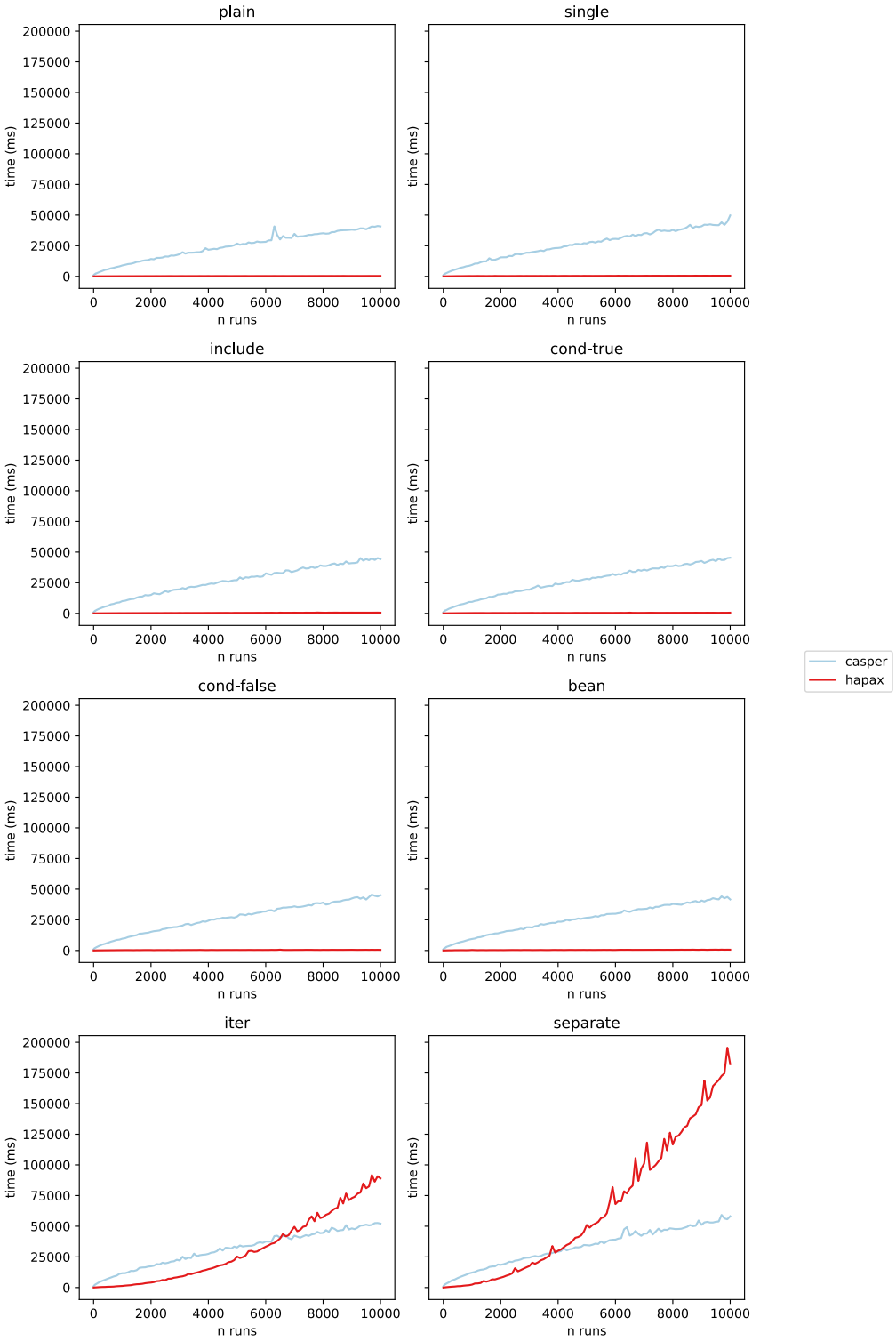


Figure 4.10.22: Set 3 *Hapax* compared with *Casper*

of the experiments. For Set 4, the number of data points was reduced again. The aim of this approach was to use the faster execution speed to enable more experimentation with template engine driver code and the specific templates used for each scenario for each template engine. The overview shown in Figure 4.10.23 and the detailed graphs in Figure 4.10.24 to Figure 4.10.31 show the results of an initial 8 runs of the Set 4 script. The reduction in the number of readings has resulted in a less smooth graph compared with Set 3, but the overall performance characteristics of the template engines remain the same, exhibiting the same slopes and minimum values.

Following the investigation into the poor behaviour of the *Hapax* template engine in previous sets and the discovery of a profusion of messages generated during the problematic scenarios, the code for the template engine driver was modified in an attempt to remove the messages and improve performance in these cases.

The problem was relatively subtle. As described in Section 4.7.1, each template engine driver has two methods. One method is called to initialise the driver, and the other is called to expand a template, provided with a context and the name of the template to expand. Typically there is some form of "impedance mismatch" between the structure and data types of the provided context, and the structure and data types of the context required by each specific template engine. Most of the template engine drivers for this cohort include a short section in the `expand` method to copy values from the supplied context to the required one. In the original driver for the *Hapax* template engine, this code used a `putContext` method that added the supplied context key and value to a *Hapax*-specific context created in the `init` method. This approach had been successfully used in several other template engine drivers. Most template-engine-specific context classes follow the example set by the built-in classes that implement the `java.util.Map` interface. These implementations treat a "put" operation for a key that is already present in the context as an "update" operation. If the provided value is the same as the existing value, then nothing changes.

For some reason, the designers of *Hapax* decided to ignore that convention and, instead, issue an warning message whenever an attempt is made to "put" a value for a key that is already present in the context. This added an overhead to every "put" of every item. In the case of *Hapax*, this was also compounded by a requirement to extract items from any collections in the context and place them independently in the context with a specific naming scheme. A similar warning message was then generated for every item in every collection.

The solution to this issue was to create a new *Hapax*-specific context at the start of

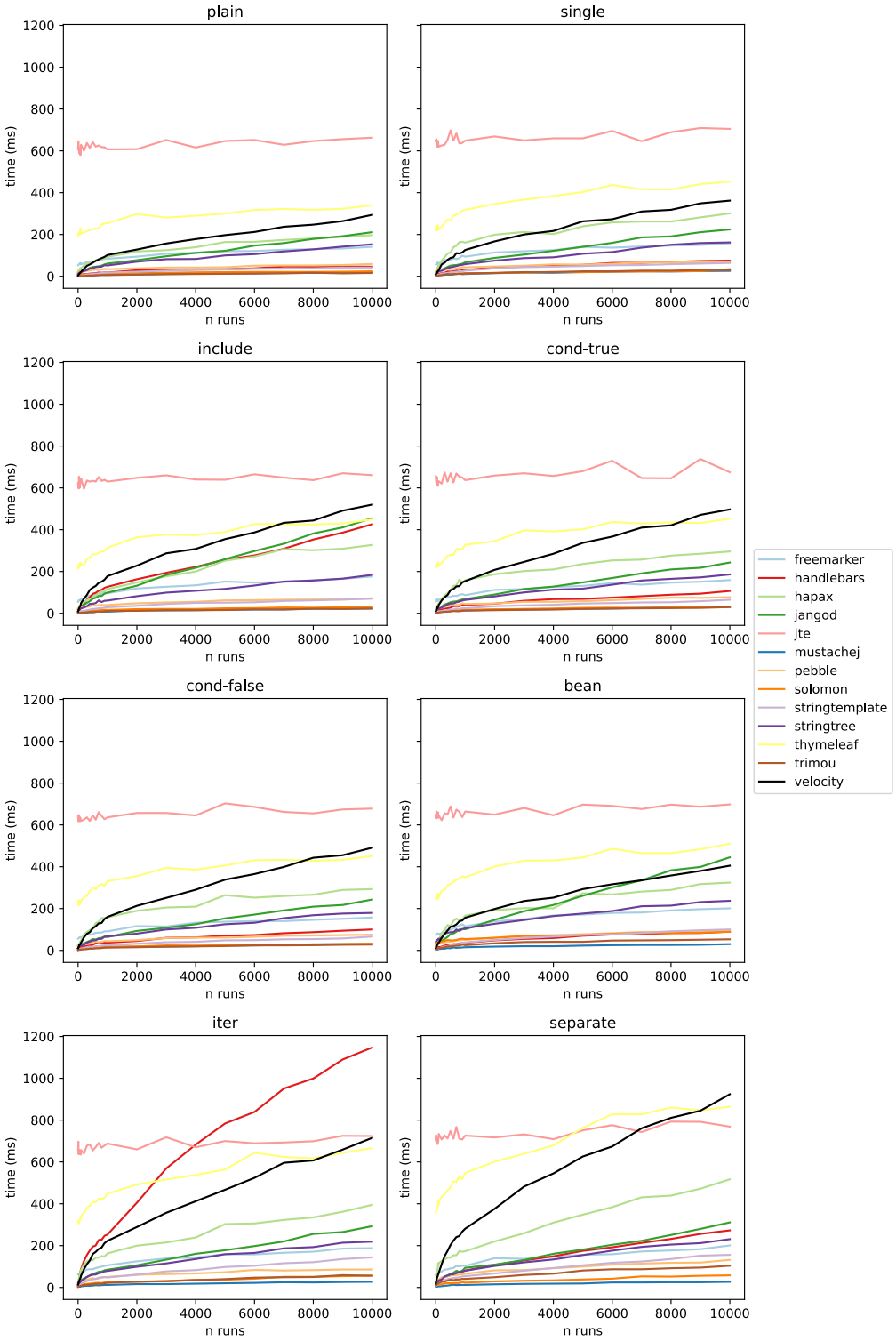


Figure 4.10.23: Performance comparison set 4 overview averaged over 8 runs with reduced number of samples

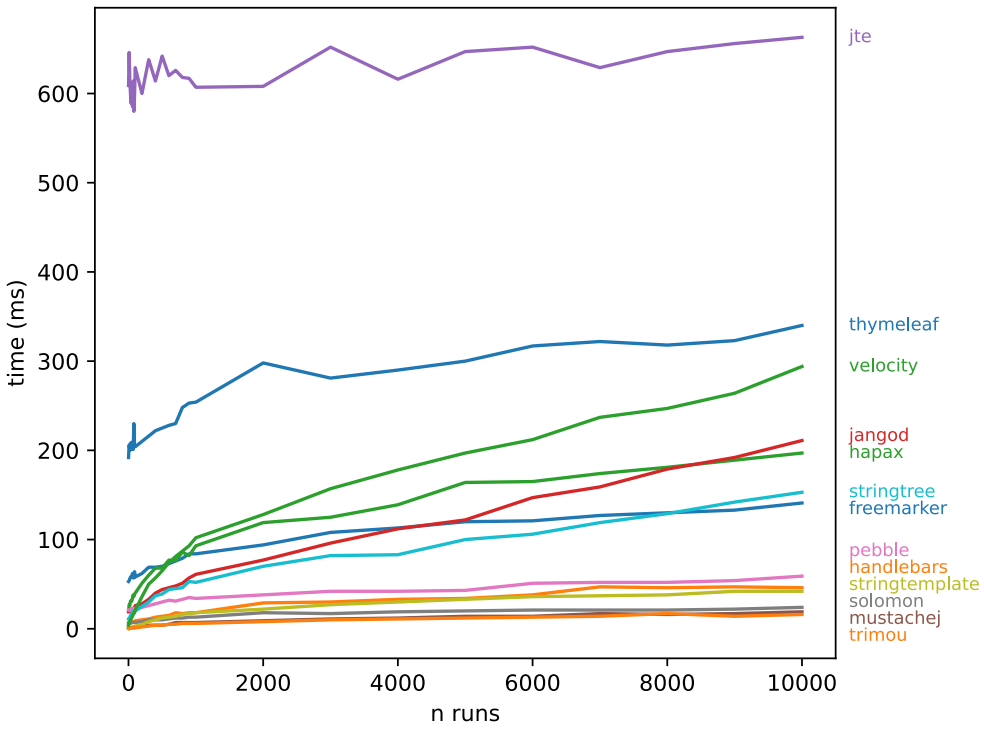


Figure 4.10.24: Set 4 performance comparison for the *plain* scenario, averaged over 8 runs

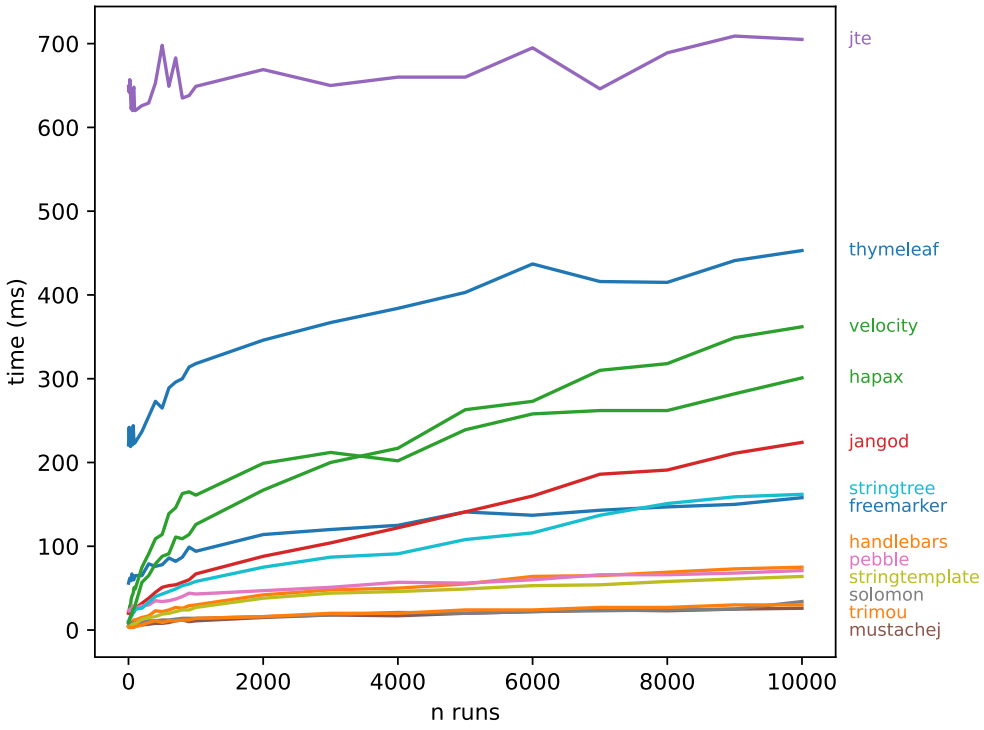


Figure 4.10.25: Set 4 performance comparison for the *single* scenario, averaged over 8 runs

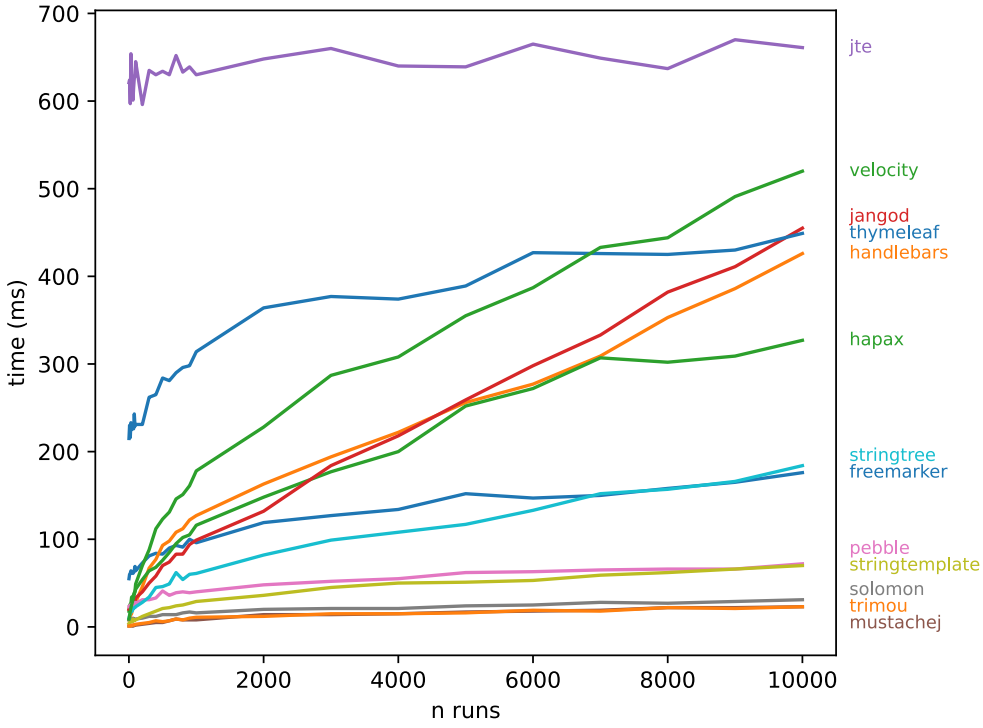


Figure 4.10.26: Set 4 performance comparison for the *include* scenario, averaged over 8 runs

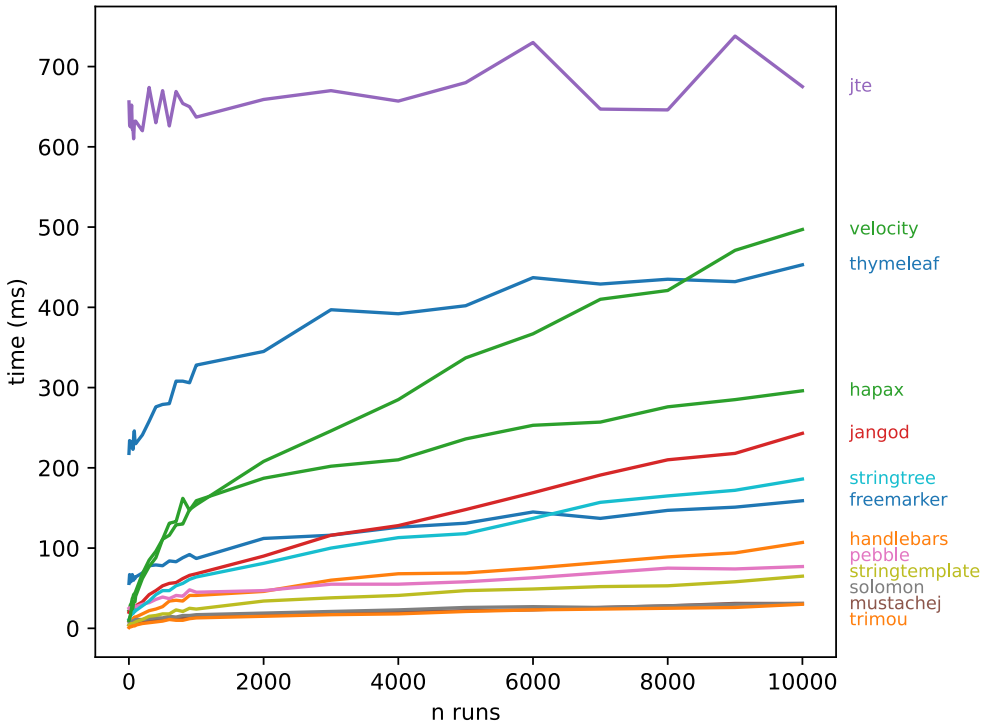


Figure 4.10.27: Set 4 performance comparison for the *cond-true* scenario, averaged over 8 runs

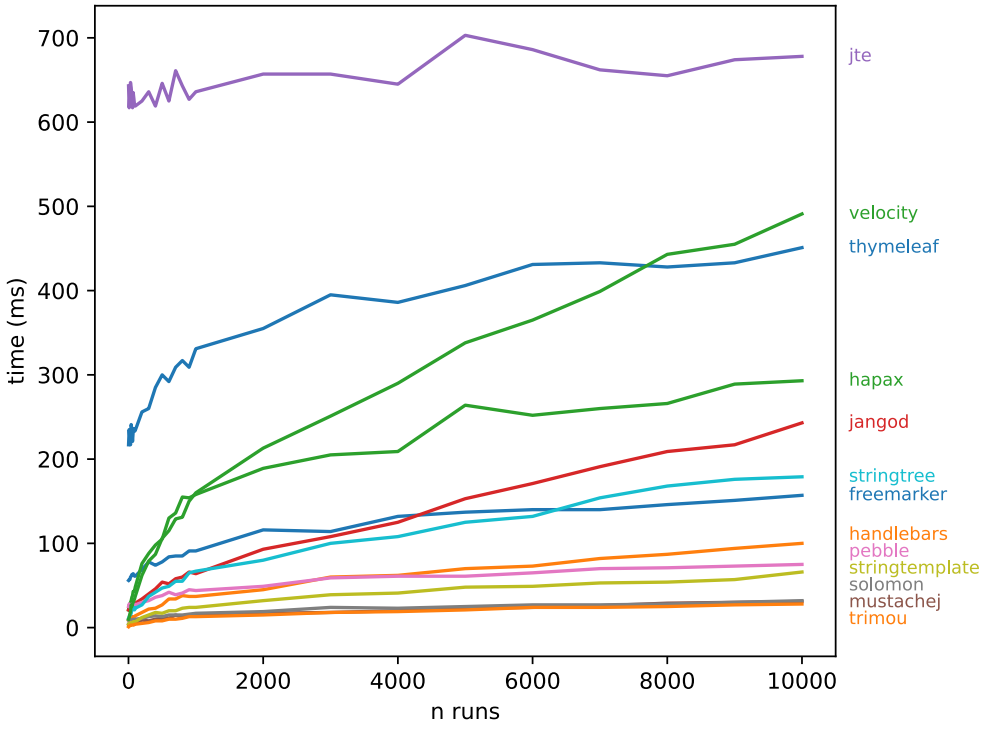


Figure 4.10.28: Set 4 performance comparison for the *cond-false* scenario, averaged over 8 runs

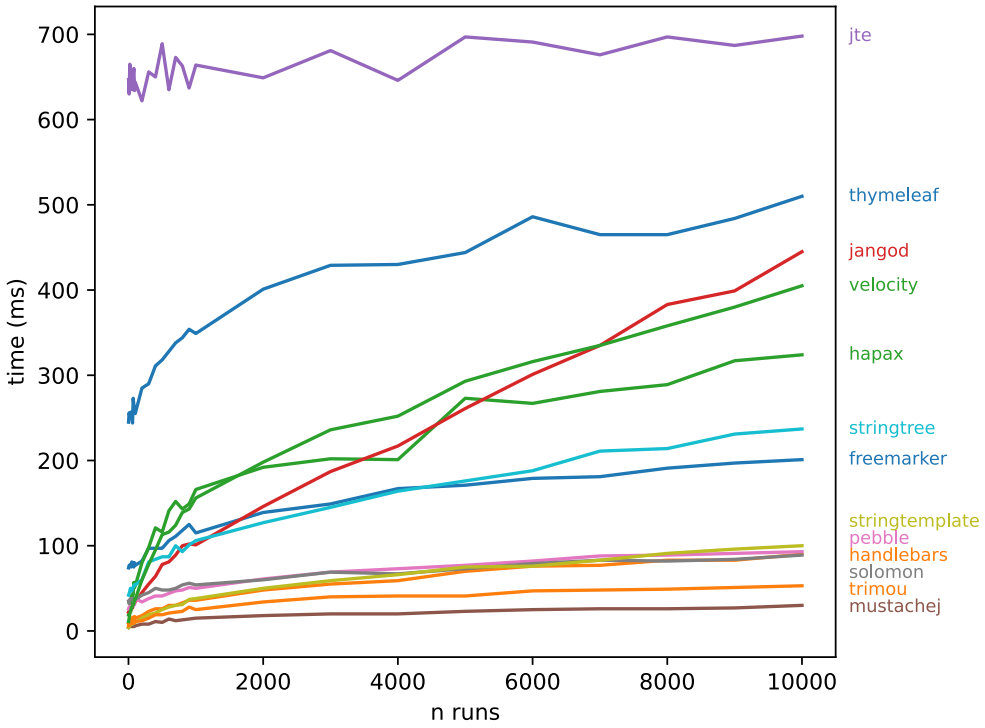


Figure 4.10.29: Set 4 performance comparison for the *bean* scenario, averaged over 8 runs

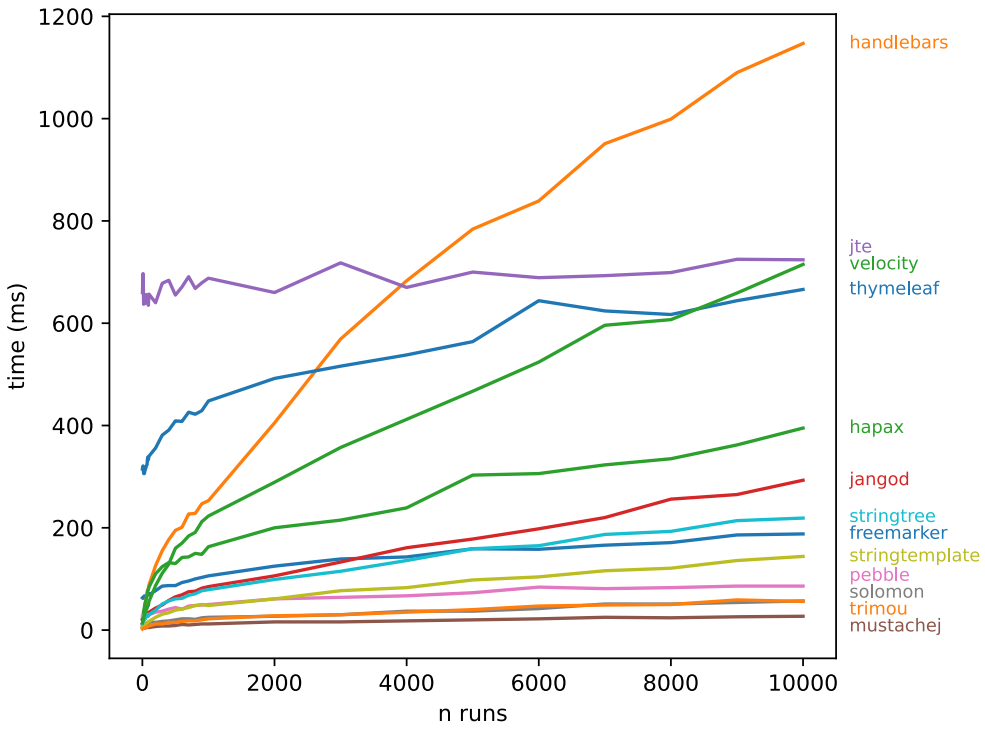


Figure 4.10.30: Set 4 performance comparison for the *iter* scenario, averaged over 8 runs

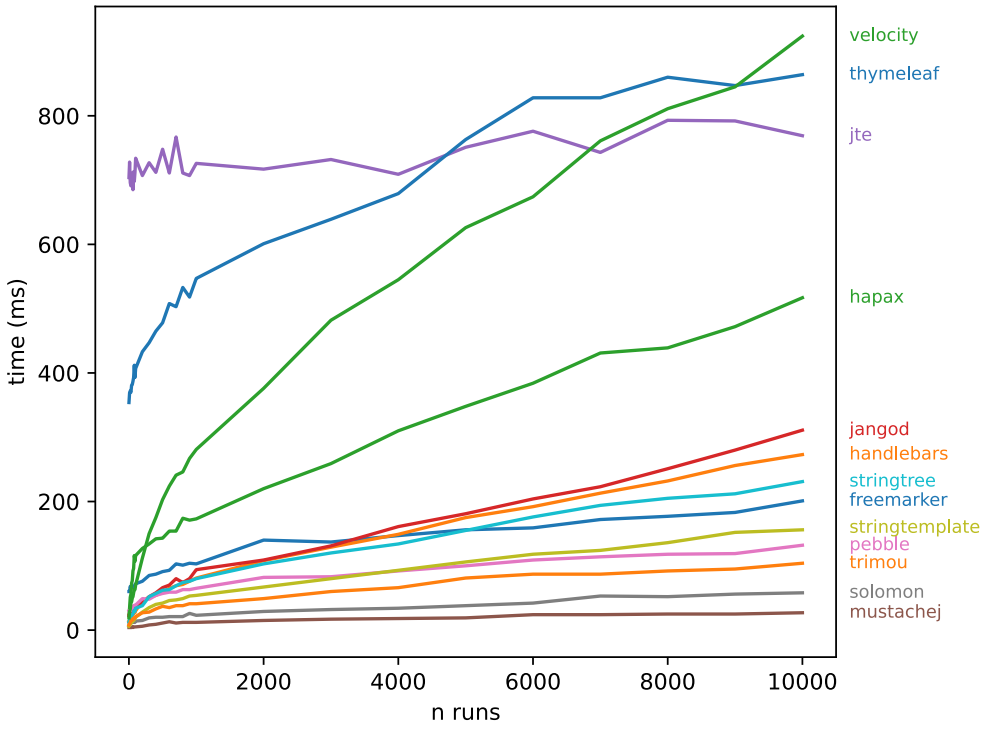


Figure 4.10.31: Set 4 performance comparison for the *separate* scenario, averaged over 8 runs

the `expand` method and discard it at the end, once template expansion is complete. This has its own issues, such as an accumulation of objects that need to be garbage collected, but as can be seen from the graphs in Figure 4.10.23, greatly improved the performance of *Hapax*, particularly in the "iter" and "separate" scenarios. The code for the original and updated *Hapax* drivers is given in Appendix B.5.2.

As discussed in Section 4.10.3, the *Handlebars* template engine also exhibited unexpectedly slow performance in the "iter" scenario. Unlike the case for *Hapax*, this did not also occur in the "separate" scenario. The difference appears to be because of a quirk in the way the Java implementation of the *Handlebars* template language processes templates. Like many of the templates in this cohort, *Handlebars* was designed primarily for the generation of HTML web pages. HTML is largely insensitive to whitespace. In an attempt to aid in the readability of *handlebars* placeholders and directives, the template parser routinely removes excess whitespace characters within the sub-templates used for loop and conditional expressions. This allows loop and conditional directives to be laid out in a manner similar to those structures in a programming language, using newlines and indentation to indicate the contents of sub-templates.

The "iter" scenario requires each item from the collection to be separated by a single space, but every attempt to add this to the template resulted in it being removed and not included in the output. Other implementations of the *handlebars* template language correctly determine that this whitespace is desired in the output, but this Java implementation appears to contain a "bug" that removes too much whitespace in this situation. In order to generate the correct output, the initial "iter" template for *Handlebars* was coded using template include directive to include a file containing just a single space character. This achieved the desired output, but at the expense of greater template processing time. It appears that *Handlebars* does not effectively cache included templates, and was taking extra time to process the sub-template and re-load the included template for every item in the collection.

In Set 4, an alternative approach was explored, of pre-storing a context value containing a single "space" character before expanding each template, and using that context value in a placeholder to ensure that the required space character would be included in the final output. This is not a perfect solution, as it requires an extra context value placeholder in any template that faces this problem and risks the name given for this context value clashing with the name of a context value provided by the application. This approach was, however, considerably faster than the original solution involving template inclusion, as can be seen in Figure 4.10.32. There may be other ways to achieve this, but they had not been

discovered at the time this set of measurements were taking place.

Another problem observed during the experiments for Set 3 was the failure of the *Stringtemplate* template engine to correctly include other templates. The template language used by *Stringtemplate* is extensively documented and claims that such a feature is supported. Unfortunately, the code to interact with the template engine from a user application is less well documented. The original design of the template engine driver for *Stringtemplate* worked correctly for everything except template inclusion, so addressing the issue with template inclusion was postponed until Set 4.

The code for the original *Stringtemplate* driver was written based on online examples. This seemed plausible, and worked in most cases. On further research, however, it appeared that this form of usage (create an `ST` object based on a supplied template and then call its `render` method) was intended as a simplified syntax for applications requiring only a single template. The created `ST` object only knows about the supplied template, and has no way to locate any others, so any use of template inclusion in the supplied template will never work.

Eventually, alternative code examples and documentation were located that explained that in order to use multiple templates a `StringTemplateGroup` object needed to be created and populated with templates. This code was added to the `init` method of the *Stringtemplate* driver, and the `expand` method altered to make use of the group. The code for the original and corrected *Stringtemplate* driver is given in Appendix B.5.3.

The results of running and averaging the Set 4 script following these changes is shown in Figure 4.10.32 and Figure 4.10.33. *Handlebars* now shows a much shallower curve in the "iter" scenario, *Stringtemplate* now produces the correct output for template inclusion and performs well in all the scenarios, and *Hapax* remains reasonable in both the looping scenarios.

While the averaging process used for graphs Figure 4.10.24 to Figure 4.10.31 has eliminated some of the "noise" by averaging multiple runs, and show the general relationship between the performance characteristics of the different template engines, they are still not as smooth as the curves from Set 3 in Figure 4.10.13. Figure 4.10.34 shows the result of applying a Savitzky-Golay filter (Schafer, 2011) to the averaged results of the second set of Set 4 measurements.

These graphs are slightly less busy than the raw data plotted in Figure 4.10.23, but also exhibit unexpected distortion of the data such as the apparent reduction in the time taken by *Thymeleaf* at high numbers of template repetitions. This appears to be an artefact of the smoothing algorithm that uses a polynomial fit

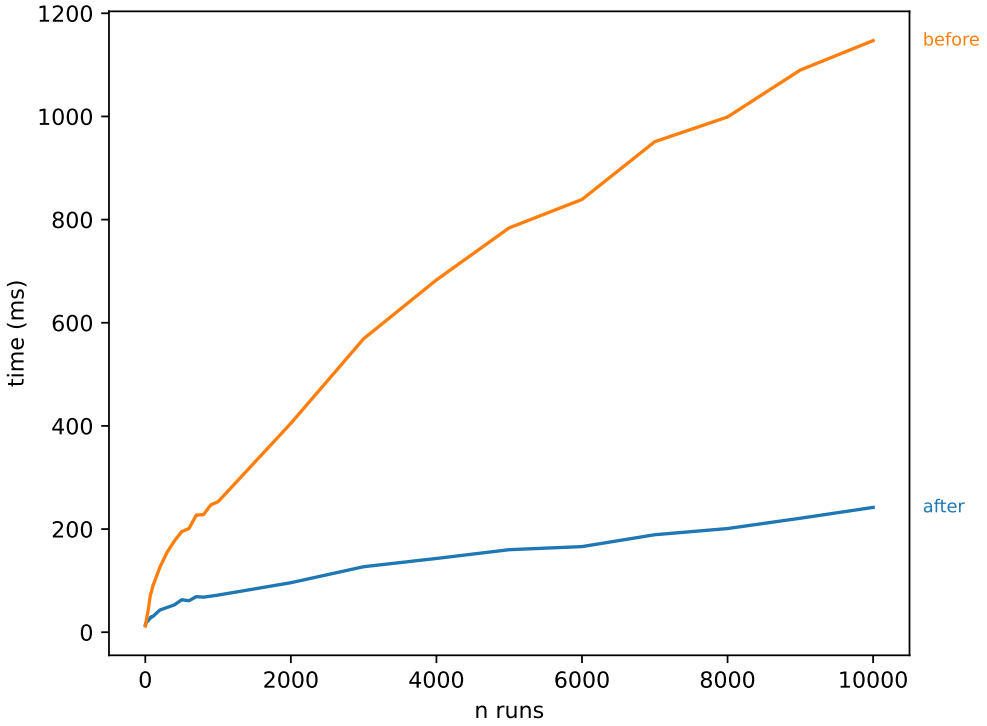


Figure 4.10.32: Comparison of *iter* scenario performance showing the difference between the original and updated *Handlebars* templates

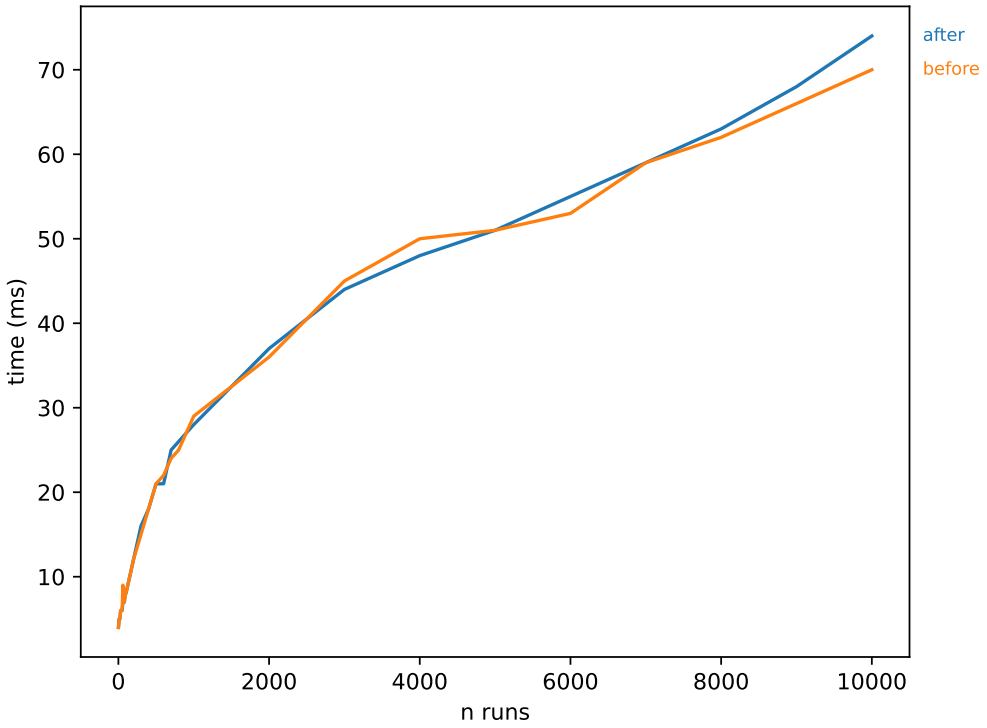


Figure 4.10.33: Comparison of *include* scenario performance showing the similarity in performance between the original and corrected *Stringtemplate* drivers

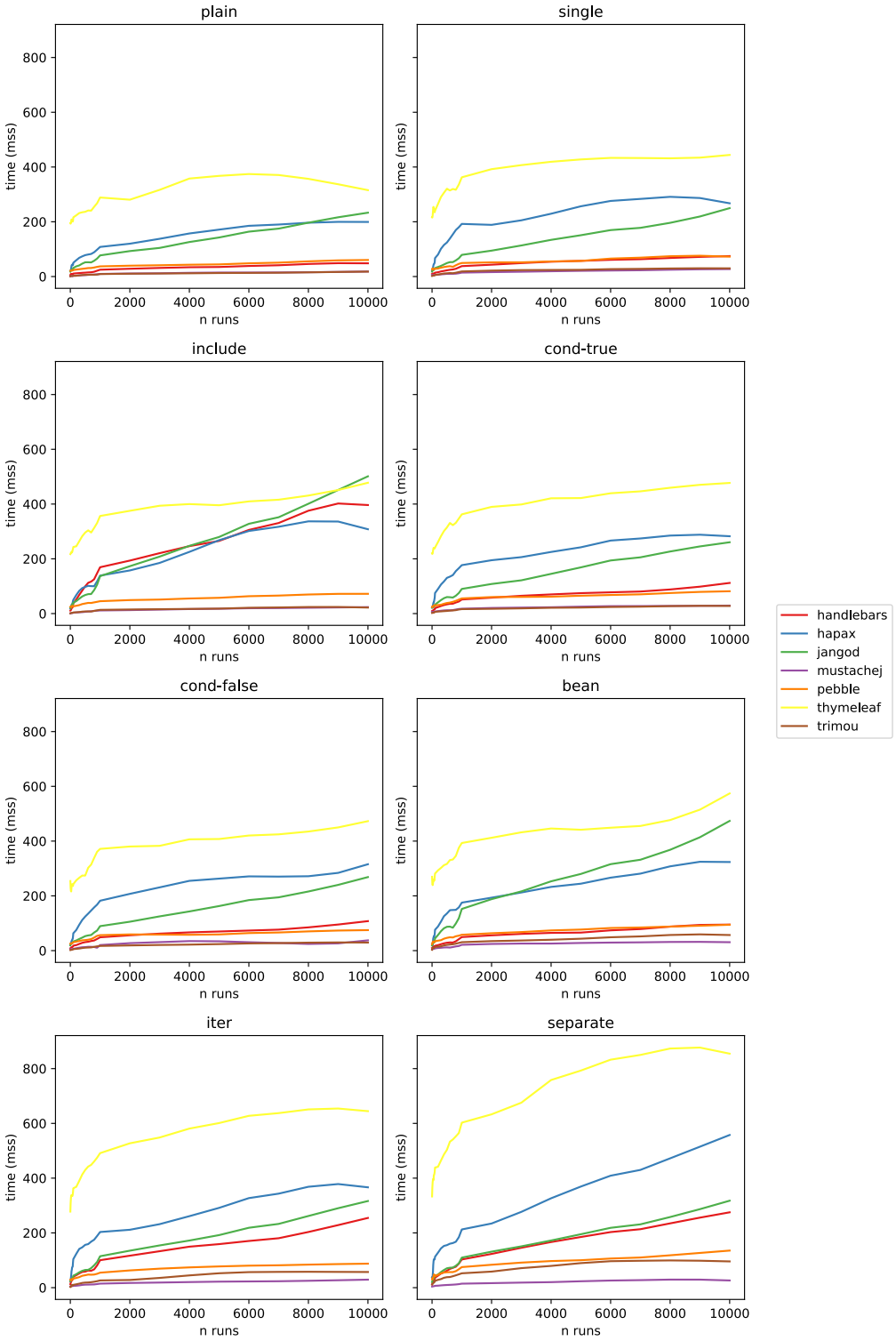


Figure 4.10.34: Overview of selected template engines showing examples of curve distortion when Savitzky-Golay curve smoothing is applied to reduce noise

approach. Such smoothed graphs should therefore not be used to draw conclusions about the detail of the template engine performance.

4.11 Discussion

The performance comparisons in this chapter exhibit a distinct difference in performance between the candidate template engines and also between the evaluation scenarios. When comparing at only a single load it can be tempting to assume that time taken scales linearly with load, but the results of these comparisons show that this does not always hold. This in turn implies that when selecting a template component based on performance, the choice should depend on the expected volume of traffic as well as other considerations.

The high minimum duration and relatively shallow slope of increase associated with the *JTE* template engine may be explained by its architecture. Unlike most of the other template engines, when *JTE* loads a template, it first translates it to Java code, then calls the Java compiler to compile that code to JVM *bytecode* for direct execution during template expansion. This process imposes a largely constant time overhead, even when the template contains no placeholders and could be transferred directly to the output. The rest of the template engines in this cohort use a variety of approaches from re-parsing and direct interpretation of the template each time to intermediate compilation into an internal data structure on loading, followed by rendering from that data structure to reduce the overhead of re-parsing the template.

These experiments revealed several issues with the template engine components themselves. Small changes to the driver code for *Hapax* made a huge difference to the performance of the component. Similarly, *Handlebars* provided alternative template syntax options to achieve the same output but with large differences in performance (see 162). The performance difference between *Trimou*. *Mustachej* and *handlebars*, which use largely similar syntax, showed that there is no direct relationship between the choice of template language syntax and the performance of the template engine.

As can be seen from the results of the individual scenario measurements in sets 1 to 4, the performance of a template engine varies depending on what is asked of it. The individual scenarios used up to this point have all been aimed at determining, and in some cases improving, the performance of each template engine for single, very specific, tasks. This is not generally how template engines are used in commercial projects. The most common application for template

engines is the generation of web pages, and a typical web page will have multiple placeholders and directives mixed in with large blocks of boilerplate text and page markup. The creation of such templates for all the template engines in this cohort would be a complex and potentially error-prone process. An intermediate representation and a software tool to generate such templates is explored in Appendix E. The performance of this kind of scenario is explored in more depth in Section 6.4, which also compares the energy use of the components.

4.12 Conclusions

The most striking conclusion from this second round of template engine performance tests is the wide range of different performance profiles across the cohort of template engines tested. The initial performance tests discussed in Section 4.3 highlighted a large difference in performance for one particular number of repetitions, but further investigation clearly shows that this relationship is not true for all scenarios and workloads. Some template engines have a large initial performance cost, but relatively stable performance regardless of the number of repetitions. Others show execution times that increase with the number of repetitions, but at a wide range of slopes. The relationship between duration and workload is also not the same for the different template scenarios. Some scenarios, such as ones involving iteration, appear to cause some template engines to work much harder and therefore take longer. The initial configuration of *Hapax*, for example, took so long to process the iteration scenarios that the results for other template engines were barely visible.

From a sustainability perspective, it is not enough just to measure and compare performance. There must also be some usable outcome. The usable outcomes that arose from the measurements during the several sets of this investigation into template engine performance were twofold.

- The first outcome was a better understanding of the characteristics of the performance of each of the template engines, which in turn led to an understanding that template engine performance can vary considerably by both scenario and volume. Both of these criteria need to be considered when selecting candidate components.
- The second outcome was a recognition of the scale of the difference made by seemingly small changes in the way a template engine is used or how templates are constructed. Selecting a template engine without understanding of these areas could result in software that generates correct

output documents, but risks very poor performance.

Such poor performance might in turn result in the software system either not achieving its goals or requiring considerably more computing resources, with the accompanying consumption of materials and energy, and emission of greenhouse gases, to achieve them.

Understanding software component performance characteristics, then using that understanding to select appropriate components and configure and use them correctly could therefore be a direct contribution to increasing the sustainability of software systems.

Chapter 5

An Apparatus To Compare Energy Usage

As discovered in Chapter 2, it is still uncommon to routinely compare the energy usage due to different software, both when selecting existing applications or components and when creating or maintaining new software. This chapter describes the design and construction of a prototype low-cost apparatus to allow software energy usage comparisons to be included in the regular software testing process. The prototype apparatus was evaluated by comparing the energy usage and performance of a range of web server software. In Chapter 6 the apparatus is also used to compare the energy usage of the components from Chapter 4.

The previous chapters have led to several important conclusions.

- The energy consumption of the internet is very large, and for sustainability this needs to be reduced.
- Software systems are a key factor in the energy consumption of the internet.
- Software systems vary widely in performance, even when performing the same task.
- Information is generally not provided about the energy-efficiency of software.

These conclusions in turn lead to the hypothesis that components and applications may vary in energy use when performing the same task. If true this implies that the energy use of web software systems may potentially be reduced by replacing components or applications with functionally equivalent ones that use less energy. In order to verify this initial hypothesis, a prototype apparatus was developed

to measure and compare the energy usage of different applications, servers, and software components, and therefore to determine whether the overall energy usage of server-side web software applications can be reduced in this way.

5.1 Existing Approaches

Attempting to determine the efficiency and energy usage of computer systems is not a new idea. What is new is the increasing realisation of the large environmental and climate impact of the world's computer systems and the corresponding desire to reduce that impact as much as possible, as soon as possible. This goal requires a holistic approach considering all aspects of the global computing ecosystem. This section provides an overview of existing approaches and research.

The largest and most diverse category of research into energy usage is that of electronics and computing hardware. This is with good reason. It is the physical electrical and electronic parts of the systems that consume energy in order to function and produce heat that requires cooling. Electronic components and subsystems are commonly provided with *data sheets* that include vital information such as voltage and current requirements, power dissipation, and operating temperature ranges. Manufacturers compete on optimising these numbers, and each new generation of technology often results in a general improvement. Reducing the overall energy requirements of the physical components that make up computing systems is an important way to reduce global energy usage and its associated greenhouse gas emissions.

Unfortunately, measuring the energy usage of electronic infrastructure is the point at which much existing research stops. Although built from electrical and electronic components, computer systems are not simple electrical systems that can be understood as a mathematical function of their components. The power consumption of a computer processor, and by implication the whole computer system, can vary by hundreds of watts depending on the software running at the time (derBauer, 2023). Software also influences the power consumption of memory, storage, networking, and other computing facilities (Basmadjian and De Meer, 2012).

However, software *changes*. The reason that software-defined systems are so different from pure electronic solutions is the ability to change and update the behaviour of a system without changing the hardware. When attempting to determine the energy usage of a programmable system, it needs to be

re-evaluated every time the software changes. Depending on the software development process in use, that can be many times per day (Shahin, Ali Babar, and Zhu, 2017). As was shown in Section 4.3, small changes can have a big impact. In an ideal scenario, whatever method is used to determine any increases or decreases in energy usage should integrate seamlessly into the software development process alongside tests for correct function and performance.

Many mobile and portable devices rely on batteries for power. The trade-off between weight, size, and battery capacity means that these devices have limited energy availability, and need to conserve energy wherever possible. Testing, and reducing, energy consumption is a key part of the design process for both hardware and software in battery-operated systems. Software that causes too great a battery drain is not fit for purpose. The simplest way to determine the energy usage of such devices is to measure how long they can continue operating before the battery is drained (Brown et al., 2006). This form of measurement, while arguably representative of real-world usage, is relatively coarse. Battery behaviour is complex (Panigrahi et al., 2001) and it can be difficult to find what is causing the problem when battery life is not as expected. Such battery life testing is, by its nature, not a quick process. Testing multiple usage scenarios can be prohibitively lengthy, and is therefore not really suitable for frequent automated testing.

A common way to measure the operational energy usage of any electronic system is connect a meter and observe the results (derBauer, 2023). This approach has the advantage of simplicity and familiarity to anyone with an electrical or electronics background. Such meters are generally available at relatively low cost and easy to connect in the power supply to the device being tested. There are several drawbacks to this approach, however, which make it less suitable for comparing the energy consumption of different software or different versions of the same software. A major issue is that it is a manual measurement, and requires a skilled observer. This does not integrate well with automated tests run many times a day, and is a challenge for geographically distributed teams. Manual measurement can also miss transient events, which might lead to a miscalculation of the real usage. On the whole, while this approach is better than nothing, it is neither accurate, repeatable, nor easily automated. Electronics manufacturers are aware of the problems of manual measurements on complex systems. Test and measurement equipment models are available that offer data logging, communication with other devices using a variety of protocols, and other complex features. Such test equipment is considerably more expensive than manually-operated equipment, though.

Computer component manufacturers are also aware of this problem, and some include power-measurement circuitry in processors or supporting chipsets. For example, modern processors from Intel include *Running Average Power Limit* (RAPL) circuitry that may be queried by software to determine the power consumption of specific components of the computer system (Travers, 2015) (Hähnel et al., 2012). In some scenarios, such as software that is very processor-intensive, this can be an effective solution, particularly when the measurement facilities are already included in the computer hardware being used for development, testing, and deployment of the software. This approach does have its disadvantages, however. The power measurement is not holistic, and may not include energy used by other components such as storage drives, cooling fans and network interfaces that are not part of the central processor or its supporting chipset. There is also the problem that the measurement and control software is being run on the same device, and its energy usage will be merged with the energy usage of the software being tested. In cases where the software under test uses a large proportion of the available computing facilities, the overhead of the test and measurement software will be small in proportion, but in scenarios with a wider range of software this can act to confuse the measurements.

If the software is deployed to a cloud hosting service, or co-located in a datacenter, the hosting partner may be able to provide some form of power usage information. Where available, this is typically aggregated across all the services in a project or all services hosted for a particular client. This kind of power consumption measurement has the advantage of being holistic, in that it includes power supplies and their associated losses as well as processors, memory, storage drives, cooling fans, network interfaces, and all the other equipment required to keep an internet service running. The disadvantage is its coarse-grained nature. Although data from such power consumption measurements can be used to track trends and indicate major increases or decreases in overall energy use, it is rarely precise enough to identify the contribution due to individual software.

In many commercial projects, there is no attempt to measure energy usage directly. Instead, monetary cost is used as a proxy. If the energy cost for running a project or service increases, then it is treated as if the project is using more energy, and vice versa. This approach has the advantage of fitting naturally with the information used to manage the financial aspects of a business. Unfortunately, it has all the disadvantages of hosting service measurements with the added complexity of changing energy prices. If this approach is then used to determine greenhouse gas emissions, it is also subject to strategies such as “carbon offsetting” that can make it appear that a product,

project, or service is "greener" that it is.

The above approaches are not exclusive, and many combinations are possible. For example, both Kaup, Gottschling, and Hausheer (2014) and Stoico et al. (2023) used an aggregated performance modelling approach in conjunction with a commercial external power measurement device to estimate energy usage in a variety of scenarios.

5.2 Apparatus Requirements

The existing approaches mentioned above have a mixture of advantages and disadvantages, but none quite fit the needs of a software development team that frequently and automatically tests individual services and complete systems during development to ensure that the end result meets requirements and is free of faults before deployment. This technique is known as continuous integration (CI) (Meyer, 2014) and is very popular in conjunction with agile development practices (Shahin, Ali Babar, and Zhu, 2017). An ideal solution for this situation would include the following features:

The proposed apparatus should:

- Measure the energy consumption of the whole system, not just certain components
- Support testing of existing software without manual modification or substantive changes
- Measure energy consumption during representative usage
- Be precise enough and fast enough to catch transient events
- Be programmable so it can be used in a CI build process
- Be able to operate repeatedly without human intervention
- Be easy to use

The following sections describe the design, construction, and testing of a prototype apparatus to address these requirements.

5.3 Design of the Prototype Apparatus

The apparatus described is a research prototype, developed on a small scale and to a low budget. Future implementations could revisit the architectural decisions, particularly the choice of power measurement and server hardware. Potential improvements and future possibilities are discussed in Section 5.8.

5.3.1 Solution Architecture

The key elements of the architecture are a computer platform on which to install the software to be tested (DUT), a way of measuring the power used by that platform, and one (or more) computers to act as clients and make requests to exercise the software (LOAD). This system also includes a separate computer to manage the operation of the apparatus and interface with the power measurement device (CTRL), and a separate server containing a database for data logging (LOG). All these systems are connected by a local network that can be separate from the internet during normal operation.

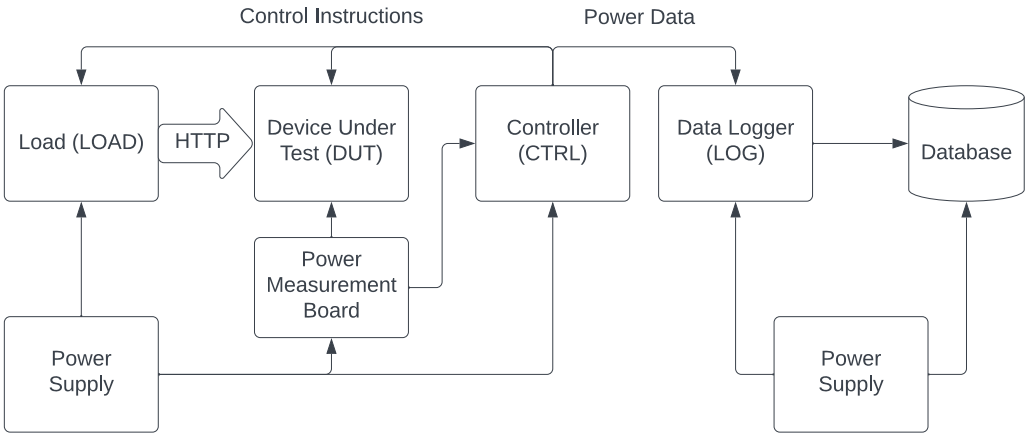


Figure 5.3.1: Prototype apparatus system diagram

5.3.2 Power Measurement Technology

During the construction of the prototype a range of power measurement technologies were considered (see Section 2.10), eventually settling on the INA260 chip from Texas Instruments (Texas Instruments, 2016) that can work with DC voltages up to 36V and currents up to 15A. This was purchased in the form of a pre-built evaluation board from Adafruit (Adafruit, 2023). The INA260 is controlled using 12C with a default address of 0x48. In this apparatus SDA and SCL lines are connected to an i2C interface on the CTRL device.

The INA260 is capable of both "high side" (in series with the positive voltage to the DUT) and "low side" (in series with the ground connection) measurements. In its default configuration, the most accurate readings are gained when used in a high side circuit (Texas Instruments, 2016), so the Adafruit board was connected in the high side of the power supplied to the DUT device. Adafruit provide a Python library for the control and measurement of this board that was used when building the control software for the apparatus.

5.3.3 Computer Hardware

The choice of power measurement hardware influenced the choice of computer platform. Any computer used for the DUT device would need to be DC powered and consume no more than 36V and 15A. A popular choice for research is the *Raspberry Pi* single-board computer (Raspberry Pi Foundation, 2023b), that fits the required power supply range.

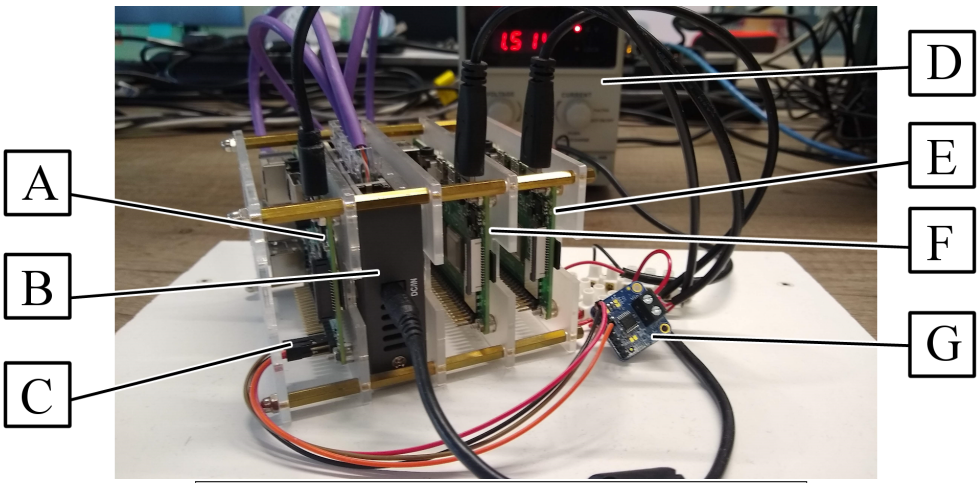
The Raspberry Pi computer is based on an ARM CPU (ARM, 2023), manufactured under license by Broadcom, and has various models with different CPU variants and memory sizes. The specific board chosen for the DUT device in this prototype was a Raspberry Pi 4 Model B with 8 GB of memory. Although the computer is physically small, it is capable of running much of the same software as larger and more power-hungry server systems (Varghese et al., 2015). To ensure that the DUT device would not just be idling during tests, a similar computer is used for the LOAD device. Although the architecture supports multiple LOAD devices, only one was implemented in this prototype. Software to simulate client interactions with a service is typically less resource-intensive than the service itself, so this was not considered to be a major issue.

For simplicity the CTRL device also used a Raspberry Pi. This role did not need such a high specification, so a cheaper Raspberry Pi 3 Model B with 4GB RAM was chosen.

In all cases the computers used a variant of the Debian Linux operating system.

5.3.4 Data Logging

The machine used for data logging needed larger and faster data storage than available on a Raspberry Pi, so a generic Intel-based desktop PC was chosen. The logging system consists of two main software components: A PostgreSQL



		D	5V Power Supply
A	CTRL Device	**E**	DUT Device
B	Network Switch	**F**	LOAD Device
C	I2C Connector	**G**	INA260 Board

Figure 5.3.2: Prototype hardware annotated to illustrate hardware components

database and a service to receive data from the CTRL device and store it in the database. The data logging service provides a REST (Fielding, 2000) interface for the CTRL device to submit experiment status and progress data as well as measurements from the INA260.

For near-real-time data visualisation during experiments a Grafana dashboard (Grafana, 2023) was configured to read, aggregate, and display power usage data every second from the database. Data gathered during experiments remains in the database for subsequent analysis.

5.3.5 Power and Networking

As discussed in Section 5.3.2, the selected power measurement technology only works with DC voltages. While this is perfect for the Raspberry Pi boards, which all work with 5V DC power, it is not suitable for traditional desktop computers that require an AC supply from which they generate DC voltages appropriate to the different parts of the system. For this reason the prototype apparatus uses two separate power sources: a stable 5V DC supply for the Raspberry Pi devices and an AC supply for the data logging server.

All the computing devices needed to be networked together, so in this prototype, a low-cost, low-power, 5-port gigabit switch is mounted alongside the Raspberry Pi devices. The ports link the three Raspberry Pi devices and the logging server as well as providing an "uplink" to a network with PCs for software development, debugging and results analysis. The selected network switch also requires 5V DC power, and is powered by the same supply as the Raspberry Pis.

5.3.6 System configuration

Each participating computer needs to know the IP addresses and ports of any other services it makes use of. Early versions of the prototype were manually configured using a simple key-value text file. This rapidly became cumbersome when using the system with other networks with different IP allocation rules. It also became a problem whenever one of the participating computers needed to be replaced or re-installed.

To overcome this issue a service registry was added that provided a single point of access for whatever services were added to the system. This registry functions in a similar way to Eureka (Netflix, 2012). On start-up, every service registers itself with the registry and receives a *lease* with a limited duration. The service

must renew this lease before expiry or be removed from the registry. This leasing process keeps the list of services "fresh" (Arnold, 1999) and reduces the risk of attempting to connect to an obsolete or unavailable service. When one service needs to communicate with another, it requests details of the endpoint from the service registry.

To minimise the number of separate machines in the system, the service registry is installed on the LOG system. This machine is allocated a static IP address, so that a fresh installation of any of the other participants can find the server, register, and obtain IP addresses and ports of the other participating services, even in a DHCP environment where IP addresses may not be consistent.

5.3.7 Software installation and version control

As well as simplifying the hardware, the choice to use similar devices for the DUT, LOAD, and CTRL roles made it possible to use a single "disc" image for all three participants. The Raspberry Pi has no hard drive or SSD storage in the usual sense, but instead uses an SD card. While this approach has some issues with performance and reliability (see Section 5.7), it has the advantages that the cards are removable and relatively easy to copy. During development a single "golden" card image was maintained containing up-to-date versions of all the services. Whenever a card became corrupted, failed, or otherwise needed replacing, it was a simple matter of copying the golden card image.

While basing all the Raspberry Pi systems on the same image made version management and failure recovery much simpler than a completely manual configuration, there was still one more step required. When the system starts it needs to know which role it should perform. In the current prototype, this information is contained in a plain text file at a known location on the card image, and must be manually edited before booting the Raspberry Pi device. Other approaches are potentially available to avoid the need for this, and are discussed in Section 5.8.

5.3.8 Preparation

In operation, the apparatus involves dynamic participation by services running on all four of the main hardware components (CTRL, DUT, LOAD, and LOG). Before measurement can start, however, the system needs to be configured for the specific measurements. This proceeds in four steps, which can be done either manually (via SSH terminal sessions to the appropriate devices) or under control

of a CI build system that has remote access to the devices.

1. Install the software to be tested. When comparing external applications this process will usually involve installing the applications according to their specific instructions, but may involve the same "Infrastructure as code" tools such as *Ansible*[1], *Chef*[2], *Puppet*[3], or *Terraform*[4], which would be used to deploy the application for real (Rahman, Mahdavi-Hezaveh, and Williams, 2019). When comparing versions of an internal application, this process may also include fetching the software from a code repository such as *GitHub*.

2. Create scripts for the apparatus to use to start and stop the software. These scripts have standard names and locations known to the measurement software and enable the system to start and stop arbitrarily complex installations without the need to modify the measurement software itself. As will be seen in Section 5.3.9, these scripts not only start and stop the software, but also notify the measurement controller when start-up and shut-down is complete, to support the measurement of software with lengthy or complex start-up and shut-down behaviour.

3. Configure the client service on the LOAD device. When evaluating this prototype, client requests were simulated using the *Siege*[5] load testing tool so the configuration consists of creating a script to call *Siege* and to notify the measurement controller when it has finished. *Siege* was selected after an examination of several alternatives. Many popular load testing tools such as *JMeter*[6] and *Locust*[7] are designed for programmable scripting and performance testing of specific HTTP requests. For realistic comparison of energy use it is important to emulate a web browser that automatically fetches any associated page resources such as images, CSS, and JavaScript. *Siege* does this and is easy to control with command-line parameters. The process of selecting and evaluating potential load testing tools was not exhaustive, for reasons explained in Section 4.1.

4. Provide the measurement system with identifying information about the upcoming measurement so that the resulting data can be retrieved from the database when the measurement is completed. Typically this information is in the form of a textual name and a version or sequence number. The

[1] `https://www.ansible.com/`
[2] `https://www.chef.io/`
[3] `https://www.puppet.com/`
[4] `https://www.terraform.io/`
[5] `https://github.com/JoeDog/siege`
[6] `https://jmeter.apache.org/`
[7] `https://locust.io/`

identifying information can be provided either manually using a web form, or by the CI process using an HTTP API.

5.3.9 Measurement Operation

Once preparation and software installation is complete, measurement is started either by an API request from a CI process or a user clicking on a button on the CTRL web interface (Figure 5.3.5). The apparatus then runs a sequence of operations managed by the CTRL device, as shown in Figure 5.3.3.

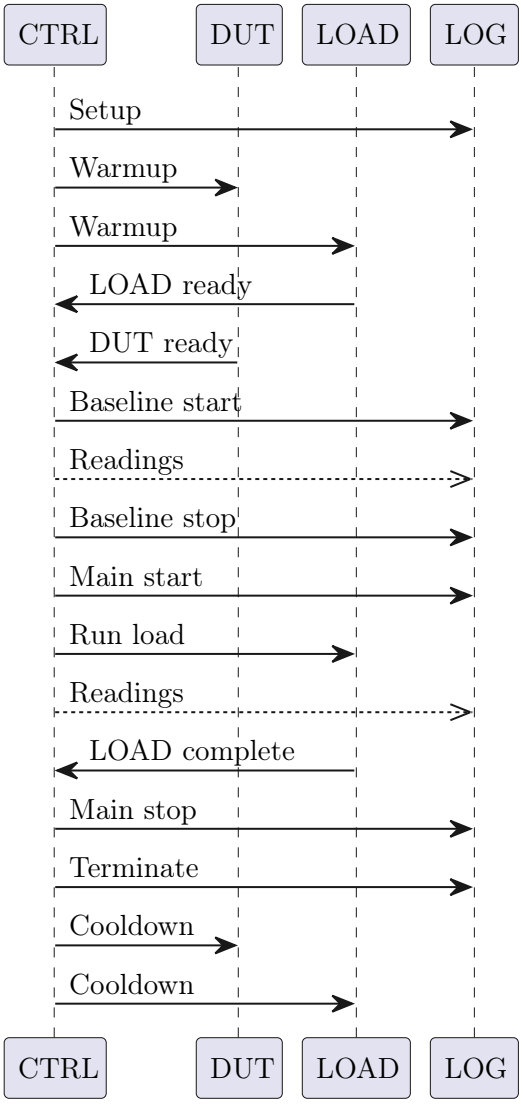


Figure 5.3.3: Measurement operation sequence diagram

The sequence begins by notifying the LOG that a measurement is starting, which enables it to group subsequent measurements together as a single experiment for later analysis. Following this, *Warmup* instructions are sent to DUT and LOAD.

These start the services using the scripts described in Section 5.3.8, which in turn notify CTRL when they are complete. When both DUT and LOAD are ready, the first of two phases of measurement begins.

Computers are complex systems and the amount of power they use, even when not explicitly running application software, can vary due to a range of factors, including ambient temperature (Jin et al., 2022), operating system software versions and updates (Williams and Tang, 2015), network activity (Canek, Borges, and Taconet, 2022), and other time or environmental factors. To eliminate as much of this variability as possible, the apparatus runs an initial phase of measurement with no LOAD activity which is then used as a *baseline* for the measurements under load that follow. LOG is also notified of the start and end of this baseline measurement phase, so that these measurements are not confused with the active phase.

When the baseline measurements are completed, the second, active, phase begins. LOG is notified to expect active readings, the LOAD process starts, and readings are collected until LOAD reports that it is complete, at which point LOG is notified that the experiment is ended. All messages to LOG are timestamped so that the length of time taken to service the load can also be included in the analysis. Finally *Cooldown* instructions are sent to DUT and LOAD to revert changes made during *Warmup*, if required.

Execution time for the whole measurement cycle varies depending on the choice of load and the requirements and capabilities of the software being investigated. During validation of this prototype, the duration of each measurement cycle was typically of the order of a few minutes.

5.3.10 External Interfaces

Each of the platforms which comprise the comparison apparatus provide multiple ways for external software to interact with them. To enable automated installation and configuration of services to be compared and client software to exercise them, the DUT and LOAD devices provide secure shell (ssh) access. Software installation and removal often requires superuser privileges, so this is also available to authenticated ssh users. This ssh access is also used by the CTRL device when it calls the *Warmup*, *Cooldown*, *Start*, and *Stop* scripts to manage comparison runs. Such access comes with significant risk, however. Accidental or malicious commands over this channel could damage the operating system or the software that operates the DUT and LOAD services or even install other malicious or exploitative software on the apparatus. The risk of this

is mitigated somewhat by two factors. The first factor is the isolation of the apparatus from other networks except when required to install software, and the second is the easy and regular re-installation of each system from an externally-created and managed "golden" disk image. In most cases for this prototype it is both simpler and safer to start each group of tests from a fresh disc image rather than relying on a clean de-installation of a previous candidate.

The REGISTRY, CTRL, and LOG systems all provide safer interaction mechanisms in the form of REST interfaces for use by integration testing software and web interfaces for manual use by humans. In each case the web interface is mainly just a wrapper around the REST requests. All operations provided by the web interfaces can also be achieved by calls to the REST API.

A screenshot of the web interface for the REGISTRY is shown in Figure 5.3.4. At the top is a table of the currently registered devices in the system with their IP addresses, the remaining time before their leases need to be renewed, and a set of options for each one to deregister, remove and shutdown individual services. This table is automatically refreshed using JavaScript, but should that fail for some reason there is also a "link button" to force a refresh of the table.

The REGISTRY web interface also includes a form for manually registering a device. This is typically only used for testing, to check that the "lookup" process is working correctly, because it does not set anything running that will renew the lease when it expires. Also used for system testing is a form that calls the REST endpoint used for looking up the IP address of a device. The final form on this page allows setting the duration of future leases. By default, the system issues leases valid for 1 hour (3,600,000 milliseconds) but when stopping, starting, or replacing devices or services, it can be useful to set the lease expiry to a shorter value.

In addition to the forms, the REGISTRY web interface also includes "link buttons" which remove expired leases (typically from devices that have been replaced and gained a new IP address), clear the whole lease table, and shutdown the registry service.

All the forms and buttons on each of the web interfaces have attached JavaScript code that shows a temporary message to confirm success or failure of the operation.

A screenshot of the web interface for the CTRL role is shown in Figure 5.3.5. At the top is a status display for the currently running experiment. "running" is true when an energy measurement is in operation. "child" indicates the status of the process that interacts with the INA260 device over the I2C interface. This

Experiment Registry

Status at 2024-03-21 15:06:15

Name	Address	Expiry	Remaining	Operations
LOG	http://192.168.0.187:9996/	2024-03-21 16:02:41	3387 secs	deregister ~ remove ~ shutdown
CTRL	http://192.168.0.178:9999/	2024-03-21 16:04:04	3469 secs	deregister ~ remove ~ shutdown
DUT	http://192.168.0.179:8000	2024-03-21 16:05:18	3543 secs	deregister ~ remove ~ shutdown
LOAD	http://192.168.0.177:8000	2024-03-21 16:05:53	3578 secs	deregister ~ remove ~ shutdown

Lease duration: 3600000 milliseconds

- Refresh status

Register

Role: Address: Submit

Lookup

Role: Submit

Set Default Lease

Duration: Submit

General Operations

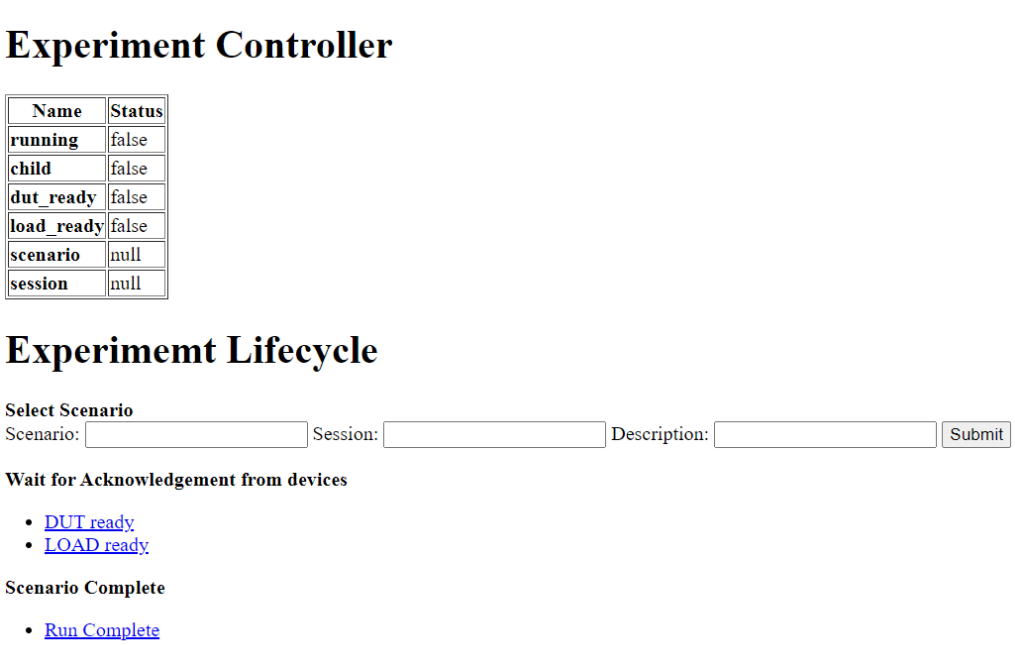
- Reap Expired Leases
- Clear Lease Table
- Shut Down Registry

Figure 5.3.4: Web interface to the Registry

process is started at the beginning of each experiment and shutdown at the end. "dut_ready" and "load_ready" indicate whether the relevant service has confirmed completion of its *Warmup* script. "scenario" and "session" reflect the information entered for the current experiment.

An experiment is started by entering a scenario name and a session number for that scenario and clicking "Submit". These values are separate to make identification easier when executing multiple runs of a single scenario for averaging. There is also an optional field for some descriptive text that will be stored in the database with the details of the experiment but has no automated use. As the experiment progresses, the values in the status display are updated.

For testing purposes the CTRL web interface also provides "link buttons" to manually trigger the normally-automated responses from the DUT and LOAD devices, as well as a way to manually mark a run as complete. These options



Experiment Controller

Name	Status
running	false
child	false
dut_ready	false
load_ready	false
scenario	null
session	null

Experimemt Lifecycle

Select Scenario
Scenario: Session: Description: Submit

Wait for Acknowledgement from devices

- DUT ready
- LOAD ready

Scenario Complete

- Run Complete

General Operations

- Shut Down Controller

Figure 5.3.5: Web interface to the Experiment Controller

can be useful when testing that the software to be measured is installed correctly and that valid *Warmup*, *Cooldown*, *Start*, and *Stop* scripts have been created and placed in the correct places. The "Run Complete" button is particularly useful if an experiment fails to terminate and is continuing to send data to the LOG service and the database.

Just as with the other interfaces, the CTRL interface also provides a button to shut down the CTRL service.

A screenshot of the web interface to the LOG role is shown in Figure 5.3.6. At the top is diagnostic information about the current active session, if any, and the current number of stored readings from the INA260 in the `log` table.. The database consists of two active tables: `log`, in which each row is a single timestamped voltage and current reading from the INA260; and `session2`, in which each row describes a measurement session with its scenario name, session number and description, as well as the starting and ending timestamps for the "baseline" and "active" readings. The SQL creation script for the database is given in Appendix D.

The LOG web interface provides a form to create a session in the `session2` table and some "link buttons" to set the start and end timestamps for the "baseline" and "active" readings. There is also a button to manually terminate a session, but in

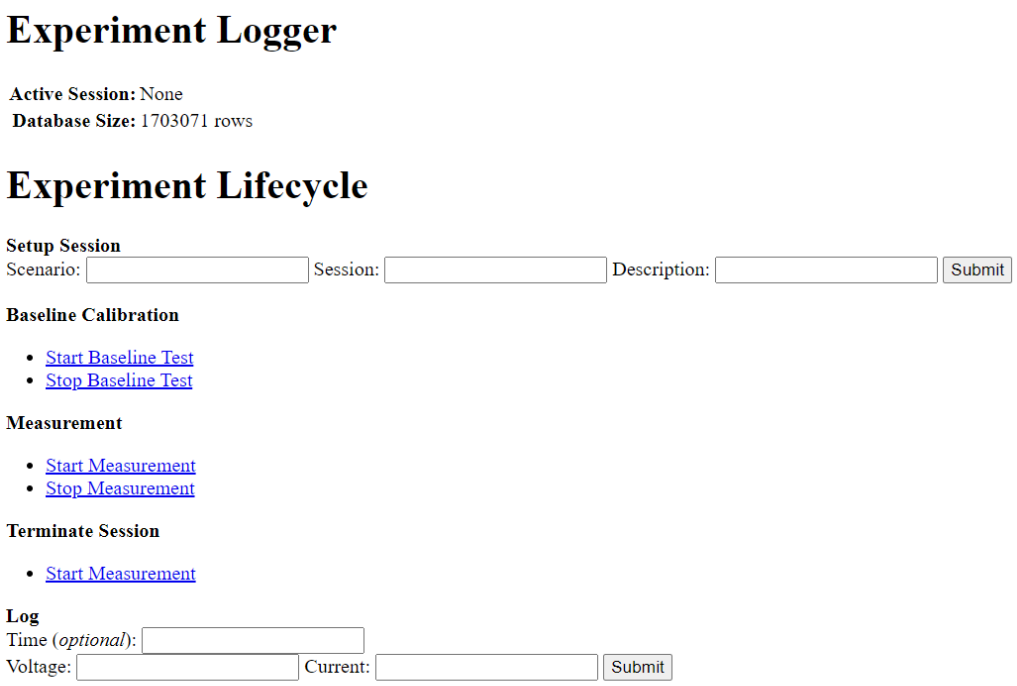


Figure 5.3.6: Web interface to the Experiment Logger

this version of the prototype this button is mis-labelled. In addition to the buttons that add data into the `session2` table, there is also a form to manually enter some dummy readings for test purposes. This form requires voltage and current values and an optional timestamp. If no timestamp is provided, the current time is used.

As with the other services, the LOG web interface provides general administration functions to truncate or rebuild the `log` table and to shutdown the LOG service.

5.4 Validation of the Design

The prototype apparatus consists of multiple collaborating parts. To verify the correct operation of the apparatus it was first necessary to validate the individual parts.

5.4.1 Voltage and Current measurement

To verify that the values read from the INA260 chip correspond to the actual voltage and current values, the apparatus was powered from a calibrated bench power supply with a voltage and current display. A range of different software was run on the DUT device, and a variety of different USB devices attached, while displaying the readings from the INA260. The INA260 has a sensitivity of 1.25mV/bit and 1.25mA/bit and typically reports results to a precision of 10mA and 10mV. The bench power supply, however, only reported voltage and current to a precision of 100mV and 100mA. For early measurements while constructing the apparatus and before the adoption of a stabilised bench power supply, voltage stability was also checked with a manually-operated digital multimeter. [Figure 5.4.1]

Figure 5.4.1: Voltage verification using a multimeter

The Raspberry Pi 4 uses a MaxLinear MX7704 power management chip that can accept voltages between 4.0-5.5V (MaxLinear, 2018). The Raspberry Pi 4 hardware includes an internal voltage monitor that reports a low-power error if the voltage drops to 4.7V. Also included in the hardware is a voltage limit to comply with the USB maximum voltage of 5.25V. The result of this is that the operational voltage of a working Raspberry Pi 4 should always be between 4.7V

and 5.25V. The bench power supply was set to regulate the output voltage to 5.0V which, given the precision of the display, can be treated as between 4.9V-5.1V.

The readings from the INA260 correctly reported the 5V input when the DUT was under no load but also reported a minor drop in voltage to around 4.9V when heavily loaded. The bench power supply was set to display but not regulate the input current. Readings from the INA260 remained within 0.1A of the readings from the bench power supply

In both the voltage and current cases the readings from the INA260 were consistent and repeatable, so the measurement approach was deemed appropriate for comparing the energy use of alternative software options.

5.4.2 Data Logging from the DUT

To validate that the readings from the INA260 were being logged correctly, it was needed to show that the power measurement circuitry will generate different readings when different software is in operation. The same approach as Kaup, Gottschling, and Hausheer (2014) was taken, by creating a simple "infinite loop" that could be run from a terminal session connected to the DUT machine. With the CTRL and LOG software running, the measurements on the Grafana dashboard, which had been set to monitor the data in the LOG database, were observed when the loop was not running and when it was running. There was a clear difference, which can be seen in Figure 5.4.2. The readings are not consistent in this graph, and exhibit some "glitches". This is due to the manual nature of the tests. For example, the drop in energy usage at around 16:48 shows the point at which the loop code was changed so that it no longer "printed" a message to the (remote) console every cycle. This shows an overall reduction in energy use by both the CPU and the network hardware. Note also that the peak power usage is around 2.5W (or 2.4W without the "print") for this run. Even though the code was running an infinite loop, which should cause the processor to work as hard as it can, the peak power usage is less than some of the readings from real application tests in Section 5.5.1. A hypothesis is that this occurs because the loop used in this test is only exercising a single core of the CPU, as observed by Basmadjian and De Meer (2012). The applications investigated in Section 5.5.1 typically run multiple threads of execution which can make use of multiple cores.

Figure 5.4.2: Initial manual measurement run

5.4.3 Interaction Between Apparatus Devices

When the apparatus starts up, each of the main test devices needs to register itself with the REGISTRY in order for other devices to be able to locate and contact. A self-updating status display was added to the REGISTRY web interface to show which devices had registered and the time remaining on their leases. To check the lease renewal behaviour the initial lease time was set to a few minutes and the devices were observed registering and renewing leases correctly. Buttons were also added to the REGISTRY status display to deregister, remove, and shutdown individual devices to ensure that the rest of the devices would recognise and adapt to the situation, preventing measurement sessions until all the required participants were present and operational.

During a test, the apparatus was designed to operate as an interaction between the various devices according to the sequence diagram shown in Figure 5.3.3. To ensure that the interactions progressed correctly from the start to the end of a test run a combination of approaches was used. Status fields and tables were placed on the CTRL web interface to show the progress of the interaction, and to override the status if, for example, a test script had an error.

5.4.4 Reliability and System Management

The apparatus was designed to be reliable in operation. The REGISTRY mechanism allows a device to be added to or removed from the set of collaborating devices without needing to modify any of the other devices. All the raspberry Pi devices run from SD cards based on the same "golden" image. The SD card image allows any of the boards to perform any of the roles, with the only difference being a single setting. For self-diagnostic purposes, if a Raspberry Pi board is started running the CTRL software but without a connected INA260 power measurement board, it loads an emulator that reports predefined values. This enables the interaction between devices to be validated even if the INA250 board breaks or becomes disconnected.

In the case of a sudden general power loss, all participating devices will shut down. This situation has been known to cause corruption of the SD card storage in some cases. All that is required to get the system up and running after such an error is to replace any corrupted SD cards with another copy of the "golden" image, configure the desired roles, and boot the devices.

Logged data is stored on more reliable hard disc media using the *PostgreSQL* database. Logged data is regularly exported and backed up to a separate machine

so that, if required, it can be re-imported to a new database for analysis. The LOG server runs in a virtual machine (see Section 3.6) which is also backed up externally. Recreating a LOG server just requires starting the backed up virtual machine image and entering the command to start the LOG server software.

Other components of the system such as the power supply and network switch are generic components. Replacements can easily be sourced. The whole apparatus is designed to be cheap and easy to acquire parts and build, if multiple instances of the apparatus are required.

5.4.5 Comparison with Other Approaches

As introduced in Section 2.10, many different approaches have been taken to estimate the energy use of software systems, and computer systems in general. For this discussion the key distinction is between approaches which use specialist hardware of some sort, and those which use only the essential hardware of the computer system and rely on software estimates.

Software-only approaches to energy usage estimation make use of system software which provides estimates of CPU usage (typically calculated by comparing the time spent in an *idle* routine compared to the time spent doing "useful work"), memory usage (typically calculated by examining an internal memory page table), or communication overhead (typically calculated by calculating the time spent executing communication routines or hardware drivers).

Such approaches are generally less valuable than hardware measurements. Depending on the power-management capabilities built into the the operating system and CPU, the system may actually consume as much energy while running the *idle* routine as it does when performing other tasks (Kansal and Zhao, 2008). Similarly, memory can continue to consume energy, even when not in use by application software, and time spent communicating is only indirectly related to the energy used by the communication hardware itself.

That said, if no energy usage measurement hardware of any sort is available, then such software can provide at least a baseline indication of the "busyness" of a system and can help to indicate which aspects of the software might be worth addressing to reduce overall energy usage. When used in combination with hardware measurement (see Section 5.4.6), it can help identify any relationship between these software estimates and the energy usage reported by the hardware measurements.

The prototype apparatus fits into the category of hardware measurements, but

has several key advantages over other hardware approaches to software energy measurement from the literature.

- The prototype apparatus is easy to construct from low-cost, easily available components. This makes it more attractive in budget-constrained situations than approaches based on expensive lab measurement systems or which require specialist servers with built-in power measurement.
- The prototype apparatus makes and logs multiple automated voltage and current measurements during a measurement run. This makes it more accurate, repeatable, and less labour-intensive than approaches based on manual measurements.
- This apparatus measures the energy consumption of the whole DUT device. This makes it more representative of the energy usage of real systems than approaches that only measure the energy usage of some parts of the device.
- The prototype apparatus is robust and allows seamless replacement of failed devices or integration of extra LOAD devices. This makes it relatively easy to keep working compared with manually configured approaches.
- The prototype apparatus provides an external API to allow it to be integrated into the continuous integration (CI) process which an organisation is already using to test software features or performance. This makes it much more attractive to software development organisations than approaches that work in an isolated laboratory setting.

The prototype apparatus does, however have some drawbacks which are discussed in detail in Section 5.7, together with potential ways to overcome them in Section 5.8. key drawbacks include:

- The prototype apparatus uses a low-cost Raspberry Pi for the DUT device. This does not have the same processor, memory, or storage as many of the servers used to serve live web software.
- The load generation software in this version of the prototype is relatively simplistic and would need to be improved to simulate any scenarios more complex than fetching a series of web pages.
- Energy measurements become less accurate at low load levels due to system "noise". However, this is a problem also faced by alternative approaches.

5.4.6 Combination with Other Approaches

The aim of the prototype apparatus is to provide a low-cost method of energy-usage comparison that can integrate with existing development and testing processes. This aim does not preclude working in conjunction with other approaches where appropriate. Some examples are given below:

- If lab measurement equipment is available, it would be a relatively small change to the CTRL software to read voltage, current, or power measurements from an external device rather than from an attached INA260 board. The use of the prototype apparatus software could assist in integrating other measurement equipment into an existing continuous integration and test process.
- If a DUT platform is chosen which contains energy use measurement circuitry for some of the internal components, then this information could be used in conjunction with the prototype apparatus. Energy usage readings from the internal circuitry could be used to help identify which aspects of the software are responsible for an increase or decrease in overall energy usage.
- If a DUT platform is running software which can provide CPU, memory, or communication estimates, this information can be used in conjunction with the prototype apparatus to identify any relationship between these estimates and the energy usage of the system as a whole.
- When using the prototype apparatus there is generally little use for manual measurements, except to provide confidence that tests and measurements are executing correctly. This is how manual measurements were used during the development of the prototype apparatus.

5.5 Results and Analysis

5.5.1 Application Comparisons

To demonstrate that the control and data collection aspects of the prototype apparatus were functioning correctly, a selection of common web server applications were compared. As of June 2023, the three most popular web

servers were *NginX*[8], *Apache*[9], and *Cloudflare*[10], together serving over 50% of the world's websites (Netcraft, 2023). *NginX* and *Apache* are open source software and were easily installed on the DUT. *Cloudflare* is proprietary software, and was not available for download and installation, but it is apparently based on the *NginX* codebase, so this was eliminated from the comparison.

Apache and NginX are general purpose web servers that can both serve pre-generated web pages from file and be configured to execute user code to respond to HTTP requests. Two other general purpose web servers were also evaluated: *Lighttpd*[11] and *Caddy*[12]. *Lighttpd* is known for its speed (Bogus, 2008), but there was no information as to whether that would translate to lower energy usage. *Caddy* is known for its extensibility, and has been used for a range of academic projects (O'Callaghan, 2017) (Ardi, Hussain, and Schwab, 2021).

In some applications, a service requires dynamic generation of HTTP responses, for example a page containing information specific to the requesting user or providing an API that returns data calculated or retrieved from elsewhere. In such cases there are two broad approaches. One approach is to use a general purpose web server to handle the HTTP protocol and pass the request data on to the application code that acts as some form of "plugin" or "extension" to the web server. The other approach is to dispense with the general purpose web server and instead for the application to handle the details of the HTTP protocol itself. To achieve this it is common to "embed" a specialist web server component into the application.

The energy usage of some well-known web server components was investigated, using each one to build a simple file-only server. Two programming languages were used that are popular for this kind of stand-alone web application development: *Java*[13] and *Node.js*[14]. With *Java*, servers were built using both the web server code built in to the standard java libraries and a third-party web server component, *Jetty*[15]. For *Node.js*, a server was built using the standard *Express*[16] library. For the standard *Java* and *Express* applications, two versions were built, one that always serves the web content from files, and one that caches frequently requested data in memory. In Figure 5.5.1, the caching

[8] `https://www.nginx.com/`
[9] `https://httpd.apache.org/`
[10] `https://www.cloudflare.com/`
[11] `https://www.lighttpd.net/`
[12] `https://caddyserver.com/`
[13] `https://www.java.com/`
[14] `https://nodejs.org/en`
[15] `https://eclipse.dev/jetty/`
[16] `https://expressjs.com/`

implementations are marked with `(*)`

To compare the energy usage of the different web servers an example website was created with just one page. That page contains text and HTML markup as well as external images and CSS styling in a manner representative of a typical public "blog" page, so it requires multiple HTTP requests to fully render the resulting page. The *Siege* load-generation software correctly recognises and fetches these external files to simulate a request from a real web browser. Before starting automated testing, each server was checked by requesting the example page using *Chrome*[17], the web browser with the largest current market share (Statista, 2023). The resulting rendered page was observed to visually check that all page assets were correctly located and rendered. Where necessary, server configurations (in the case of the general purpose servers) and code (in the case of the server components) were adjusted until a successful page was achieved.

To determine the energy usage impact of an application running on a general-purpose web server, *WordPress*[18] was installed into each of the general-purpose web servers. *WordPress* is a "content management" application (Patel, Rathod, and Parikh, 2011) estimated to run approximately 50% of the world's websites (W3Techs, 2022). The static files and the *WordPress* application were configured to serve identical web pages.

Once there was confidence that each web server was configured and working correctly, the automated energy usage measurements began. Each server was subjected to a load of 100 page requests while measuring the power used by the DUT. Each experiment was repeated several times to determine a range of energy usage and to reduce the impact of accidental or unrelated activity during measurement. For each experiment, the mean power usage in Watts was multiplied by the length of time it took for the application to handle the load in seconds to give a total energy usage in Joules. The results, showing ranges and mean values, are summarised in Figure 5.5.1.

In addition to measuring the energy usage during each experiment, the time it took for each of the servers to complete serving the requests from LOAD was also tracked. The results, laid out in the same framework as Figure 5.5.1, are shown in Figure 5.5.2.

[17] `https://www.google.com/intl/en_uk/chrome/`
[18] `https://wordpress.org/`

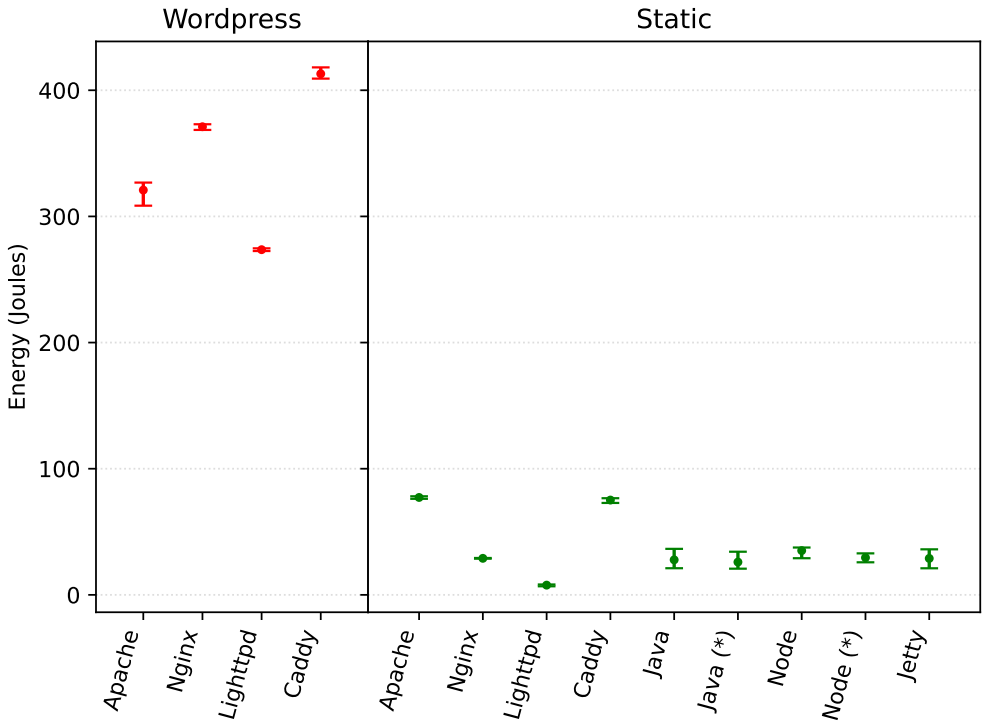


Figure 5.5.1: Energy usage grouped by server

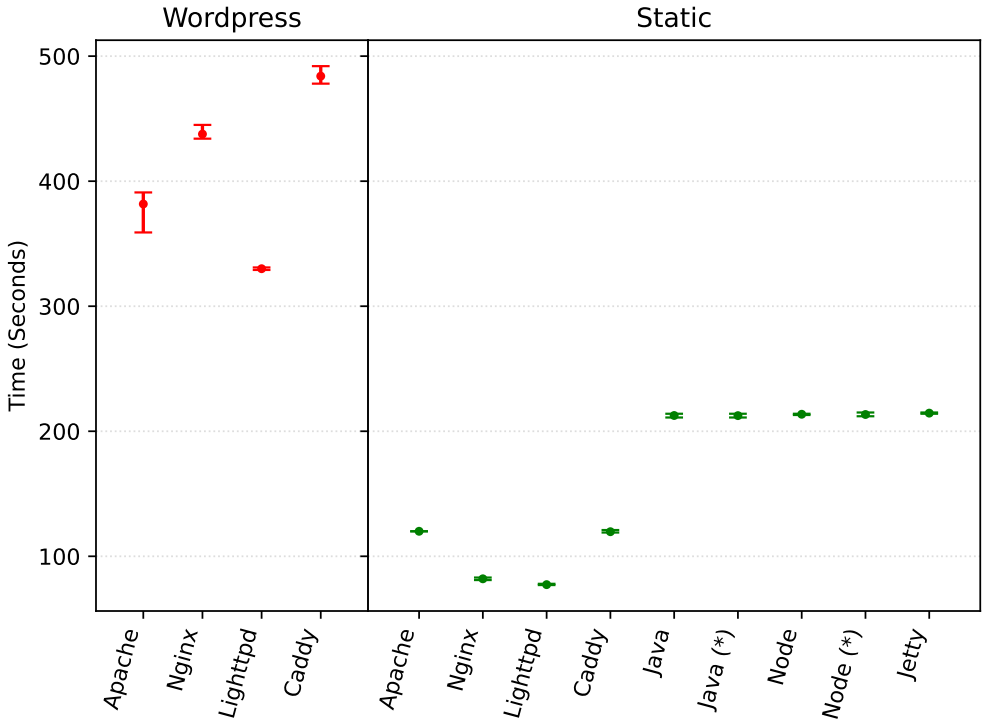


Figure 5.5.2: Experiment duration grouped by server

5.5.2 Analysis

The most obvious observation from these results is that, in every case, WordPress consumes a lot more energy to serve the same website than any of the static servers. The median energy usage for all the WordPress experiments was 347J, while the median usage for the static servers was just 29J. That's roughly 12 times the energy to perform the same task. Comparing the median values does not tell the whole story, though. The purpose of this apparatus is to enable decisions to be made between alternate options with the aim of reducing overall energy usage. Comparing the "worst" approach in this group (*WordPress* running on *Caddy* with a mean energy usage of 413J) to the "best" (*Lighttpd* serving static files with a mean energy usage of 12J) shows that *WordPress* on *Caddy* uses over 33 times more energy for this scenario.

The extra energy used by the *WordPress* servers is due to a combination of several factors. *WordPress* is written in the *PHP*[19] programming language, known to consume more energy than the compiled languages typically used to write efficient servers (Pereira et al., 2017) (Pereira et al., 2021), and that does not run natively in any of the servers investigated. Once an incoming HTTP request is received and decoded by the host server it must then be passed on, using one of a variety of methods, to a separate process responsible for running the *PHP* code. When the *PHP* code has completed processing the request, the response must be passed back to the host server and then returned to the client. All this communication takes resources. *PHP* is a language that sits within HTML markup. To prepare a response, the *PHP* processor must parse the combined file, execute any contained PHP code, which may in turn require the loading and execution of other *PHP* files, and then combine the results and the original markup into a final web page. Depending on the configuration of the host server, this *PHP* code may need to be fetched from files and re-parsed every time, or some or all of it may remain in memory. All this page loading and processing also consumes energy. In the particular case of *WordPress*, much of the page content is stored in a database. To generate a response page requires not only all the processing described above, but also one or more requests to a database, all of which consumes yet more energy.

In contrast, the process for serving static websites is much simpler. A request is decoded and resolved to a filename. That file is then loaded, either from storage or from a memory cache, and returned as-is to the client. The experimental results show that this process clearly takes less energy than generating a *WordPress* page on any server.

[19] `https://www.php.net/`

While comparing a complex process to a simple one may seem like an unfair comparison, it is important to consider this from the point of view of the end user of the website. To such a user a web page either contains useful and interesting information and facilities or it does not. The technology used "behind the scenes" is unimportant. If the same end result can be achieved by a simpler, and less energy-hungry, process then the energy used by the complex process is unnecessary waste.

Although not explored in this research, technologies do exist to mitigate some of the issues of *WordPress*. Typically these are positioned as ways to improve the response speed and throughput, but they are also likely to have an impact on energy usage. Plugins are available for WordPress that cache full or partial pages, eliminating the need for some of the database accesses and page construction (BoldGrid, 2022). However, such plugins are themselves written in *PHP* so still suffer the issues of communication overhead and relatively slow, energy-intensive execution. Depending on the host server there may also be caching available at the request and response level, eliminating the need to invoke the *PHP* code at all for some frequent requests. Caching is also possible in the network infrastructure between the client and the server (Kusuma, Widyawan, and Ferdiana, 2017). This is a common use-case for the *Cloudflare* software mentioned in Section 5.5.1 and contributes to its position in the server rankings. Finally, there are tools that can convert some kinds of *WordPress* website to a static site. All of these techniques require some combination of extra configuration, installation of plugins (Data, Luthfi, and Yahya, 2017), and costs (Strattic, 2023), which limits their uptake.

As mentioned in Section 5.5.1, *Apache* and *NginX* are currently the most used servers for general websites. Between these two the situation is more complex. When running *WordPress*, the two are roughly similar. *Apache* shows a lower mean energy usage, but a wider range. When serving static websites, *NginX* is a clear winner with *Apache* using roughly 2.7 times as much energy on average. Usage of *NginX* is currently increasing, largely at the expense of Apache (Netcraft, 2023).

Of the four general-purpose web servers examined, the lowest energy usage in both static and *WordPress* scenarios was *Lighttpd*, averaging 286J for the *WordPress* runs and 12J for the static website. This appears to validate the claim that *Lighttpd* is faster, and also the assumption that this translates into using less energy to process the same load. Conversely, the worst server was *Caddy*. Using roughly 6 times as much energy as *Lighttpd* for the static scenarios and roughly 1.5 times as much for the *WordPress* scenarios. *Caddy* may be more extensible, but that comes at a cost.

The component-based servers all showed broadly similar energy consumption to NginX when used for static files. Not as good as *Lighttpd*, which has obviously been highly tuned, but better than *Apache* and *Caddy*. There is a slight indication that caching frequently-used pages in memory improved the energy consumption of the *Node.js* server, but had little effect for the *Java* server. Further testing with a broader range of page requests would be needed to see if this holds for more realistic scenarios.

Of particular interest were the similarities and differences between the energy usage and speed of the different servers. Software performance, in the sense of speed to complete a task or throughput for non-completing services, is much more commonly measured than energy usage (Freitas and Vieira, 2014) (Küber, 2023). In software the term *efficiency* is almost universally considered in performance terms. Several researchers have attempted to determine a mathematical relationship between time performance and energy usage (Kalaitzoglou, Bruntink, and Visser, 2014) (Stoico et al., 2023), in the hope of eliminating the need for external energy measurement apparatus.

In both aspects the *WordPress* servers were considerably worse than the static servers, with median duration of 412 and 213 seconds, respectively. The difference is not as great as the energy usage figures, however. The *WordPress* servers show a similar pattern of better and worse. This might be taken to imply that experiment duration is a reasonable proxy for energy usage, at least for a simple "bigger or smaller" comparison. This relationship does not hold for the static servers, though. The component servers, which consumed roughly the same energy as *NginX*, and less than *Apache* and *Caddy*, take over twice the time to service the same requests as *NginX* and are considerably slower than both *Apache* and *Caddy*. Even within the general-purpose servers the relationship between performance and energy usage does not hold well. *NginX* uses 3-4 times as much energy as *Lighttpd* for roughly similar performance.

The apparent similarity between the performance of the component servers, despite being written in different programming languages and using a range of approaches, is an option for further research (see Section 7.4). There may be some underlying factor that is constraining their performance but does not apply to the general-purpose servers.

5.6 Evaluation Against Requirements

In Section 5.2, a set of requirements were introduced for the apparatus described in this chapter. This section evaluates the prototype apparatus against these requirements.

5.6.1 Measure the energy consumption of the whole system, not just certain components

The prototype apparatus met this requirement. The use of the INA260 voltage and current measurement chip in the 5V power line to the DUT device ensures that all hardware and activity on the DUT device is measured. In this aspect, the prototype apparatus is superior to energy measurement techniques that only measure processor and/or memory energy use.

5.6.2 Support testing of existing software without manual modification or substantive changes

The prototype apparatus met this requirement, at least for software that will run on the Raspberry Pi device chosen for the DUT role. Popular software such as the *Apache* and *Nginx* web servers and the *WordPress* website management application were installed and run on the apparatus without changes.

However, this apparatus is not capable of running *all* potential software, as some software requires different hardware with features such as a specific type of processor, more processor cores, or more memory. See Section 5.7 for discussion of the hardware limitations of the apparatus.

5.6.3 Measure energy consumption during representative usage

The prototype apparatus met this requirement for web applications driven by HTTP requests from web users. Software running on the DUT device is unaware that HTTP traffic is being generated by the LOAD device rather than from the "real" web. The *Siege* load testing tool selected for use on the LOAD device generates HTTP traffic that is largely representative of the requests made by a web browser including fetching images, styles, and script files.

For applications with different usage patterns or requirements, the REGISTRY software is flexible enough to allow multiple or alternative LOAD devices to be

registered. These alternative LOAD devices do not have to be Raspberry Pi devices built into the apparatus hardware, but can be any kind of device that is capable of sending HTTP requests to the REGISTRY and DUT, and of receiving commands from CTRL. the prototype apparatus supports LOAD devices running whatever load simulation software is desired. These kinds of load simulation are often performed during existing load tests or performance tests, so the prototype apparatus supports energy measurement during whatever load and performance tests would normally be run.

The prototype apparatus may not provide accurate energy comparisons during tests in which the testing code runs on the same DUT device as the code being tested. This is a fairly common pattern for component performance testing (as seen in Chapter 4) but when used on the prototype apparatus the energy usage of both the code being tested and the code running the tests will be combined into a single result. If the energy used by the code running the tests is consistent between tests, however, then the difference in energy usage between tests will still be representative of the difference in energy usage between the components being tested.

5.6.4 Be precise enough and fast enough to catch transient events

The prototype apparatus met this requirement to a large degree. Current and voltage measurement is performed much more frequently than is possible with manual measurement or with most domestic "energy meter" devices. Experimental readings were able to demonstrate capturing energy spikes during software execution. However, the nature of using an operating system on the DUT device powerful enough to run real web server software means that there are also other processes running in addition to the code being tested, and these can also generate transient changes in energy usage.

5.6.5 Be programmable so it can be used in a CI build process

The prototype apparatus met this requirement. Once the basic apparatus software has been installed on the CTRL, REGISTRY, and LOG devices, operation of the prototype apparatus is completely programmable. The apparatus provides access for external CI scripts to install, start, stop, remove, or update software on the DUT device and a HTTP API to trigger measurement runs and monitor test status. Once each measurement run is complete, test data remains available from the LOG database for analysis even after the software on the DUT and/or LOAD devices has been reconfigured the next measurement run.

5.6.6 Be able to operate repeatedly without human intervention

The prototype apparatus met this requirement. Once the basic apparatus software has been installed on the CTRL, DUT, LOAD, REGISTRY, and LOG devices, operation of the prototype apparatus requires no human intervention, except in cases where a hardware, network, or operating system error has occurred. Although each of the devices in the apparatus provides a Web UI for manual operation, this is primarily for diagnostic tasks such as validation of test scripts and is not necessary for normal operation of the prototype apparatus.

5.6.7 Be easy to use

Ease of use is a classic example of a "soft" requirement (see Section 3.3.1) that can only be evaluated subjectively. Once set up to integrate with a particular CI process, use should be trivially easy, typically involving just *pushing* a new version of the software to be tested to a version control repository. This then triggers the rest of the steps: installing of the software to be tested onto the DUT machine, setting up the appropriate load test scripts on the LOAD machine, then triggering a measurement run. This process will be repeated automatically for each new version of the software to be tested.even if this is several times per day.

Setting up the steps to be performed by the CI process will probably be relatively familiar to anyone with experience installing and using the software to be tested. Installation of the software on the DUT device can be done either using manually created scripts or using a deployment tool such as *Ansible*, *Chef*, *Puppet*, or *Terraform*. Setting up load generation software on the LOAD machine can use any scripts or load generation tools that would normally be used to test the function or the performance of the software. Once these CI steps have been set up, they can then be used automatically by the CI process for each new test.

It is expected that the apparatus will normally be used under the control of CI build process, but manual operation is also possible. Typically this will be used when creating or modifying installation and test scripts to check that everything is working as expected. Each of the devices in the apparatus provides a simple web user interface (see Section 5.3.10). A typical working session might start by accessing the REGISTRY server interface using a web browser. The address of this device is fixed, so can be placed in a bookmark, for example. The web interface for the REGISTRY server provides web interface links and status indication for all the participating devices, enabling access to their web interfaces in a single click. To run a measurement set for software that is already set up on the DUT

and LOAD it is merely necessary to click the link to navigate to the CTRL web interface, enter a unique scenario name and session number (used to identify the results associated with this run) and an optional description, then click the submit button. The complete test run will then be carried out automatically and the results stored in the LOG database for analysis.

5.7 Limitations and Drawbacks

The apparatus and software described in this chapter have been developed as a prototype to investigate the feasibility of a low-cost way to include energy usage comparison in the normal process of software acquisition and development. However, this system differs in several key areas from the platforms in common use in commercial servers.

5.7.1 Processor

Raspberry Pi computers use 32 or 64-bit ARM *Reduced Instruction Set Computer* (RISC) processors. While computers using such processors are becoming more popular in datacenters, they are still in a minority (Korolov, 2022), with most servers using *Complex Instruction Set Computer* (CISC) processors from Intel or AMD, even though switching to RISC processors could potentially provide equivalent performance at a reduced energy consumption (Varghese et al., 2015). The Raspberry Pi 4 used in this prototype has 4 CPU cores. This is the maximum for current Raspberry Pi devices, but Intel processors, for example, are available with tens (Intel, 2022) or even hundreds (Intel, 2023) of cores. Software that can make use of more CPU cores could exhibit a different energy usage profile when run on such processors (Basmadjian and De Meer, 2012).

ARM processors are designed around a RISC model with relatively few processor instructions compared to equivalent designs from Intel and AMD. Specific implementations of the ARM processor core may have different amounts and speeds of local instruction and data cache and may include or omit features such as instruction look-ahead. These differences can affect both manual and automated code optimization strategies, and in turn may affect the performance of programming languages and applications (Hartley et al., 2022).

5.7.2 Storage

The initial choice of the Raspberry Pi Foundation to use SD cards as the sole form of persistent storage was unusual. With the exception of a few niche cases (Raspberry Hosting, 2023), commercial servers use spinning hard drives or solid-state SSD or NVMe drives. SD cards are known for being slow in comparison with other forms of storage. Typical SD cards range from 5-15MB/s write speed compared to 160MB/s for a typical hard drive, 550MB/s for SSD, and 5000MB/s or more for NVMe (Tekie.com, 2023). Software that makes a lot of storage access may become *IO bound* - unable to process data at full speed because it is waiting for the storage device - and therefore exhibit a different energy usage profile. SD cards are also known to be less reliable than the other forms, with some Raspberry Pi hosting companies allowing only network storage rather than support the risk of SD card failure (Mythic Beasts, 2023)

There are ways of connecting other forms of storage to the Raspberry Pi. Every Raspberry Pi has USB ports, and on the Raspberry Pi 4 two of those are USB3 and therefore theoretically capable of up to 5GB/s. In practice the available speed will depend on both the type of drive and the protocols supported by the USB drive controller, as well as the internal architecture of the Raspberry Pi 4 which shares a single 4GB/s PCIe lane between all the USB ports. Network storage is also a possibility. Modern Raspberry Pi devices can be configured to boot and operate entirely from network storage (Raspberry Pi Foundation, 2023a). As this apparatus is intended for network-driven stress testing, It was considered that the extra network traffic associated with storage access might affect the performance and thus the power usage readings. Both USB storage and network storage have a bigger problem for this application, though. Many USB hard drives and network storage devices require more power than the Raspberry Pi is able to provide, which means that they would need an external power source. This would exclude any power usage associated with storage from the results and act against the intention to perform a holistic measurement.

Possible ways around this limitation are discussed in Section 5.8.

5.7.3 Memory

The Raspberry Pi is an all-in-one single-board computer. The amount of memory is fixed at time of manufacture and the current maximum is 8GB. While this is enough for simple or undemanding applications it is not representative of many real systems. Commercial servers are commonly available with as much as 256GB of memory (FastHosts, 2023). Any software that requires more than the available

memory on the Raspberry Pi board will either fail to operate or suffer changes to its performance and energy usage.

5.7.4 Apparatus software

The software used both for running the application being tested and for providing representative load are unlikely to be identical to those found "in the wild". The current approach to simulating usage is simplistic and repetitive and therefore does not execute a wide variety of code and data in the application being measured. This has the specific side effect that applications which cache frequently requested data can quickly "warm up" and may exhibit different performance and power usage than if they were receiving a broader and less predictable range of requests. Note however, that the LOAD software makes use of *Siege*[20], an open source load tool of the type commonly used for performance and functionality testing during development, so this problem is not unique to this apparatus.

The software for starting. stopping, and measuring idle power consumption of applications is also relatively simplistic. Applications to be tested are started by shell scripts that may not accurately match the real-world installation and start-up process. Applications are left running only for the duration of their measurement cycle, which may mask longer-term caching, secondary compilation, and other performance strategies. Supporting services such as databases may or may not continue running after a measurement run is complete, depending on how they have been installed, and this may affect the measurement of idle power consumption and even possibly the active power consumption of later applications.

5.7.5 Reflections

It is clear, for all the reasons mentioned above, that the prototype apparatus is far from identical to a typical web server used for real software deployments. Whether it is representative enough to be useful is a different question. The aim of the apparatus is not to provide an accurate *absolute* measurement of the energy used by a particular installation of software. Instead, the aim is to enable *comparisons* between different pieces of software. Such comparisons are useful both when choosing between candidate software components or applications, and when comparing different versions of the same software as part of development. In either case, the apparatus provides a relatively consistent environment for such comparison. A similar approach is commonly used when testing software

[20] `https://github.com/JoeDog/siege`

functionality and performance during development (Collins and De Lucena, 2012). Such tests are performed in a "test environment" or "development environment", which may be smaller or otherwise different from the live deployment.

It is suggested that if software A runs correctly and uses less energy than software B for a similar task when measured on this apparatus, it can be taken as an indication that it is likely to use less energy when deployed live. Such information is potentially very useful and otherwise hard to obtain.

5.8 Improvements and Future Possibilities

5.8.1 Addressing Limitations

A prototype is not a finished product, and exists as much to discover limitations and provoke suggestions for improvements as it does to validate an idea. As can be seen from Section 5.7, this prototype apparatus has several aspects in which it may not be fully representative of a real deployment environment.

Some limitations of the prototype could be partially mitigated without major changes. For example, the speed and reliability issues associated with SD cards could be improved by sourcing high-speed USB storage with low enough power requirements that it can operate as the primary storage for the DUT and LOAD devices. Likewise, the issue of a simplistic and potentially unrepresentative pattern of requests from LOAD to DUT could be addressed by finding or creating a more sophisticated load service.

Other limitations would require larger changes to the hardware of the apparatus. The largest change would be to replace the Raspberry Pi with an alternate platform that addresses some or all of the limitations of that device. Several manufacturers now produce equivalent single-board computers with a range of memory and storage options. Most also use ARM processors, but some are available with Intel, AMD, or Risc-V processors instead. In most cases, features such as larger memory, more CPU cores, or on-board SSD or NVMe storage also increase the power consumption and price of these devices. After this research took place, a fifth iteration of the Raspberry Pi was released that includes a relatively high-speed PCI interface. Several after-market adapter boards are now available that allow the connection of NVMe storage, which is more representative of common server deployments.

The popularity of USB Power Delivery (PD) for charging mobile devices has also led to an uptake in the use of USB PD for general computer use. An increasing

number of laptop and "mini PC" devices now use this standard for operational and battery charging power. Replacing the DUT with such a device might be physically cumbersome, but should pose no major software or operational issues. The prototype apparatus is capable of measuring power usage of any device using USB PD up to version 3.0. However, version 3.1, introduced in 2021, allows voltages up to 48V that are beyond the range of the INA260 power measurement chip.

Any changes to the power supply to the DUT device would probably also require the separation of power supplies. In this prototype, DUT, LOAD, and CTRL are all powered by a single stable 5V supply. USB PD includes a negotiation process and is therefore only suitable for a single device at a time. If just the DUT device requires USB PD, then potentially LOAD and CTRL could still share a 5V supply.

The current prototype does not use all the features of the INA260 measurement chip. It is possible that the accuracy of the readings could be improved by, for example, using the power usage calculated by the INA260 directly rather than the existing approach of measuring voltage and current and combining them to derive the power.

To support DUT devices requiring higher DC voltage and current, or even an AC supply, would require a different power measurement technology. Integrated circuits, boards, and full devices are available from a variety of manufacturers, but these would represent a major change to the prototype apparatus and would also potentially increase its cost.

5.8.2 Software Improvements

As well as addressing the differences between environments, there are other potential software changes that could improve the usability and effectiveness of the prototype. For example, the need to manually adjust the SD card "disc" image for each of the Raspberry Pi devices is intrusive and error-prone. There are several potential ways to address this including: storing some form of role ID in the Raspberry Pi EEPROM so it persists between SD card changes; placing "jumpers" or other connectors on the Raspberry Pi GPIO pins that can be read on start-up; requiring a USB "key" to identify the device role; and so on. While such changes would be an improvement, they are dependent on the hardware of the raspberry Pi, and would be unlikely to work if a different device were to be used instead.

The current prototype software makes the assumption that an application running

on the DUT, but receiving no requests from LOAD, uses negligible energy. This has been observed to be the case for the small number of applications investigated so far, but arguably may not be the case for all applications. To address this oversight, there should be the option to also measure the baseline energy usage before instructing the DUT to *Warmup*.

This prototype has so far only been used with a small range of applications that were manually installed prior to measurement. *Warmup* and *Cooldown* scripts were also created by hand. This approach is not suitable for an environment requiring more frequent or varied, automated, measurements. The addition of an *install* and *uninstall* steps to the measurement process would allow, for example, a call to an external API or "webhook", which would then manage the installation without the measurement software needing to know the details. When the installation step is complete, the system can proceed with the rest of the measurement cycle, then notify the external system to undo the installation in the same way.

5.8.3 Future Possibilities

The current prototype embodies a relatively simple model of a single apparatus testing the overall energy usage of a single application at a time. While this functions as a proof of concept, there are ways in which this model could be enhanced to be easier to use, provide more precise information, and scale more effectively to larger and more complex systems.

A relatively simple enhancement would be to support multiple installations of the measurement apparatus communicating with a shared LOG service and database. This would require the provision of a unique identifier to each apparatus, and that id to be included in all messages to the database. The apparatus id would then be stored with the status updates and readings in the database, enabling analysis to be performed either on the results of a specific apparatus or aggregated across all installations. A system such as this would allow either parallel evaluation of the same measurements on multiple candidate applications or simultaneous evaluation of multiple different load patterns on a single application.

At the moment the prototype software treats all power measurements during a run as relating to a single session. The assumption is that the load client generates "realistic" load that can be used to measure the power consumption of the application under representative usage. The collected results can then be used to statistically determine values such as the total, average, and peak energy usage for the scenario. While this is arguably useful, it is not particularly fine-grained.

To measure any different scenario requires a complete test run including *warmup*, *cooldown*, baseline measurements, and potentially also installation and removal of the software being tested. An alternative approach would be to split the main phase of the measurement, once the baseline measurement has completed, into distinct sections representing different volumes or patterns of load. This could potentially speed up the process and allow a greater variety of measurements in a given time.

In agile software development, software is typically tested many times and at many levels during development, not just when an application is complete (Fowler, 2012). When evaluating potential third-party code, components, or libraries to include in an application, it would be useful to be able to evaluate the relative energy consumption of the alternatives. In both these cases, the code to be evaluated is not in the form of a whole application service which exposes an API for a load client to exercise. Instead, the system needs to evaluate fragments of software. This is a more complex challenge. One approach is to build a "wrapper" application to contain the software fragments and translate client API requests into internal software calls. This has the advantage of fitting well with the existing architecture, but for some components the overhead of the containing application could outweigh the energy usage of the components. An alternative approach is to exercise the software from within the DUT device without the overhead of processing network requests. While this could be more effective, it would require changes to the architecture to ensure that measurement readings and status notifications to the LOG database are correctly synchronised.

The current prototype apparatus was designed for low cost and ease of construction and testing. More work would be needed to produce a version which would be suitable for inclusion in commercial build processes. The hardware would need to be self-contained, robust, maintainable, and protected from accidental damage. The software would need to be flexible to support unknown future applications and usable without the need to make internal changes. The improved hardware would require sourcing of components and a casing as well as physical and electrical testing. The software changes would require improving the architecture to support configurations, plugins, or "hooks" to integrate with existing test and application infrastructure.

5.9 Conclusions

The primary purpose of the research in this chapter was to determine if the energy usage of software applications and components varies, and therefore

whether selecting a more energy-efficient combination of components would show an overall reduced energy use compared with an application constructed using less energy-efficient components. Arguably this has been shown to be the case, and the results illustrate measurable variation in the energy usage of different software applications which perform similar functions.

The energy usage results from the web server comparisons showed considerable differences in energy usage and performance between candidates. The energy usage results from experiments using *WordPress* software are considerably higher than those from a static website with a similar appearance and functionality. Further research is needed to validate these results in real deployments, but initial indications are that application energy consumption can be reduced by switching to a more energy-efficient web server application and that a large amount of energy, with its associated greenhouse gas emissions, could be saved by switching to static websites wherever possible.

The secondary purpose of the research in this chapter was to determine whether a low-cost apparatus could be constructed and used to compare the relative energy usage of different software. An apparatus was successfully constructed and used to compare a variety of software applications. A low-cost apparatus of this nature could therefore be useful both for selecting between functionally equivalent software applications, and for tracking and addressing changes in energy usage between releases of software during development.

This research also highlights the variable relationship between performance and energy usage. Although the same word "efficiency" is used in both cases, the meaning and the measurements can be very different. The amount of energy used by an application to perform a particular task depends on many factors including programming language, application architecture, and the use of libraries and components. Performance measurement, along with other estimation techniques such as static analysis, is not a reliable method of determining energy usage.

This prototype apparatus demonstrates that the concept of low-cost automated energy usage comparison is valid. It is hoped that energy usage comparison will soon become a common part of both the software acquisition and software development processes.

Chapter 6

The Energy Usage of Template Engine Components

Previous chapters have included performance comparisons of software components and the construction and testing of a prototype apparatus to compare the energy use of running software. The results showed that common server applications differ in the energy they use to perform the same task and that there is a wide variety in the performance of software components. Arguably, therefore, both the amount and capability of servers required (which depends to some degree on application performance) and the energy those servers use in operation (which depends on the software they are running) could potentially be reduced by substituting components for ones with better time- and/or energy-efficiency.

This chapter explores the energy usage of software components, in particular the template engines investigated in Chapter 4. A range of experiments are conducted to determine the energy use and its relationship with performance for a range of scenarios. The feasibility of comparing the energy use of software components is considered in Section 6.2; the experimental approach is improved in Section 6.3; and a more realistic scenario is constructed and evaluated in Section 6.4 to explore the energy and time use of template engines in context.

6.1 Performance Tests on Different Platforms

6.1.1 Method

The initial performance experiments in Chapter 4 were conducted on a standard laptop, before the construction of the prototype comparison apparatus. Changing the underlying platform is likely to change the absolute values of the performance measurements, but potentially also the relative performance. To provide context for the energy comparisons in this chapter, the final "Set 4" performance comparisons were run on the DUT platform of the energy comparison apparatus.

6.1.2 Results

The results of running measurement set 4 on the DUT platform are shown in Figure 6.1.1 to Figure 6.1.8. Note these graphs show the results of a single run without averaging. A combined set of results for the same measurement set on the Intel PC platform can be seen in Figure 4.10.23 for comparison.

6.1.3 Discussion

There are several key points to take from the comparison of template engine performance on the two platforms.

All the performance measurements run roughly 10 times the speed on the Intel PC platform compared with the Raspberry Pi-based DUT platform. This is not surprising given the higher clock speed, faster memory and storage, and greater number of processor cores on the Intel PC platform. This overall difference, in itself, does not diminish the usefulness of the comparisons. Neither of these platforms are identical to current web server platforms, and future web platforms will be different again.

The difference between the performance tests run on the two platforms is more than just scaling up, however. On both platforms the performance profiles vary according to the scenario being examined but there are also more subtle differences beyond a simple increase in speed. Consider the results from running the "separate" scenario. In particular, the performance curves of three of the template engines *JTE*, *Velocity*, and *Thymeleaf*. This scenario is of interest as it

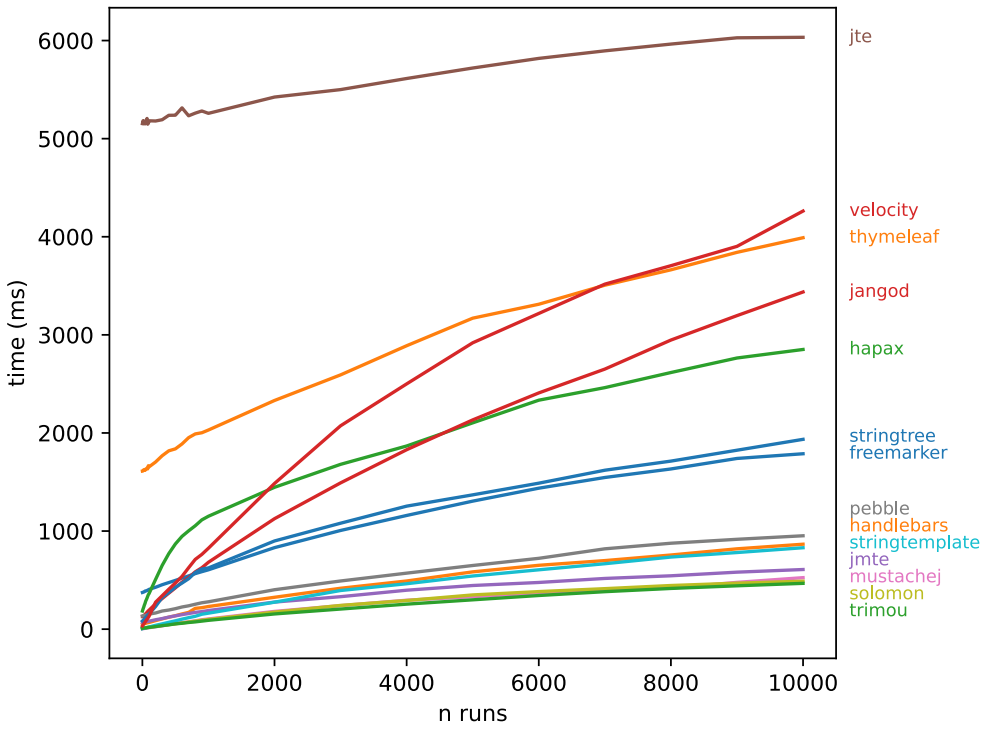


Figure 6.1.1: Set 4 performance comparison for the *plain* scenario, run on the *DUT* platform

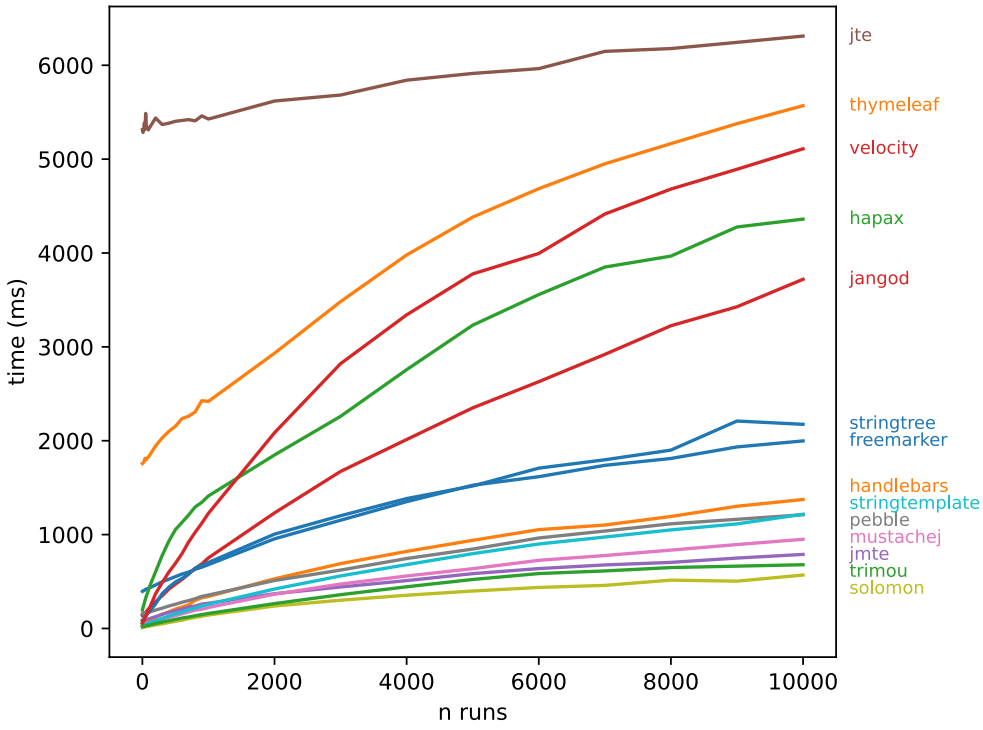


Figure 6.1.2: Set 4 performance comparison for the *single* scenario, run on the *DUT* platform

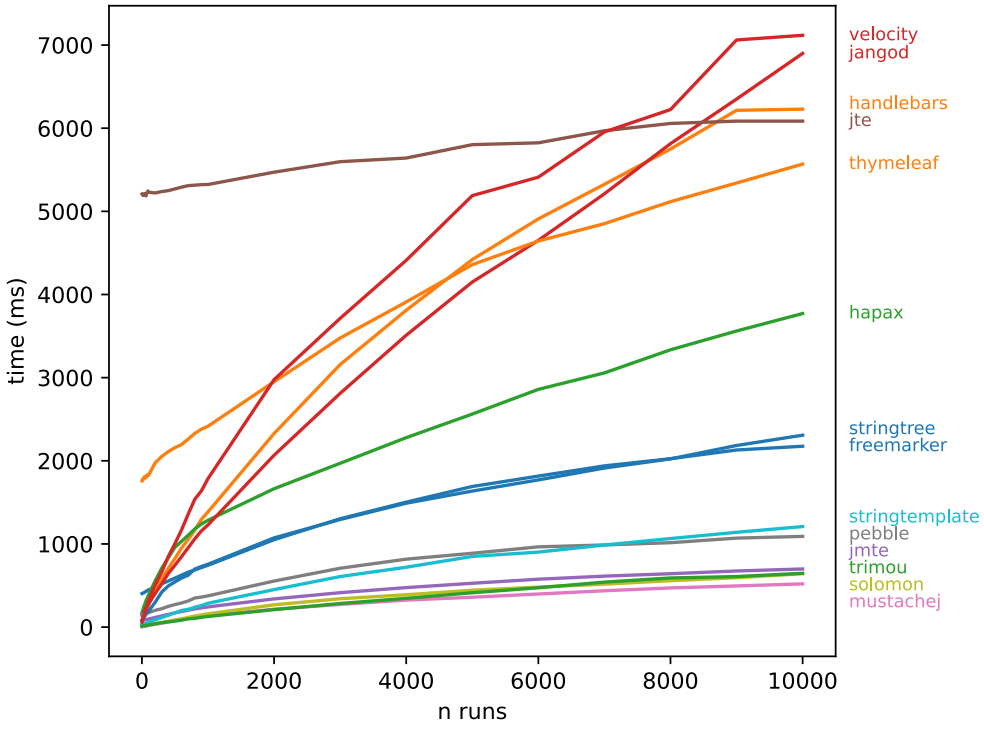


Figure 6.1.3: Set 4 performance comparison for the *include* scenario, run on the *DUT* platform

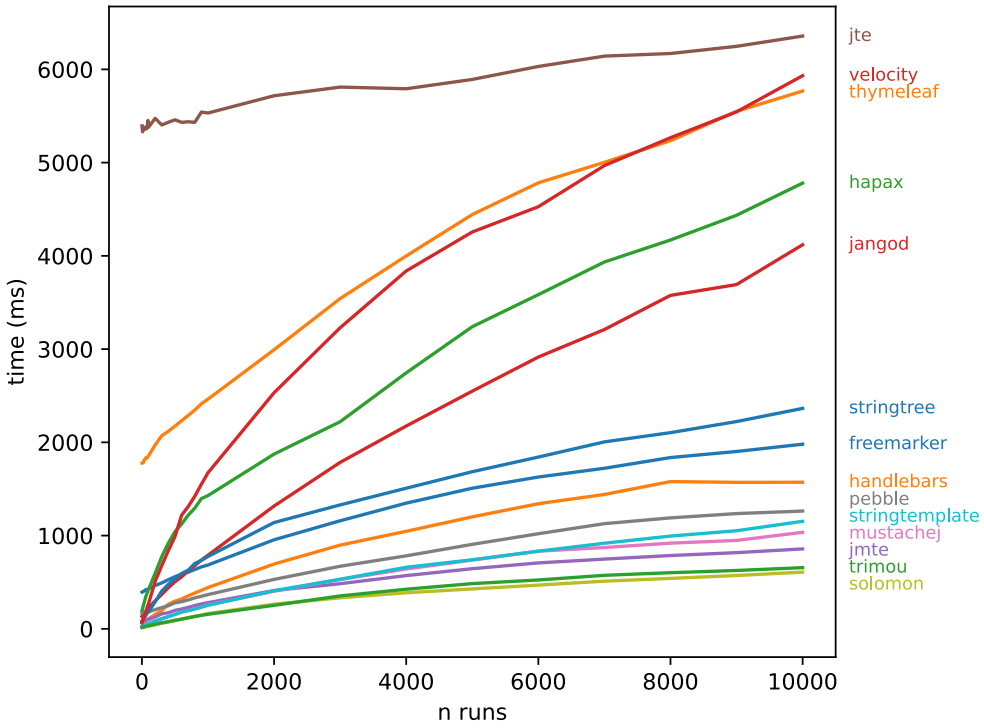


Figure 6.1.4: Set 4 performance comparison for the *cond-true* scenario, run on the *DUT* platform

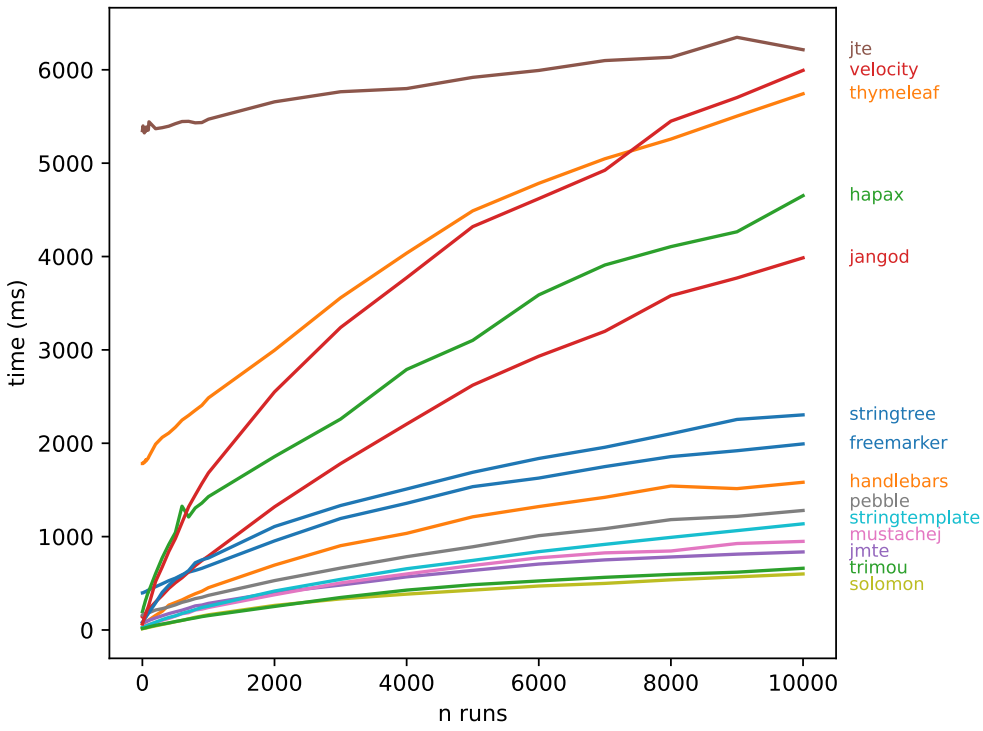


Figure 6.1.5: Set 4 performance comparison for the *cond-false* scenario, run on the *DUT* platform

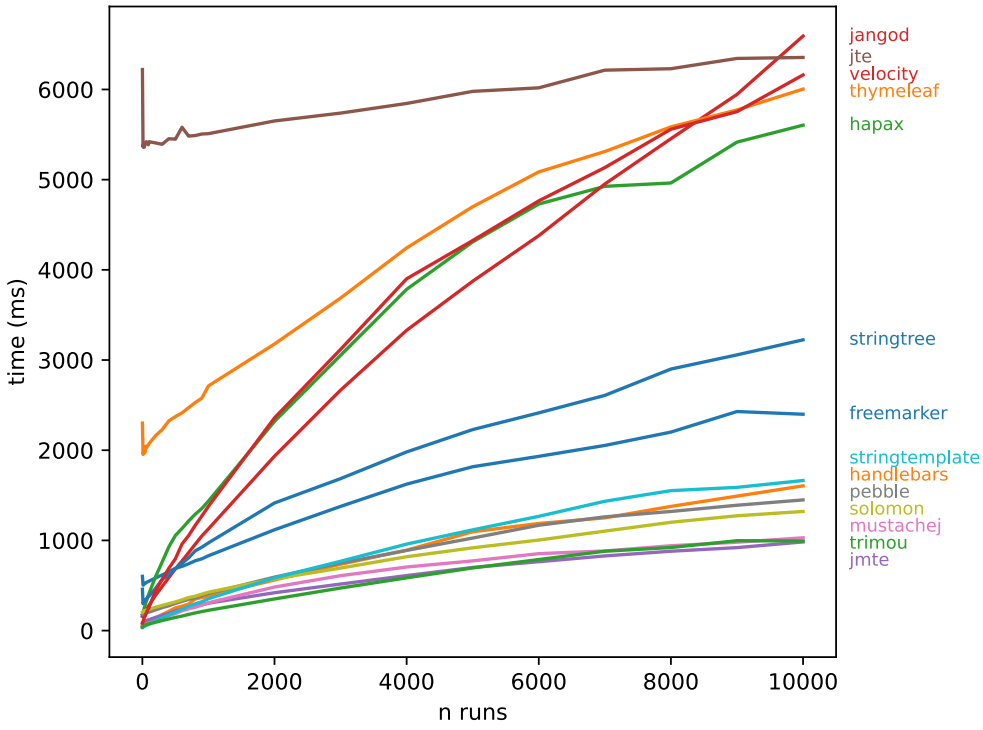


Figure 6.1.6: Set 4 performance comparison for the *bean* scenario, run on the *DUT* platform

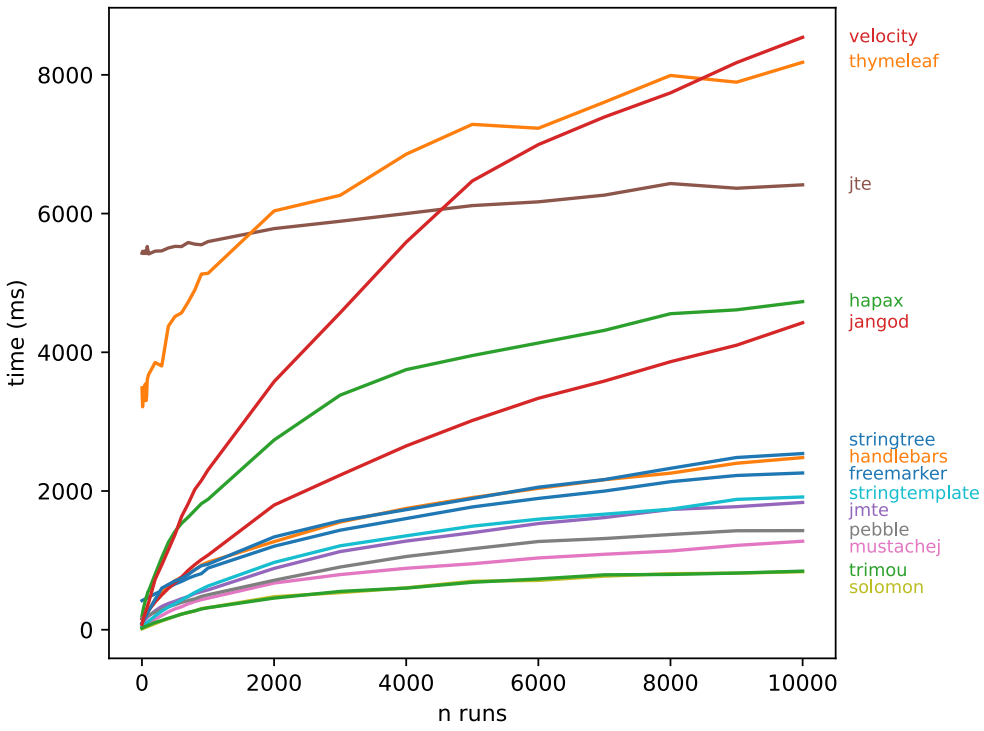


Figure 6.1.7: Set 4 performance comparison for the *iter* scenario, run on the *DUT* platform

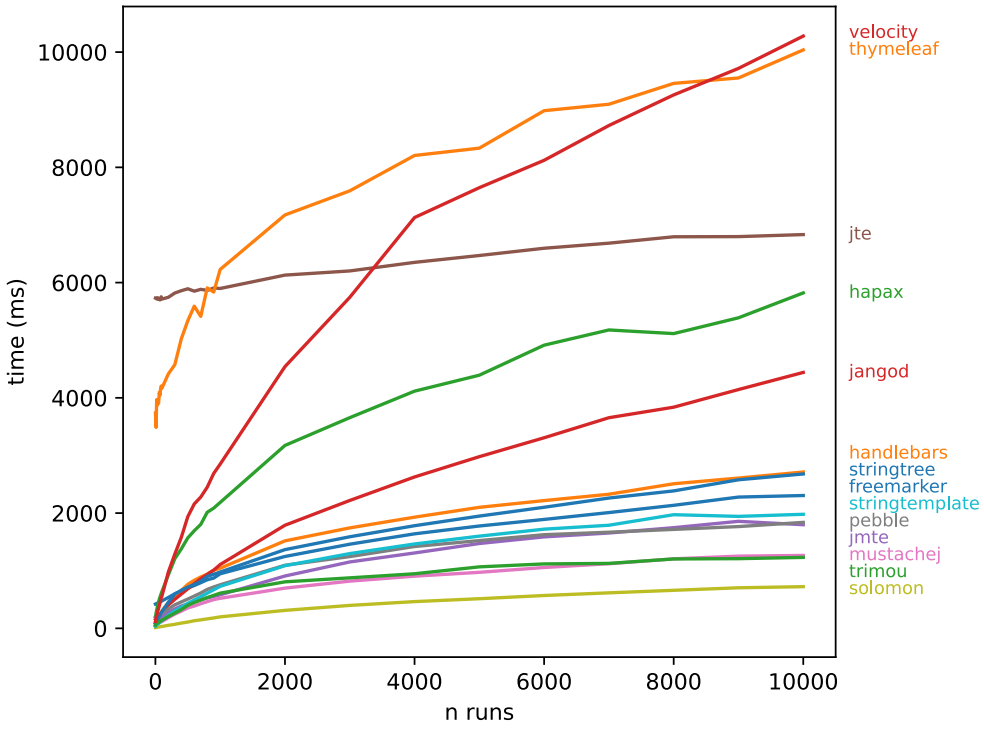


Figure 6.1.8: Set 4 performance comparison for the *separate* scenario, run on the *DUT* platform

is one of the few cases where the performance curves cross each other on both platforms. These results for these three template engines on both platforms are shown in Figure 6.1.9 with different scales to emphasise the comparison between the shapes of the curves.

The two graphs share some similarities. In both the Intel PC and Raspberry Pi measurements *JTE* has a relatively high minimum duration and a relatively gentle slope, *Velocity* has a much lower minimum duration but a steeper slope, and *Thymeleaf* is somewhere in between those two. Where the two sets of measurements differ is in the crossing points and the effect that has on the suitability of the different template engines for different situations.

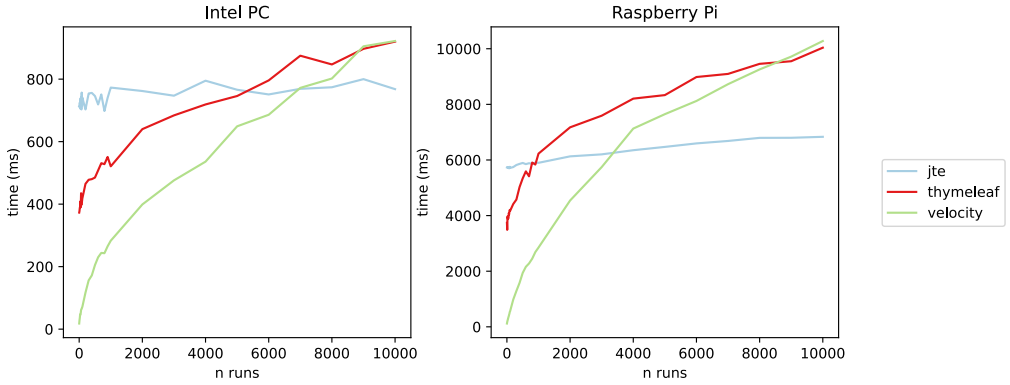


Figure 6.1.9: Performance comparison between Intel PC and Raspberry Pi running the *separate* scenario

On the Intel PC platform, *JTE* becomes a better choice than *Thymeleaf* at around 5000 runs, and a better choice than *Velocity* at around 7000 runs. On the Raspberry Pi DUT platform, however, *JTE* becomes a better choice than *Thymeleaf* at around 1000 runs, and a better choice than *Velocity* at around 3000 runs. The relationship between *Thymeleaf* and *Velocity* is more consistent, however, with *Velocity* appearing as the better choice up to around 9000 runs, at which point *Thymeleaf* takes the lead.

The consistency between *Thymeleaf* and *Velocity* tends to indicate that it is JTE that is the outlier, with a generally better performance on the Raspberry Pi device. The distinctive characteristic of *JTE* in this cohort of template engines is that it converts its templates to Java and pre-compiles them on first use. This adds a large time overhead to the first page request, but the resulting templates execute relatively fast, leading to a shallow slope as the volume of requests increases. This shallow slope is similar in both the Intel PC and Raspberry Pi cases, but the difference is in the time taken to prepare the templates on first use. It appears that it is this process which is comparatively faster on the Raspberry Pi device. A hypothesis is that the simpler *RISC* architecture of the ARM processor used

in the Raspberry Pi makes the compilation process simpler and thus faster.

The use of program compilation, a process that is usually done as part of software development before deploying and running a software application, places *JTE* in a category of its own among this cohort of template engines. This choice influenced the design of the *JTE* template language which, in turn, caused problems integrating *JTE* with the intermediate language discussed in Appendix E and prevented its use in Section 6.4.2. Further investigation of the behaviour of *JTE* was not pursued, as this aspect of its design was considered part of the software development process and therefore excluded from this research as per the scoping decisions in Section 2.16.5.

6.2 The Feasibility of Comparing Component Energy Use

6.2.1 Method

The apparatus from Chapter 5 had already been used to compare the energy usage of web server software written in Java, so no extra support was needed to run the evaluation scenarios. However, the comparison apparatus had been designed to trigger execution of server code via HTTP requests from the `LOAD` device. To integrate the code from the performance study with this architecture required either changes to the performance study code, or the addition of a server to listen for trigger requests and invoke an appropriate performance study script.

For simplicity, and to minimise any accidental issues due to modification of the performance study code, it was decided to use one of the web servers compared during the evaluation of the apparatus and to invoke the performance study code using the *Common gateway Interface* (CGI) protocol. CGI was designed as a method for invoking local programs from an HTTP request, so seemed a suitable choice for this task. The web server *Lighttpd* had consumed the least energy when serving static pages when evaluated in Chapter 5 so this server was used.

Lighttpd is very configurable and was supplied with an example of the configuration needed to serve CGI requests. All that was required was to modify this configuration slightly: to indicate where to look for scripts to execute when a CGI request is received, and to add a mapping to indicate that shell scripts should be executed by the system shell `/bin/bash`. The configuration changes were tested using a simple “hello world” CGI script.

Once the operation of the server had been confirmed, a CGI script was created that called the *one.sh* script described in Chapter 4. This script evaluates a single template engine by running all the scenarios a specified number of times. For the initial experiments, a count of 1000 times for each scenario was used. As this was just an initial study to determine if any difference in energy consumption could be observed, each template engine was measured just once.

6.2.2 Results

The measured net energy use (after subtracting the "baseline" energy use of a quiescent system) for each of the template engines is shown in Table 6.1. The same information is shown as a bar chart in Figure 6.2.1 for easier visual comparison.

Engine	**Energy (J)**
Freemarker	11.4936
Handlebars	12.4403
Hapax	10.7424
Jangod	9.33103
JTE	5.13393
Mustachej	7.22679
Pebble	8.48842
Solomon	4.90431
Stringtemplate	6.20053
Stringtree	8.39519
Thymeleaf	15.6721
Trimou	7.98896
Velocity	11.5069

Table 6.1: Net energy use for 1000 sets of scenarios by template engine

6.2.3 Discussion

The results from Table 6.1 show a discernible difference in energy usage between the template engines. However, the scale of the difference is considerably less than shown in the performance figures from Chapter 4. Also, the energy use figures from this experiment do not align well with the performance figures. A striking example is *JTE*, which exhibited the slowest performance at 1000 repetitions of

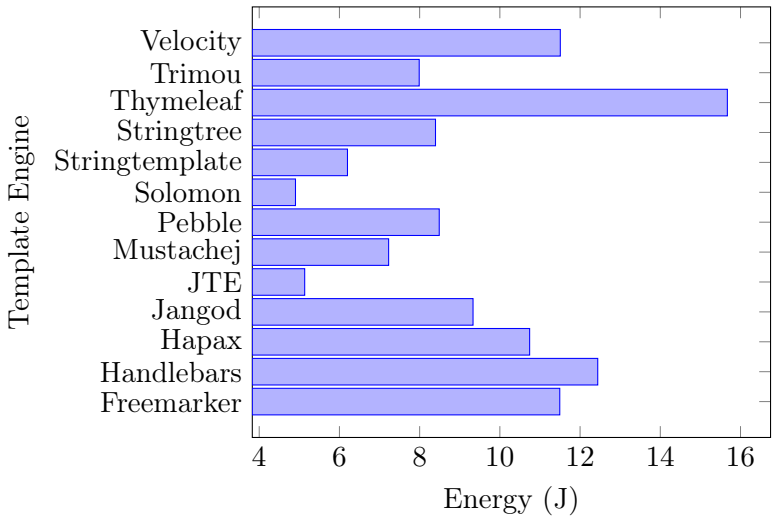


Figure 6.2.1: Net energy use for 1000 sets of scenarios by template engine

all the template engines in this cohort (see Figure 4.10.23. According to the "folklore" described in (Yuki and Rajopadhye, 2013), slower performance implies longer running time, which implies greater energy use. However, this component shows the second lowest energy usage in this experiment.

While the energy usage figures from this experiment are interesting, they should not be considered authoritative for several reasons:

Low number of samples The typical duration of 1000 repetitions of the suite of template scenarios for these template engines was between 10 and 20 seconds. The experimental test apparatus was configured to sample current and voltage at 1 second intervals, which meant that each run only had 10-20 readings from which to derive an average energy figure.

To illustrate this problem, Figure 6.2.2 shows the energy readings from 1000 runs of the *Pebble* template engine. *Pebble* was chosen as representative for the illustration as it is both a popular template engine used in multiple open source projects and, along with the less-commonly-used *Stringtree*, sits in the middle of the range for both the energy and time experiments shown in Figure 6.3.2 and Figure 6.3.3.

The readings varied during this period although, with a variance of 0.0367 and a standard deviation of 0.1917, most readings stayed relatively close to the mean of 2.3508 W. Despite this, there is clear evidence of "noise", so it is entirely possible that one or two outliers, either high or low, could perturb the average.

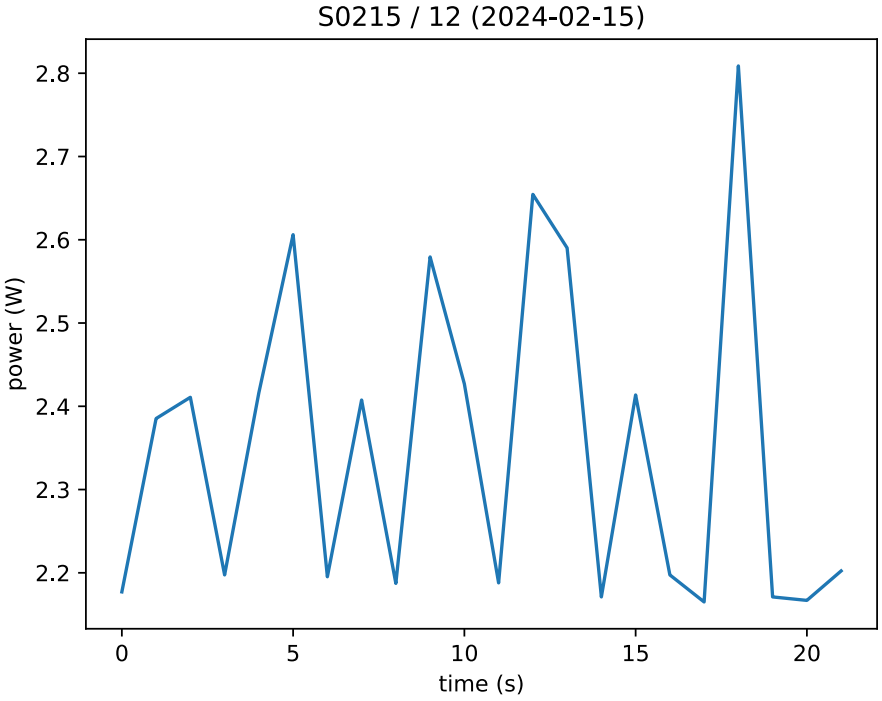


Figure 6.2.2: Energy readings from 1,000 runs of *Pebble*

System "noise" and other external factors The contribution to the energy due to the components in these experiments is small compared to the energy usage of the system as a whole, including the energy needed to run the *Lighttpd* server. Background processes and the general functioning of the systems required to support a modern operating system and its applications add "noise" to the measurements which can, on occasion, overshadow the energy usage of the components. Background processes were minimised during these experiments, leaving only those required to enable the operation of the software being tested. When combined with the low number of samples and the experiment being run only once for each template engine, this also acts to reduce confidence in the comparison.

Unrepresentative scenarios The performance experiments in Chapter 4 were designed to investigate the differences in behaviour of template engines in specific circumstances. In those experiments it made sense to repeat each specific scenario many times in order to amplify the differences and minimise the influence of external factors. If the intention is to obtain a directly applicable comparison between the energy usage of components when deployed to real applications, then a more representative scenario might be more appropriate.

Platform differences The performance experiments in Chapter 4 were conducted on an Intel CPU running a 64-bit operating system, while the energy experiments in this section were conducted on the energy measurement apparatus, which uses an ARM CPU running a 32-bit operating system. Differences between the performance results for the two different platforms are explored in Section 6.1.

6.2.4 Conclusions

The initial conclusions from this experiment were that there was an apparent difference in the energy consumption of the template engine components when tested with the simple scenarios used for the performance comparisons, but further investigation was required. Further experiments would need to include a larger number of cycles within each run and the averaging of several runs to minimise the impact of system noise and other external factors as well as the construction of further, more representative, scenarios. These further experiments are described in Section 6.3 and Section 6.4.

6.3 More Samples and Averaging

Following the approach taken in Section 4.3 and Section 6.1, a further energy comparison experiment was carried out in which each template engine was exercised 10,000 times for each of the performance test scenarios. Both energy use and time taken to process the test scenario were recorded. Each set of 10,000 was repeated three times for each template engine and the results were averaged to reduce the effect of system noise and interference.

6.3.1 Results

Increasing the number of repetitions of each template scenario served to reduce the variability of the energy readings in most cases. Figure 6.3.1 shows relatively consistent energy use during the measurement, with the addition of noise in a similar manner to the performance graph in Figure 4.10.1.

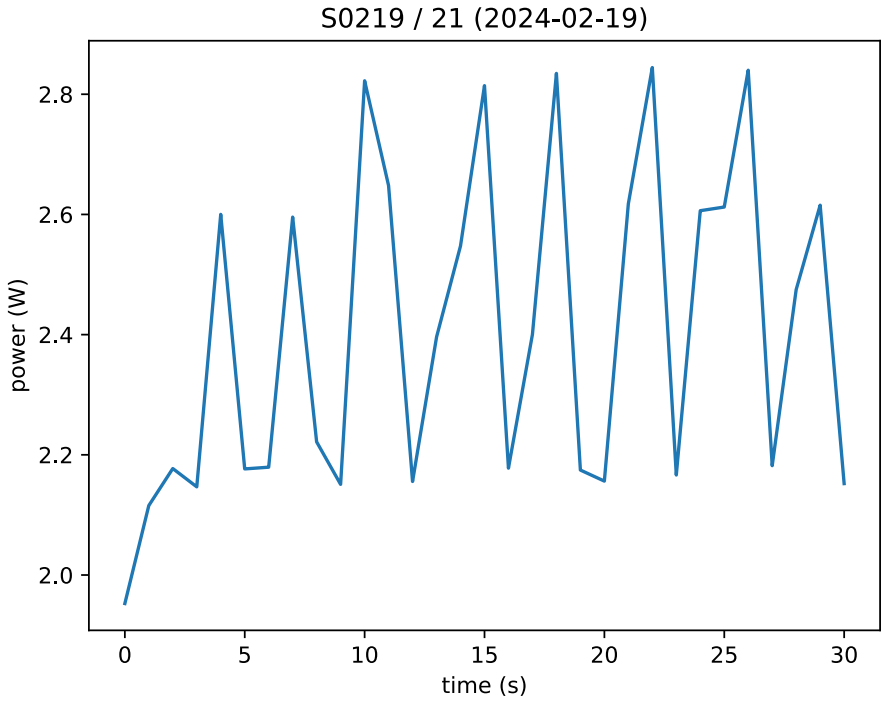


Figure 6.3.1: Energy readings from 10,000 runs of *Pebble*

Energy Use Figure 6.3.2 shows the comparative energy use of the selected template engines with the centre dot representing the average of three runs and the range bars indicating maximum and minimum values. The graph clearly shows a wide variation in energy use between template engines for this scenario.

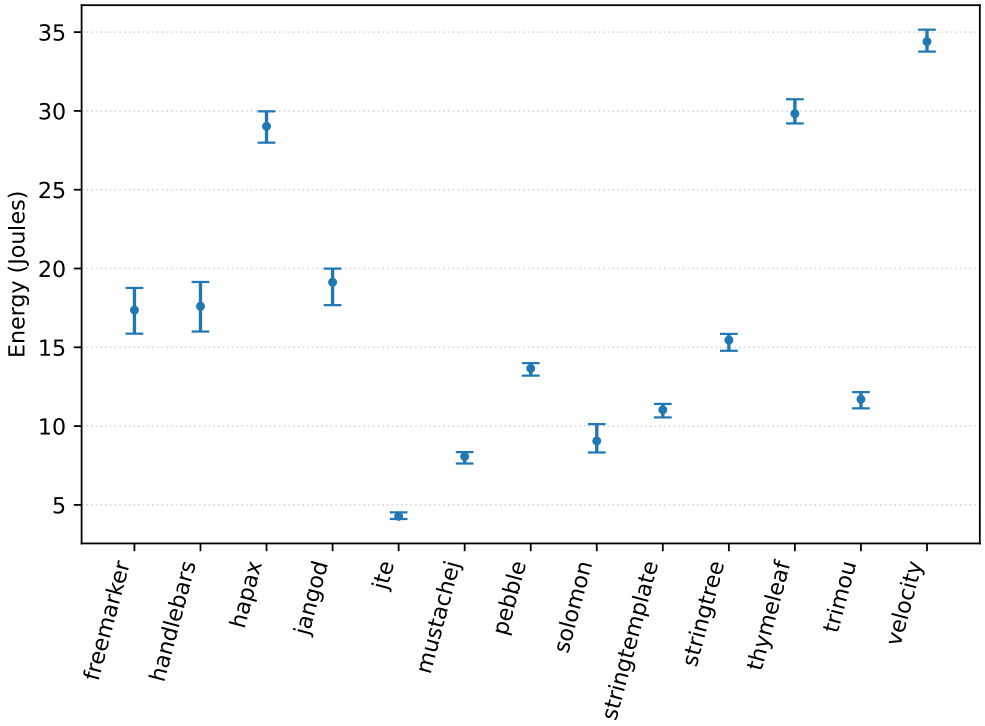


Figure 6.3.2: Averaged energy use by template engine

Time Taken Figure 6.3.3 shows the time taken for the experiments shown in Figure 6.3.2. There is also a wide variation between these figures which has some similarity to the energy usage but also has some important differences. For example, *Hapax* shows as faster than *Jangod*, but achieves that speed at the expense of extra energy usage.

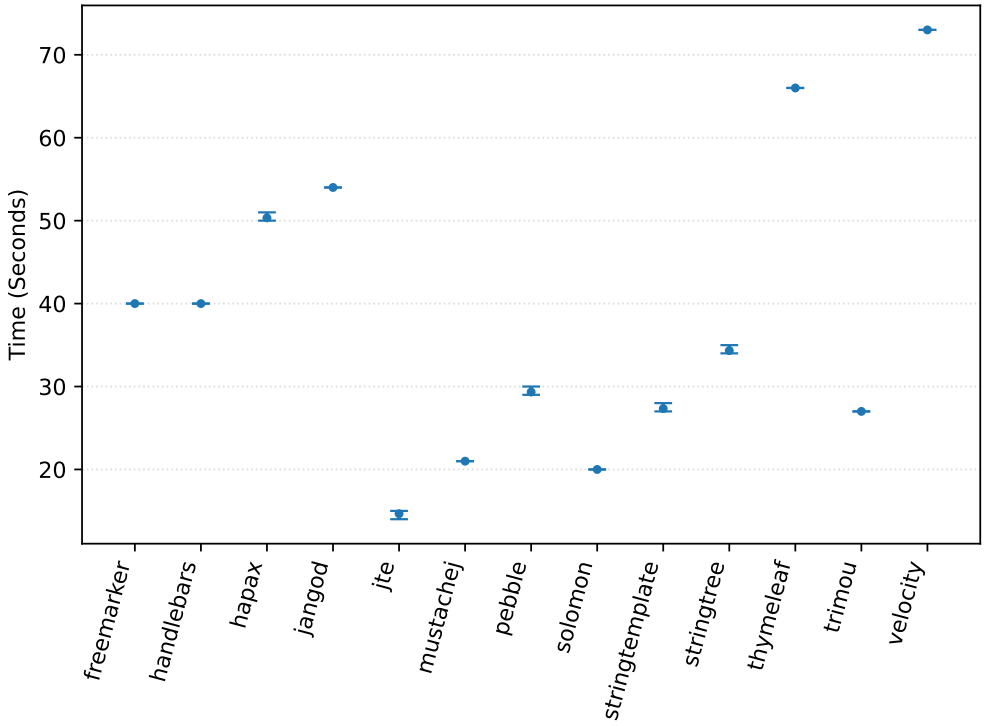


Figure 6.3.3: Averaged test duration by template engine

Running Power Consumption Having recorded both total energy use and time taken for each of the test runs, it became possible to derive the average running power consumption for each of the template engines by dividing the total energy used by the total time taken for each run. These derived values are shown in Figure 6.3.4.

6.3.2 Discussion

Unlike the improved performance experiments in Section 4.6, the experiments that provided the energy and duration values in this section only represented a single quantity of requests. However, there is a clear variation between the evaluated template engines in both time taken and energy used.

While the emphasis of this research is on comparing the energy used by different

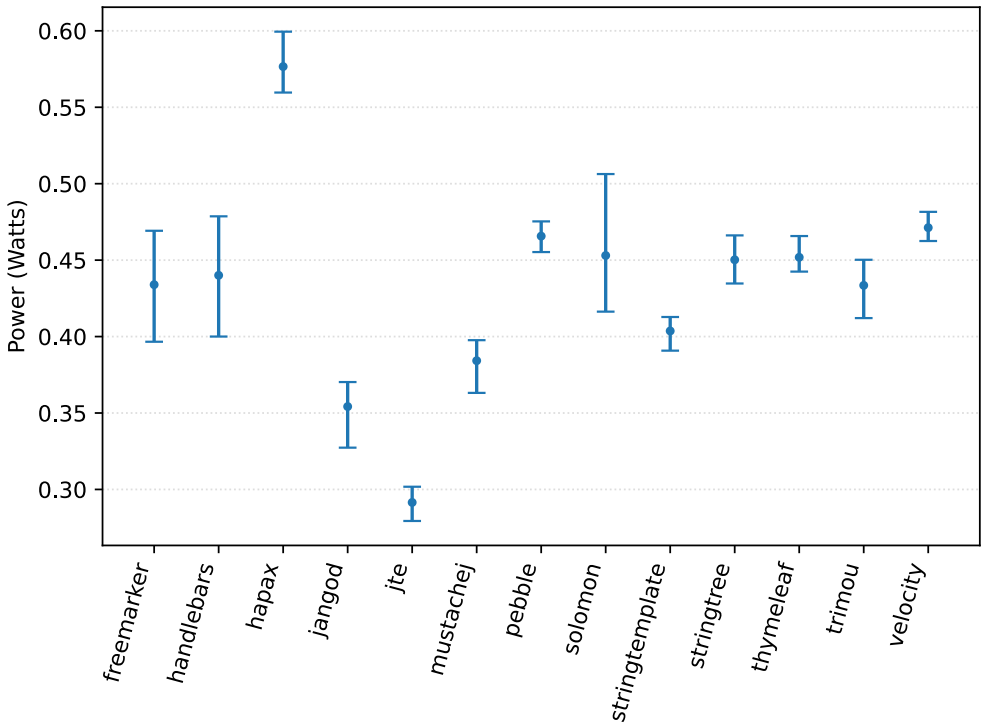


Figure 6.3.4: Averaged running power by template engine

software components, the addition of performance measurements also enables investigation into the running power consumption of the different components and its potential use as an indicator of the relationship between performance and energy use. This relationship between energy use and time taken, as shown in Figure 6.3.4, is also different for the different template engines. In these results, *Hapax* uses approximately twice the running power of *JTE*. A component that uses less running power is not necessarily a better or worse choice than one that uses more, however. The important values for capacity planning are performance, which can be used to determine the number of machines required to serve a desired number of requests per second, and total energy use, which indicates which components are more or less energy-efficient for the same task.

The disparity in calculated running power, even at a single quantity of repetitions, indicates that performance measurement is a poor analogue for energy use. However, the combination of the different measurements can help decide which components to choose, at least for this kind of scenario on this kind of platform. For example, in this experiment, *Velocity* exhibits both the worst performance and the worst energy use so is unlikely to be a good choice.*JTE*, on the other hand performs consistently well in both performance and energy use. One concern with *JTE* in this scenario is that it completes so quickly that it does not allow enough time for a large number of energy readings.

6.3.3 Conclusions

This experiment confirms that there is an observable difference in both performance and energy use of different template engine components. Combining the two sets of measurements also shows that the different components have different relationships between performance and energy use. This indicates that, without prior calibration specific to the code being executed, performance measurement is not a reliable way to predict overall energy use.

Results from Chapter 4 showed that the different components perform differently under different volumes of requests, but the experiments in this section all used the same volume. It is expected that energy use, time taken, and running power will also vary depending on request volume. Further experimentation is needed to validate this assumption as well as to determine how well such abstract test scenarios represent real deployments.

6.4 Measuring Template Engines in Context

6.4.1 Introduction and Scope

The preceding experiments have concentrated on testing the behaviour of specific template engine features. While informative, such scenarios may not be fully representative of the way such template engines are used on the web. To better simulate that kind of usage an example web page was converted to an intermediate template using the *GILT* template language discussed in Appendix E. This intermediate template was then used to generate templates for a selection of template engines. The performance and energy usage of the selected template engines were compared using the apparatus described in Chapter 5. The results of these comparisons are described in Section 6.4.3.

6.4.2 Methods

Selecting A Web Page and Generating Templates Web pages on the public internet vary widely. From simple pages holding a single image or a small amount of text to large, sprawling pages containing many distinct sections and structures. For the purposes of this investigation, a page was needed containing representative usage of many of the template features compared in previous sections. Such a web page would need boilerplate text, simple substitutions, template inclusion, iteration through a list, and boolean selection. Method invocation is not supported across the full range of template engines, so that was excluded from this investigation.

The desired set of features are found on many blogs and other similarly-structured sites. The blog page used for the comparison between servers in Chapter 5 only contains the content for a single page with little opportunity for iteration, so it was decided to use the "front page" of a blog that contains multiple tags, archive links, and sections for different blog posts. To avoid potential repeatability issues with relying on an externally-controlled website, a snapshot of the front page of the website and blog associated with this research[1] was selected.

Generating Templates The aim of the experiment was to compare the performance and energy use of a range of template engines. In order to do this, templates would be needed in appropriate template languages for each of the engines. The intention was to generate these templates from a single

[1] `https://greenprogrammer.net/`

intermediate template coded using the *GILT* intermediate language described in Appendix E. It quickly became clear during the creation of this intermediate template that the only way to test for correctness was to generate a template for one of the candidate template engines and then expand that template using the appropriate template engine. This process was cumbersome, and led to confusion whenever there was an error as to whether the issue was in the intermediate template, the template generation driver for the chosen target template language, or the behaviour of the target template language itself.

To eliminate some of these variables, and allow verification of the correctness of the intermediate template without using it to generate a template in a different template language, a new template engine was created. This template engine, named *gilt-native*, uses the *GILT* intermediate language directly as its template language. This template engine enabled faster development and testing of intermediate templates without the need for third-party template engines and direct comparison of the expanded template with the expected web page.

Once a intermediate template containing uses of all the desired features was complete, the *GILT* template generator was used to generate appropriate templates for each of the template engines. The full set of templates, both the *GILT* intermediate language and the generated results for the *Freemarker* template language, are given in Appendix F.

Selecting a Test Method When evaluating the effectiveness of the apparatus in Chapter 5, web pages were served by a selection of servers using both dynamic (Wordpress) and static page data. When comparing performance and energy use of components in Section 6.2 and Section 6.3, each test scenario was invoked directly using a script. Both approaches showed clear differences between energy use but in the server-based tests the differences between server implementations had the potential to overshadow the differences between components. In order to directly compare the performance and energy use of the candidate template engines it was decided to run each engine from a script in a similar manner to the component tests.

Scripts from the separate scenario tests were copied and modified to load appropriate context data and invoke each template engine using its `EngineDriver` implementation using its generated template.

Evaluating the Template Engines Prior to use on the measurement apparatus, each template engine was invoked with the intended context data and generated template, and the result was compared with the expected web

page. This highlighted several problems that needed to be addressed before the performance and energy use of the template engines could be measured and compared.

The most common problems were differences in the handling of "whitespace" such as space, tab, and newline characters. All the template engines in this cohort were designed primarily to generate output formatted using HTML. The rendering of HTML by web browsers is mostly insensitive to the presence or absence of such whitespace, so the template engine developers have not always cared about exactly preserving whitespace when generating web pages. For example, *velocity* failed to preserve a new line following a template inclusion directive (see Figure 6.4.1).

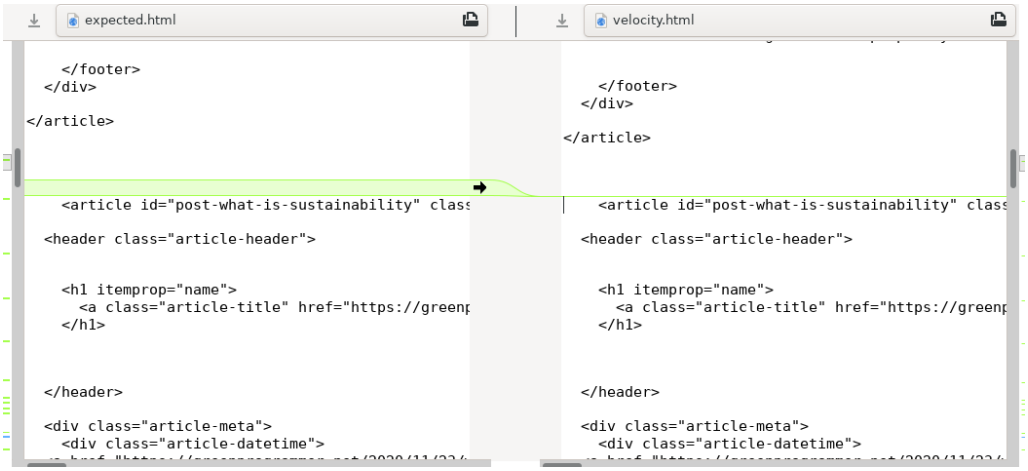


Figure 6.4.1: Difference in generated whitespace

After examining the output from these template engines, the acceptance criteria for a successful template expansion were relaxed to allow such minor differences in whitespace.

Other template engines, however, had bigger issues. Even though each of the template engines considered for this phase of the investigation had passed the individual tests in Chapter 4, some failed to generate acceptable output for this more complex scenario. For example, *Solomon*, which had performed flawlessly on the individual scenarios, initially generated a much smaller output file than the other template engines. When the output was examined it was clear that the template inclusion directives had failed. To understand why this was happening, the source code for the *Solomon* template engine was used instead of the provided library, and instrumented to discover the cause of the problem. An issue was discovered in the filenames of the templates to be included. *Solomon* expected the names of included templates to conform to the rules for identifier names in Java: starting with a letter then containing a combination of letters, digits, `_` and `$`. The filenames of the blog post summaries to be included had names beginning

with their creation date in the form `2022-10-23` which meant that the filenames were not valid Java identifiers. Once this issue was resolved, by renaming included template file to be valid Java identifiers, *Solomon* generated the correct output. This issue was subsequently addressed in a later release of *Solomon*.

Of the remaining template engines, some initially failed to process the generated template at all.

JTE, which had performed well in the tests of individual features, uses an approach that converts a supplied template into Java source code, then compiles that source code and runs it to expand the template. This approach requires additional specialist annotation in templates to enable the correct Java code to be generated. Unlike all the other template engines, which provide access to data values via a shared context visible to all templates, *JTE* requires that context value names and types be declared in a header section at the start of the template. When including one template in another, any context values used in the child template need to be both declared in the child template, and passed as "parameters" in the include directive. During the evaluation of individual template engine features and the development of the *JTE* driver, no child template made use of context values, so this requirement was never discovered. Although the *JTE* driver generated child templates with the correct declarations, it failed to generate the correct "parameters" in the parent template, which in turn meant that *JTE* generated invalid Java code that *JTE* could not compile. To add this feature to the *JTE* driver would require the template generation code to model the complete, transitive, contents of every child template in order to generate a correct parent template. This would require major changes to the architecture of the template generator, so *JTE* was excluded from this phase of the template engine evaluation.

Stringtemplate also required included templates to be valid Java identifiers but, even after the names of included templates had been adjusted as for *Solomon*, the template generation process still produced invalid templates. Further investigation determined that the *Stringtemplate* driver was using incorrect delimiters for some situations. The driver was updated to correct this error, and then *Stringtemplate* correctly processed the generated template.

Handlebars crashed when attempting to expand the generated template. Investigation indicated that this was a problem with the specific version of the *Handlebars* library that was being used, which made use of some features incompatible with recent versions of Java[2] [3]. Updating the library removed this

[2] `https://github.com/swagger-api/swagger-codegen/issues/10966`
[3] `https://teamtreehouse.com/community/broke-when-adding-blocks`

problem, but revealed further issues. All previous comparisons had been performed with a single version of the *Handlebars* library. Updating to a new version, even if the further issues could be addressed, would raise questions about the comparability of the measurements, so *Handlebars* was also excluded from this phase of the template engine evaluation.

Mustachej required some minor changes to the Driver, but otherwise functioned correctly.

Hapax does not fully support loops and conditionals, so was excluded from this phase of the template engine evaluation.

Thymeleaf employs an HTML-specific syntax that would require altering the intermediate template document in ways that would make it incompatible with the other template engines, so it was also excluded from this phase of the template engine evaluation.

Energy and performance comparisons were therefore performed on the following template engines: *trimou*, *solomon*, *freemarker*, *stringtemplate*, *stringtree*, *jangod*, *velocity*, *pebble*, *mustachej*, and *gilt-native*.

A script was created to load a specified context and use a single template engine to expand the templates generated from the example website. Each page expansion was run 100,000 times. The overall run of 100,000 expansions was then repeated twice more, and the results averaged, to minimise the impact of external interference.

6.4.3 Results

The energy usage of the averaged runs for each template engine are shown in Figure 6.4.2 and the time taken by each template engine is shown in Figure 6.4.3. These two data sets have been combined to produce running power figures for each template engine, and these results are shown in Figure 6.4.4

6.4.4 Analysis

In Figure 6.4.2 there is an observable difference between the energy usage of the different template engines. There is also a clear distinction between two groups of template engines: those which use less than 100J, and the four template engines that consume over 350J for the same scenario. This distinction suggests that there is some key aspect of the design or implementation that differs between the two

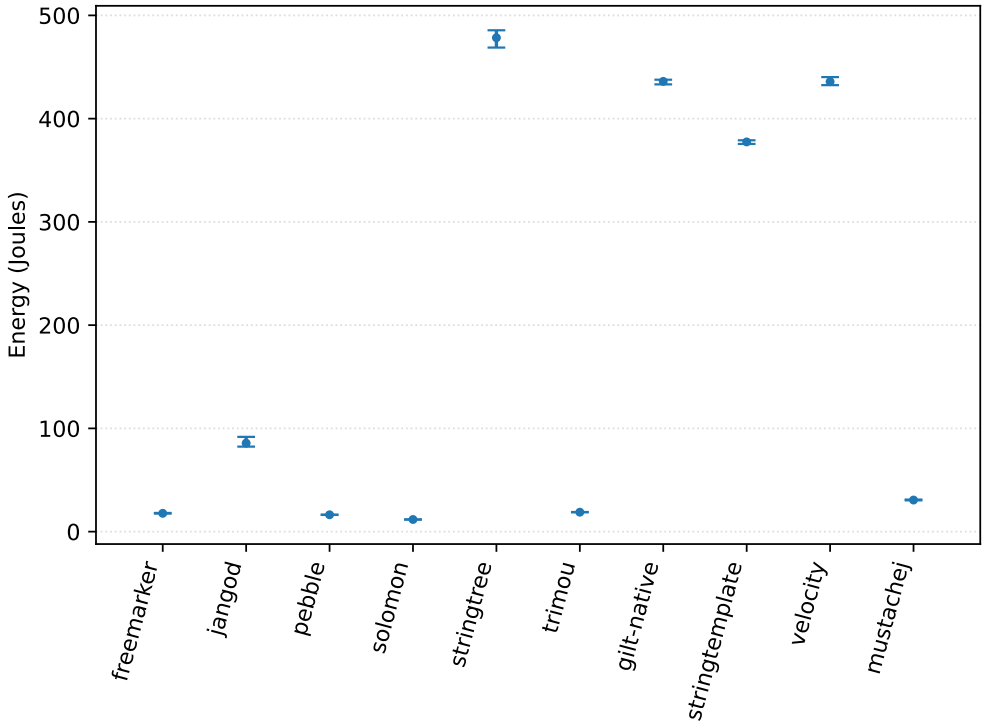


Figure 6.4.2: Energy usage of 100,000 page expansions

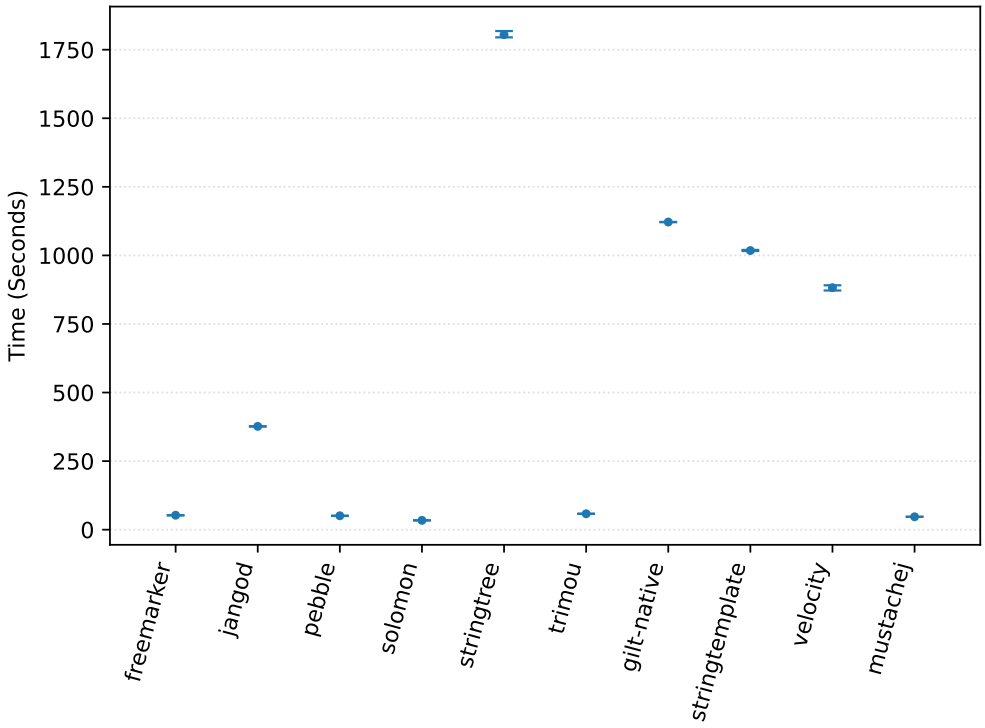


Figure 6.4.3: Time taken for 100,000 page expansions

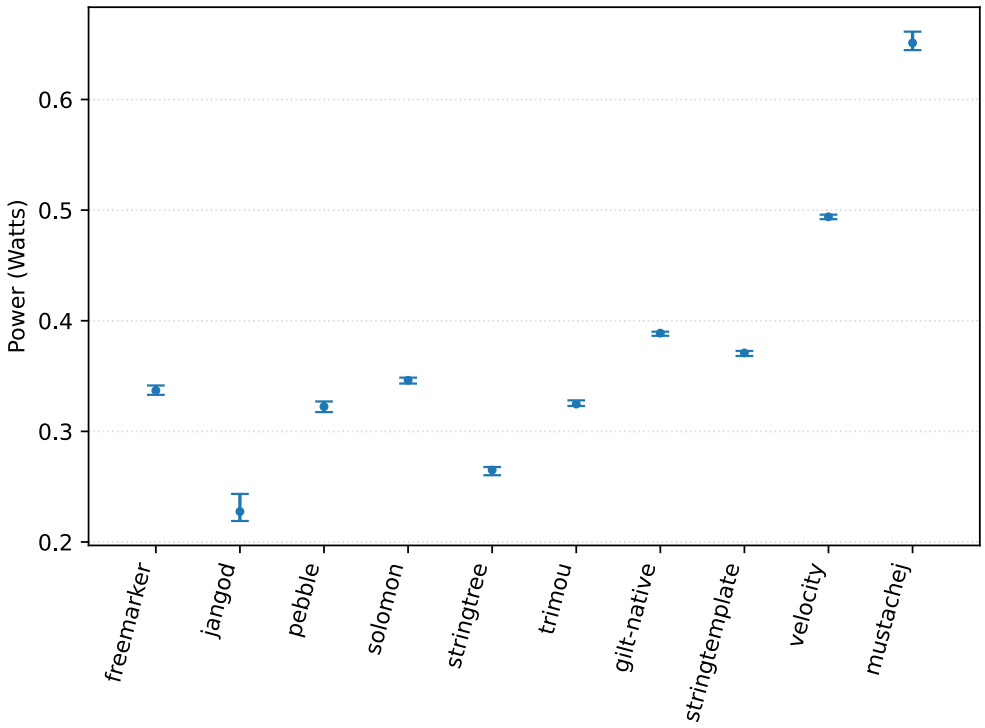


Figure 6.4.4: Average running power for 100,000 page expansions

groups. Investigation of the reason or reasons for this difference is out of scope for this research, but such a large difference could indicate a potentially fruitful area of future research (see section:future work).

The times taken shown in Figure 6.4.3 largely echo the energy usage results, with the same four template engines taking the longest to process the specified number of template expansions. The relationship between the template engine energy usage and duration is not constant, however. For example, when considering only energy usage, *velocity* is noticeably worse than *stringtemplate*, but for time taken, these positions are reversed. This difference can be clearly seen in Figure 6.4.4. *velocity* uses more energy in a shorter time, which shows as a much higher running power consumption.

While the power figures in Figure 6.4.4 can be useful when comparing components whose energy usage and time taken are roughly similar, they are not indicative of any notion of efficiency. *mustachej*, for example, shows by far the highest running power consumption, but is still among the lowest for total energy consumption and time taken to complete the task.

6.5 Discussion

This chapter has explored some different ways of comparing the energy usage of template engine components. Comparing software components is challenging because, unlike full applications, they cannot be executed without other software to configure, initiate, execute, and shut down the components. This external software can potentially influence the time and energy performance of the components. The results in this chapter show that software components do not usually perform identically in different scenarios.

The experiments have shown clear differences in energy usage between template engine components. The combined individual scenario experiments in Section 6.3 provide a ranking of template engines by energy usage as shown in the first data column of Table 6.2. If considered in isolation, this ranking would seem to provide a way to select the most energy-efficient template engine for a project. However, when the same template engines were compared using the arguably more realistic scenario of generating a complete web page, the rankings were very different as shown in the second data column of Table 6.2.

To examine the correspondence between these two sets of rankings, a Kendall Rank Coefficient (also known as Kendall's τ) was calculated. This measure is a non-parametric hypothesis test for statistical dependence and produces a value

Template Engine	Energy Efficiency Ranking	
	Individual Scenarios	Full Page Generation
freemarker	8	3
gilt-native	*n/a*	8
handlebars	9	*n/a*
hapax	11	*n/a*
jangod	10	6
jte	1	*n/a*
mustachej	2	5
pebble	6	2
solomon	3	1
stringtemplate	4	7
stringtree	7	10
thymeleaf	12	*n/a*
trimou	5	4
velocity	13	9

Table 6.2: Comparative energy-efficiency rankings for all template engines

between -1 and 1, with positive numbers indicating greater correspondence and negative numbers indicating lesser correspondence.

However, this calculation is complicated by the incomplete nature of the two sets of rankings. Kendall's test requires directly comparable sets of rankings, so the data needed to be adjusted to achieve this before the statistical test could be performed. There was no problem with the rankings for template engines that appear in both lists, but template engines that only provided rankings for one of the comparisons have no obvious position in the rankings for the other.

Four approaches to address this issue were considered:

- replace missing values with a low or out-of range value, such as 0 or 1
- replace missing values with a high or out-of range value, such as 13 or 14
- replace missing values with a middle or average value
- remove template engines with missing values from the comparison.

The high and low approaches were rejected as they would skew the results of the

test and reduce the confidence in the calculated value. The middle or average approach, while potentially having less impact on the calculated result, would also invalidate the calculation, as the Kendall Rank Coefficient assumes distinct ranking values. As a result, only templates with ranks for both the individual scenarios and the full-page generation were considered for the comparison. Removing rows where one side or the other was missing left some gaps in the rankings. These gaps were closed by moving adjacent rankings down until both columns contained only the numbers 1 to 9.

The final list of template engines, with their adjusted rankings, is shown in Table 6.3.

Template Engine	Energy Efficiency Ranking	
	Individual Scenarios	Full Page Generation
freemarker	7	3
jangod	8	6
mustachej	1	5
pebble	5	2
solomon	2	1
stringtemplate	3	7
stringtree	6	9
trimou	4	4
velocity	9	8

Table 6.3: Adjusted rankings for template engines that participated in both studies

The adjusted rankings were then able to be used to generate a Kendall Rank Coefficient (τ) of 0.277. The τ value is positive, indicating that the two sets of rankings are more correlated than they are un-correlated.

However, taking as a null hypothesis that the two sets of rankings are independent, which would produce a τ value of 0, this results in a p value of 0.358, which is low, but not low enough to reject the null hypothesis completely.

The use of the GILT template generation software was an enabler for these experiments but did not figure in the measurements themselves. All GILT processing took place to generate all the templates before any timing or energy experiments were run. This use of an eager evaluation approach removed the time and energy required for the GILT processing from the software being measured. The main benefit of using the GILT software was that it enabled

easier interchangeability of template engine components by removing the dependency on manual editing of different template formats.

6.5.1 Comparisons With the Performance Studies

The performance studies in Chapter 4 could also potentially be used to rank the template engine components based solely on performance. However, these experiments revealed that the various template engines exhibit different performance characteristics based on the overall volume of requests. Any potential rankings based on the performance studies of individual template engine features would only be valid for the specific features being evaluated at that particular request volume. The energy comparisons were performed at fixed request volumes in order to gain a workable number of energy usage samples, but it is expected that energy-efficiency of some template engines would also vary with request volume.

6.5.2 The Relationship Between Energy Use and Performance

For the components evaluated in this chapter, the relationship between energy usage and time taken was neither constant between template engines nor constant between scenarios. This relationship, formulated as average running power is shown in Section 6.3.1 and Figure 6.4.4. In both scenarios the ratio between energy use and time taken, expressed as Joules per second (Watts) ranged from under 0.3 W to around 0.6 W. This variability suggests that models which aim to predict energy usage based on time performance are unlikely to be generally applicable.

6.6 Conclusions

The key conclusion drawn from the experiments in this chapter is that different software components can vary widely in both energy usage and performance when performing the same task, and that the relationship between performance and energy usage is not constant across different evaluation scenarios.

One implication of this is that models which attempt to predict energy use solely from other measurements such as performance, task complexity, or static analysis of code will not yield generally-applicable results. However the results affirm the importance of measuring actual performance and energy usage during system

tests as an effective way to compare energy use between software changes. The apparatus described in Chapter 5 can enable this testing as part of the regular software development and automated testing process.

Chapter 7

Conclusions

7.1 Reflection

Environmental sustainability is one of the major challenges of the modern world. We need to immediately and significantly reduce the production and emission of greenhouse gases. Energy production, driven by energy consumption, is a major contributor to greenhouse gas emissions. The energy consumption of the world's ICT systems is a significant contributor to global energy demand.

This research has explored ways to improve the sustainability of large-scale ICT systems by reducing energy consumption and its associated greenhouse gas emissions. Most ICT systems are controlled by software that has the capacity to increase or decrease the required energy. Software systems are created and maintained by software developers in the context of existing software and infrastructure, organisational culture, and personal skills. Exploring this context of software development gave insight into many reasons why energy efficiency and the sustainability of software systems are often not a high priority.

The literature on computing and sustainability revealed "vague, diverse and contradictory ideas" and confusion between subdisciplines. Several research teams have released manifestos or proposed frameworks in an effort to clarify the state of sustainable computing research, but there is little evidence of this extending beyond academia.

A research gap was identified in the use of component substitution to reduce the energy use of large web software, and web template systems were selected as a representative category of component for investigation. A general lack of information on the performance and energy usage of these components was

identified, so a series of experiments were carried out to discover the feasibility and effectiveness of component substitution as a sustainability strategy.

A feasibility study was conducted to determine whether the scale of differences between a selection of template components was large enough to warrant further investigation. The results of this study indicated that the fastest component (*Solomon*) was, on average, 2642 times faster than the slowest (*Casper*). Further performance comparisons were performed that showed that although individual component performance varied depending on task and load, there were still considerable differences in performance between components. Selecting components with better performance could therefore potentially reduce the number and/or capabilities of computers needed to run a popular service, thereby reducing both energy usage in operation and the embodied carbon costs of manufacturing and disposal of the hardware.

Although optimising performance can help reduce hardware requirements, it is not a direct measurement of energy use. A prototype test apparatus was designed and constructed to support automated, programmable, comparison of the energy used by applications or software components under different scenarios and loads. Comparisons using the apparatus produced some interesting results, including that the popular *WordPress* application used by a very large number of websites requires a lot more energy to serve each web page than an equivalent traditional "static" website.

The feasibility of component substitution as a strategy depends on the cost and complexity of replacing one component with another. However, software components provide individual APIs and are rarely directly interchangeable. To explore potential solutions to this problem, a framework was constructed to simplify the creation of "drivers" for different component APIs. Components can also differ in data formats and data storage requirements. To explore this issue, an intermediate data format and associated conversion tools were created to simplify the generation of appropriate data for each component. This combination of framework, data format, and tools was then used to create more complex scenarios to evaluate the energy consumption of the selected template components. The results showed that there was also a considerable difference in energy consumption between components, although this did not always align with the performance of the components.

The implication of this research is that replacing software components with alternatives with better performance can potentially help improve sustainability. However, performance on its own is not a good proxy for operational energy usage.

A more effective and direct way to assess and reduce the energy usage of a software system is to use an energy comparison apparatus, such as the prototype developed as part of this research, to include energy usage comparison in existing automated test suites. This approach would allow comparisons of energy usage to be performed in load and deployment scenarios customised to the needs of the software in use.

7.2 Evaluation of Research Objectives

A set of research objectives were described in Section 2.17. This section examines the outcomes and evaluates the effectiveness of this research against these objectives.

7.2.1 Investigate the context of web software and its development

Research was conducted on the scale of the internet, the history of computing, and how software is made, including development team roles and how software developers choose components. It was discovered that many forces act to prioritise other goals than sustainability. Improved education, reliable information, and energy measurement tools that fit in with existing development processes are required to enable software developers to improve the sustainability of the software being produced or maintained.

7.2.2 Explore the differences in performance of a selection of template engine components

A representative category of software component was selected and a small cohort of implementations was chosen from within that category. A software performance test harness was created for a feasibility study to explore the performance of the implementations in a range of scenarios. This initial feasibility study clearly indicated large performance differences between these components and demonstrated that the area was worthy of further study. However, there were some issues with the interaction between the performance test harness and the components being evaluated.

An improved performance test harness was constructed to address the issues raised by the feasibility test. A driver framework was created to allow

independent loading of template engine components into the comparison framework. A new component could be added and compared by writing a driver for the new component with no need to modify or recompile the comparison framework. The comparison framework was used to compare both component performance and (when used with the energy test apparatus) component energy usage.

This confirmed the large differences in component performance found in the feasibility study and also discovered that component performance varies not only between components and scenarios but also between the volume of requests.

This research was effective in showing that there are ways to simplify the complex and potentially expensive task of comparing software components. The development of standard tools for performance and energy usage comparison as well as tools to overcome the differences in component APIs and data formats has helped compare the performance and energy-use differences between components. With the ability to compare components comes the ability to select ones with better performance and lower energy use. This, in turn, could lead to an overall reduction in energy use and associated carbon emissions.

This research was also effective in uncovering the scale and complexity of the performance differences between supposedly-equivalent software components. This stands in sharp contrast to the documentation and other public information available about these components. Such documentation rarely mentions performance, and when it does it is usually in vague terms such as "fast" or "powerful". Using this information to select components with optimum performance for a desired application could potentially reduce the number of computers needed to serve that application, thus reducing the embodied carbon in the system.

7.2.3 Construct a test apparatus to compare the energy use of software during operation

An apparatus was designed, constructed, and evaluated to assess the energy usage of running software under different scenarios. This phase of the research was very successful, resulting in a useful apparatus that was then used for subsequent experiments.

This research was effective in showing that an apparatus could be constructed to compare the energy usage of both components and whole applications in a variety of scenarios. Importantly, this apparatus was built for much less cost

than traditional lab equipment and provided software interfaces so it could be included in the continuous integration processes commonly used to test software during development. Comparisons run on the apparatus revealed the wide range of differences in energy consumption between equivalent software, in particular, the much larger energy required to serve websites using WordPress software.

7.2.4 Use the apparatus from Chapter 5 to compare the energy use of common web software and the feasibility and effectiveness of substituting template engine components

The energy comparison apparatus was used to compare the energy usage of a selection of popular web server applications, both when serving static and dynamic web pages. The results of this comparison clearly showed not only a difference in energy usage between servers but also highlighted the large hidden energy cost of dynamic website generation.

An intermediate language was designed to represent common features of template languages and a software tool was constructed to convert templates expressed in the intermediate language into the various formats required by different template engines. The template generation tool also used a driver mechanism, allowing new template languages to be added without the need to modify the generation tool. This intermediate language and tool were successfully used to generate representative web page templates for template engine comparisons, based on a real website, for each of the supported template engines.

The cohort of software components that had been compared for performance was then compared for energy use. Energy use was compared both when re-running the performance evaluation scenarios and when running new scenarios designed to be more representative of common use. The results of this comparison showed differences in the energy usage of the components but the relationship between performance and energy usage was different for each of the components.

The ability to compare both performance and energy usage of the same software highlighted the differences in the relationship between the two metrics. Although many attempts have been made in the literature to derive ways of using performance to predict energy usage, this research clearly showed that the relationship between the two varies both between components and between usage scenarios. Any models which claim to relate the two metrics will therefore only be of use in very limited circumstances.

7.2.5 Research Objectives Conclusions

Of the four research objectives, three were very successful, providing a range of contributions to knowledge that should help improve the sustainability of software systems. The objective of investigating the context of web software and how it is developed confirmed existing concerns about the lack of support for the development of sustainable software. There is increasing interest in this area by academic researchers, but this does not appear to have had much influence on existing software development practices. However, that research did provide the justification for exploring ways to improve the sustainability of software systems through other means that do not require rapid changes to the practices, structure, and roles of software development teams.

7.3 Contributions To Knowledge

This research has generated several contributions to knowledge, which are described below.

7.3.1 A Novel Self-Contained Apparatus for Comparing Software Performance and Energy Consumption

While there have been several studies that have measured or compared the energy usage of software, they typically involve expensive laboratory equipment or manual measurements, require specific computer hardware, or concentrate on the investigation of theoretical models rather than enabling energy measurement and comparison as part of routine software development and testing.

This research has developed a self-contained, low-cost prototype for an apparatus that can be integrated into the automated continuous integration processes used by many industry teams when developing and testing software. The apparatus can operate without human intervention to compare the performance and energy use of a wide range of software applications and software components with customisable scenarios and load levels. The apparatus is designed to integrate with automated test systems, but also offers a manual web interface, which can be useful when evaluating candidate software to include in a project.

7.3.2 Energy Use and Performance Comparisons for a Cohort of Web Server Implementations

The apparatus described in Chapter 5 was evaluated by comparing the energy use and performance of a variety of web server implementations, including two of the most widely used servers worldwide, to serve some representative web content. The results clearly indicate the differences between the implementations. This information is not available elsewhere.

7.3.3 The Relative Energy Usage of WordPress Compared to Static Websites

WordPress is a very popular tool for creating, managing, and serving websites, yet users are mostly unaware of its energy consumption. This research has revealed that *WordPress* requires many times more energy to serve the same web pages compared with a traditional "static" website. Although *WordPress* can provide website features that static sites cannot, these features are not used by all *WordPress* websites. There are a potentially large number of websites currently running *WordPress* that could be switched to a static approach for an immediate saving in energy.

7.3.4 A Novel Extendable Java Framework for Template Engine Comparison

Selecting a template engine component for a project can be a daunting task. The lack of detailed information provided by the component creators leads to a choice between committing to one component (and hoping it is a good choice) or conducting comparative experiments. The construction of rigorous comparative experiments can be complex, but that complexity can be reduced using the template engine comparison software framework constructed for this research. When using this framework all that is needed is to define the template scenarios to be tested and write driver implementations to suit the API of each candidate template engine. The framework takes care of dynamically loading the correct drivers and templates, running the selected scenarios, and collecting results ready for analysis.

7.3.5 Energy Use and Performance Comparisons for a Cohort of Template Engine Implementations

As part of this research, the comparison framework described in Chapter 4, the intermediate language described in Appendix E, and the apparatus described in Chapter 5 were utilised to compare the energy usage and performance of a cohort of open source template engines of the kind commonly used for the generation of dynamic web pages. These comparisons involved multiple platforms and a range of scenarios and load levels. This information is not available elsewhere.

7.3.6 A Challenge to the Notion of Execution Speed as a Proxy for Software Energy Usage

It seems to be a common opinion, even being referred to as "folklore" in one paper (Yuki and Rajopadhye, 2013), that energy usage of software is in some way related to the time taken to accomplish a task. Many researchers have attempted to construct models to relate the two quantities in the hope that relatively simple and cheap performance measurements could be used to produce an estimate of energy usage. This research has shown that when evaluating a range of software that performs a broadly similar task, there is no constant or reliable relationship between execution speed and energy use. Different components have different relationships, which vary between different evaluation scenarios.

This research leads to the conclusion that the only practical and effective way to compare the energy use of software in operation is to measure it.

7.3.7 A Challenge to the Notion of Task Complexity as a Proxy for Software Energy Usage

Many works on the topic of software energy usage have made the implicit or explicit assumption that the energy use of a software system is primarily dependent on the complexity of the task the software system performs. This may be true in cases where there is a very large difference in task complexity. In such cases, reducing the complexity of the task is likely to reduce overall energy usage. However, this research has shown that the architecture and design of the software to perform the task can also make a big difference, even when the task being performed is the same.

7.3.8 The Efficacy of Component Substitution as a Strategy to Improve Software Sustainability

This research has discovered major differences in the performance and energy use of different software components when performing similar tasks. This clearly indicates that the choice of software components, when designing a new system or modifying an existing system, could be a viable strategy to improve the sustainability of software.

7.4 Future Work

Each of the studies in this research has also revealed the potential for further work.

7.4.1 The Performance of Software Components

The performance of software components under real conditions is an under-researched area. Historically, academic research has concentrated on the study of algorithms and data structures, but such abstract concepts are less-often considered in practical software development. Instead, software architects and developers attempt to design and construct systems from components, but lack detailed information to use when selecting between many candidate components. Although a poor proxy for energy use, in general, component performance comparisons can assist in designing software systems that require fewer or less powerful computer hardware systems to support them.

Within the category of template engines, more research is needed to investigate the design and characteristics of the selected components to investigate why their performance is so different, and to determine if any software development or evaluation guidelines can be derived from this information.

The generally similar performance of the embedded web server components seen in Section 5.5.2, despite their different implementations and energy use, is also an anomaly which would benefit from further investigation.

This research has concentrated on one specific type of component, but there is a very broad range of software component types available, many of which also have a large number of substitution candidates. Further research is needed on the performance of other types of software component to investigate whether the results from this research are in any way representative.

Further research would also be useful on the impact on application performance of including multiple components and the possibility of models that could predict or estimate overall performance or hardware requirements from a "software bill of materials".

Most research in this area is limited by a lack of information, so there is also potential for future research to find ways to grow the range, detail and reliability of information available on software components. An approach to this might be to develop tools and benchmarks that can be used to compare components. For template engine components these might take the form of standardised template scenarios that can be applied to many candidate template engines.

7.4.2 An Apparatus To Compare Energy Usage

The energy comparison apparatus developed in this study was only a prototype and thus there is ample opportunity for further research and development. Potential research areas include:

- evaluating the use of alternative "DUT" platforms to better simulate the hardware used in commercial applications.
- evaluating or developing different energy measurement circuits that can handle larger voltages and currents.
- improving the software to support multiple scenarios in a single test run.
- improving the software and database to support the parallel operation of more than one apparatus.
- evolving the apparatus into a robust, self-contained device.
- investigating and constructing tools to help integrate the apparatus with specific software continuous integration systems.
- enhancing the output and presentation of comparison results to better inform decision-making

Further research is also needed in the broader field of software energy usage measurement and communication. The results of the energy usage of different software applications and components need to be disseminated so that potential users can make an informed decision without always having to perform their own comparisons of energy usage. This in turn would need standards and repeatable methodologies as well as devices such as this apparatus.

7.4.3 Substitution of Incompatible Software Components

The software developed as part of this investigation to support the substitution of components that are partly functionally equivalent but provide different APIs and data formats was specific to template engine components, but the approach is potentially applicable to other classes of components. More research would be required to determine how broadly applicable this approach is to other contexts.

7.4.4 The Energy Use of Software Components

The performance results of the evaluation of template engine components using the comparison apparatus in Chapter 6 are interesting, but do not align well with the performance results of the studies in Chapter 4. The measured values were expected to be different from the previous experiments as they were executed on different computers with different processors. However, what was not expected was that performance *rankings* would be so different. There is clearly something else different about the two platforms, which needs further investigation.

The difference in performance rankings raises questions about the accuracy of the energy use comparisons and rankings for the same scenarios. These also need to be investigated, ideally in a range of different request volumes as was done for the performance studies in Section 4.6.

7.4.5 Other Related Future Research

This research has continually shown that there is a lack of detailed and reliable information about software in general, but in particular about software components. Electrical appliances have energy ratings, electronic components have data sheets, but software components mostly have little more than feature lists and usage instructions at best. Further research is needed into ways to obtain, share, and compare information about the characteristics of software components.

The comparisons of server energy use in Chapter 5 highlighted the poor performance of dynamic websites and, in particular, *WordPress*. Further research is needed on the development of software that can make the management of static websites as easy and flexible as *WordPress* but without the high energy cost.

7.5 Final Remarks

As mentioned at the very beginning of this dissertation, this PhD was motivated by concern for the future of our environment. This research has revealed the increasing environmental impact of computing technology and with it the importance of improving the sustainability of software-driven computing. The experimental results of this research have shown that the environmental impact of both software applications and software components can be reduced by choosing alternatives with better performance and/or better energy efficiency and by improving the process of software development to include these aspects alongside other requirements and document them alongside other features.

As a researcher and the author of this dissertation, I earnestly hope that this work will make a positive difference.

Appendix A

Java Template Engines on GitHub

- httl https://github.com/httl/httl
- Rocker https://github.com/fizzed/rocker
- Trimou https://github.com/trimou/trimou http://trimou.org/
- jte https://github.com/casid/jte
- Liqp https://github.com/bkiers/Liqp
- Chunk https://github.com/tomj74/chunk-templates
- Smarty4J https://github.com/linux-china/smarty4j
- Rhythm https://github.com/rythmengine/rythmengine
- Water https://github.com/tiagobento/watertemplate-engine
- Wit https://github.com/febit/wit
- Basis https://github.com/badlogic/basis-template
- japid https://github.com/branaway/Japid
- japid42 https://github.com/branaway/japid42
- gt-engine https://github.com/mbknor/gt-engine
- gt-engine-play2 https://github.com/mbknor/gt-engine-play2
- Cambridge https://github.com/erdincyilmazel/Cambridge
- golang-like https://github.com/proninyaroslav/java-template-engine

- Min Velocity `https://github.com/pfmiles/min-velocity`
- Mixer2 `https://github.com/nabedge/mixer2`
- Panettone `https://github.com/caelum/vraptor-panettone`
- Carrot `https://github.com/codeka/carrot`
- PowerStat `https://github.com/PowerStat/TemplateEngine`
- Knotx `https://github.com/Knotx/knotx-template-engine`
- Inflectible `https://github.com/gvlasov/inflectible`
- te4j `https://github.com/whilein/te4j`
- static-mustache `https://github.com/sviperll/static-mustache`
- vtte `https://github.com/hgschmie/vtte`
- jade `https://github.com/dhanji/jade`
- Soulspace `https://github.com/lsolbach/TemplateEngine`
- jtt `https://github.com/tokuhirom/jtt`
- sitebuilder `https://github.com/JonathanGiles/sitebuilder`
- succinct `https://github.com/tliron/succinct`
- jelly `https://github.com/pranitbose/jelly`
- jjenkov `https://github.com/jjenkov/template-engine`
- antlr-template `https://github.com/worstenemy/antlr-template`
- datatree-templates `https://github.com/berkesa/datatree-templates`
- Simple-Template
  `https://github.com/jalian-systems/Simple-Template`
- Template72 `https://github.com/template72/template72`
- stencil `https://github.com/impossibl/stencil`
- coffeemaker `https://github.com/mouse0w0/coffeemaker`
- Cedilla `https://github.com/ansorre/Cedilla`
- rstl `https://github.com/rstl/rstl`
- templar `https://github.com/synapticloop/templar`
- jhaml `https://github.com/iafonov/jhaml`
- ljincheng `https://github.com/ljincheng/template`
- disc99 `https://github.com/disc99/template`
- iNamik `https://github.com/iNamik/iNamik-Template-Engine`

- Mini-templator https://github.com/bsorrentino/minitemplator
- Jakotem https://github.com/moznion/jakotem
- swiftmarker https://github.com/swiftech/swiftmarker
- stuartdd https://github.com/stuartdd/template
- zazl https://github.com/zazl/zazl
- josson https://github.com/octomix/josson
- luther94 https://github.com/luther94/templateEngine
- Yan https://github.com/GmailYan/TemplateEngine
- Holdbelief https://github.com/holdbelief/TemplateEngine
- buckhx https://github.com/buckhx/TemplateEngine
- kettlescott https://github.com/kettlescott/TemplateEngine
- grub https://github.com/prezi/grub
- twig4j https://github.com/palmenhq/twig4j-core
- ngoy https://github.com/krizzdewizz/ngoy
- templet https://github.com/toluju/templet
- moennig https://github.com/rnoennig/template-engine
- tempera https://github.com/kasonyang/tempera
- antiaction https://github.com/antiaction/common-template-engine
- templates4j https://github.com/danishdynamite/templates4j
- LagoVista https://github.com/LagoVistaTechLLC/java-template
- maddaluno https://github.com/danielemaddaluno/simple-template-engine
- Debuggable https://github.com/gaelyk/debuggable-template-engine
- Shitstorm https://github.com/hectorlf/shitstorm-template-engine
- Poorman https://github.com/chtz/PoormanTemplateEngine
- Toy https://github.com/CHZone/toy_template_engine
- Tiny https://github.com/ferronrsmith/tiny-template-engine-java
- Albirar https://github.com/albirar/albirar-template-engine
- Quarkus https://github.com/DavutKeskin/Quarkus-Template-Engine
- Tinwatchman https://github.com/tinwatchman/Simple-Java-Template-Engine

- Atian `https://github.com/ate47/AtianTemplateEngine`
- adrianromero `https://github.com/adrianromero/templateengine`
- mentjes `https://github.com/rnentjes/Very-simple-templates`
- round-string `https://github.com/round-lang/round-string`
- templait `https://github.com/seanrmilligan/templait`
- trodix `https://github.com/trodix/teengine`
- nate `https://github.com/SeanTAllen/nate`
- engender `https://github.com/kazimsarikaya/engender`
- ccte `https://github.com/CCLooMi/ccte`
- yyuc `https://github.com/yyuc/StringTemplate`
- hte `https://github.com/HBTGmbH/hte`
- gastirit `https://github.com/gastirit/templateengine`
- jst `https://github.com/hibnico/jst`
- Rocheteau `https://github.com/JeromeRocheteau/templater`
- quick-termplates `https://github.com/rajeshputta1983/quick-templates`
- TEFT `https://github.com/marggx/TEFT`
- stencil `https://github.com/erichonorez/stencil`
- simple-mustache `https://github.com/piotrkot/simple-mustache`
- jetg `https://github.com/fbsgen/jetg`
- yatte `https://github.com/glefur/yatte`
- simple-templates `https://github.com/sbohmann/simple-templates-java`
- Nabucco `https://github.com/NABUCCO/org.nabucco.framework.template`
- kala `https://github.com/Glavo/kala-template`
- scalpel `https://github.com/bluetorch/scalpel`
- JTemplate `https://github.com/And390/JTemplate`
- utl `https://github.com/lamerexter/utl`
- temporize `https://github.com/h34tnet/temporize`
- Webengine `https://github.com/ASofterSpace/WebEngine`

- fronton https://github.com/skozlov/fronton
- Rtpl https://github.com/shenfe/Rtpl
- raptor https://github.com/jebl01/raptor
- djtemplate4j https://github.com/tanob/djtemplate4j
- contemplate https://github.com/almondtools/comtemplate
- templ8 https://github.com/mpobrien/templ8
- jangular https://github.com/cupmanager/jangular
- Hotplate https://github.com/terazzo/Hotplate
- Trensetim https://github.com/trensetim/java-template
- jaml https://github.com/zhuochun/jaml
- jagglate https://github.com/nobuoka/Jagglate
- Simplate https://github.com/SuperPagh/Simplate-engine
- Jango https://github.com/CptPhenomeno/Jango
- BenFaerber https://github.com/benfaerber/WebServer
- libjlte https://github.com/ergor/libjlte
- JTTemplate https://github.com/chen3961/JTTemplate
- minitemplator https://github.com/chdh/minitemplator-java
- omnitemplate https://github.com/omnifaces/omnitemplate
- temmental https://github.com/jfgiraud/temmental
- mistigri https://github.com/jido/mistigri-java
- lostTemple https://github.com/Darkylin/lostTemple
- KDom https://github.com/Kademi/KDom
- Templatefeather https://github.com/aliteralmind/templatefeather
- Javamarkup https://github.com/kmmanoj/JavaMarkup
- cupjate https://github.com/marvinsiq/cupjate
- java-latte https://github.com/peterhalada/java-latte
- jBraces https://github.com/anars/jBraces
- Jinjava https://github.com/HubSpot/jinjava

Appendix B

Source code referenced in the text

B.1 Test code for the Template Engines in the Feasibility Study

Example 'Developer Test' to check that the *TemplateSystem* implementation for a specific template engine compiles and expands a template. Note that this test does not check that the resulting document is fully correct, but only that it generates a document containing appropriate context data without errors or crashes.

```
package test;

import java.util.Arrays;
import org.junit.jupiter.api.Test;
import wrapper.TemplateSystem;
import wrapper.ThymeleafTemplateSystem;

class SingleEngineSmokeTest {
  @Test
  void test() {
    TemplateSystem system = new ThymeleafTemplateSystem();
    system.defineTemplate("smoke",
        "[[# th:each=\"person: *{family}\"] [[${person}]] [/]]");
    system.putContext("family", Arrays.asList(new String[] { "Frank
        ", "Margaret" }));
    CheckResult result = system.check("smoke", 1, "hello Frank", "
        smoke");
    System.out.println(result);
  }
```

```
}
```

Listing B.1: Test code for the Template Engines in the Feasibility Study

B.2 Template Engine Plugin Driver

`init()` requires no parameters but is slightly unusual in that it returns an object of type `TemplateEngine`. In most cases this method simply returns the same `TemplateEngine` object that it was called on, but this approach supports template engines with more complex initialisation requirements which might need to create a new or different `TemplateEngine` object. As a beneficial side-effect, this approach also allows for a 'fluent' style of method calling (Java Design Patterns, 2023) as shown in Listing B.2.

```
    Context context = new MapContext();
    engine.init().expand(context, "example");
```

Listing B.2: Fluent method call

`expand()` requires the context and the name of the template as parameters, and returns a text string containing the resulting document. The definition of the `TemplateEngine` interface is shown in Listing B.3.

```
package shared;
import java.io.IOException;
import com.efsol.context.Context;

public interface TemplateEngine {
    TemplateEngine init() throws IOException;
    String expand(Context context, String template) throws
        IOException;
}
```

Listing B.3: TemplateEngine interface

B.3 Plugin Driver Factory

The Java `interface` definition for these classes contains just two methods, as shown in Listing B.4.

```
package shared;

import java.io.File;
import java.io.IOException;
```

```
public interface EngineFactory {
    TemplateEngine create(File templateFolder) throws IOException;
    String getName();
}
```

Listing B.4: EngineFactory interface

Each plugin contains a class named `plugin.EngineFactory` that implements this interface. As an example, the code for this class in the *Trimou* plugin is shown in Listing B.5.

```
package plugin;

import java.io.File;
import java.io.IOException;
import shared.TemplateEngine;

public class EngineFactory implements shared.EngineFactory {
    @Override
    public TemplateEngine create(File templateFolder) throws
        IOException {
        return new TrimouTemplateEngine(templateFolder).init();
    }

    @Override
    public String getName() {
        return "trimou";
    }
}
```

Listing B.5: Trimou EngineFactoy class

B.4 Performance Test Runner

```
public static void main(String[] args) {
    List<String> real = new ArrayList<>();
    for (String arg : args) {
        if (arg.startsWith("-")) {
            if ("-v".equals(arg)) {
                Run.verbose = true;
            } else if ("-w".equals(arg)) {
                Run.warmup = true;
            }
        } else {
            real.add(arg);
        }
    }
```

```
    int nargs = real.size();
    String engineName = nargs > 0 ? real.get(0) : "dummy";
    String scenario = nargs > 1 ? real.get(1) : "plain";
    int n = nargs > 2 ? Integer.valueOf(real.get(2)) : 1;

    try {
        Run run = new Run(engineName, scenario, n);
        String stamp = format.format(new Date());
        CheckResult result = run.execute();
        if (verbose && result.status() != CheckStatus.OK) {
            System.err.println(result.engine() + "/" + result.test
                () + "\n  actual " + result.actual()
                + "\nexpected " + result.expected());
        }
        System.out.println(stamp + "," + result.engine() + "," +
            result.test() + "," + result.runs() + ","
            + result.time() + "," + result.status());
    } catch (Exception e) {
        System.err.println("ERROR engine:" + engineName + "
            scenario:" + scenario);
        e.printStackTrace();
    }
}
```

Listing B.6: Run class main method

Within the `try..catch` block, after processing the command-line arguments, the test runner calls the constructor method of the `Run` class to create a new object, passing in the template engine name, scenario name, and the number of times to expand the template. The constructor method stores the parameters for later use, applies the `EngineFactory` technique described above to load and initialise the specified template engine driver, and creates a `StopWatch` object to measure the time taken to run the specified number of template expansions. The code for the Run constructor method is shown in Listing B.7.

```
public Run(String engineName, String scenario, int n) throws
    IOException {
    this.engineName = engineName;
    this.scenario = scenario;
    this.n = n;

    this.engine = new EngineFactory().create(new File("../" +
        engineName + "/templates/test/"));
    this.clock = new StopWatch();
}
```

Listing B.7: Run class constructor method

Once the `Run` object is constructed, the `execute` method is called to run the

experiment. This method returns a `CheckResult` object which contains a tuple of (template engine, scenario, number of expansions, time taken, and test status). The test status is determined by whether the result of expanding the template matches the expected result. This result is then emitted as a CSV row which can be appended to a data file for later processing. The code for the `execute` method is shown in Listing B.8.

```
private CheckResult execute() throws IOException {
    TestSpec test = loadTest();
    Tract page = test.getPage();
    String templateName = test.getTemplateName();

    // do one run before starting the clock, to separate one-time
        costs from ongoing ones
    if (warmup) {
        engine.expand(page, templateName);
    }
    clock.reset();
    for (int i = 0; i < n - 1; ++i) {
        engine.expand(page, templateName);
    }
    String result = engine.expand(page, templateName);
    clock.stop();

    return new CheckResult(engineName, scenario, n, clock.get(),
        test.getExpected().equals(result) ? CheckStatus.OK :
            CheckStatus.NOTMATCHED,
        test.getExpected(), result);
}
```

Listing B.8: Run class execute method

Performance Test Scripts

B.4.1 Script `run.sh`

```
#!/bin/bash
engine="${1:-dummy}"
shift
scenario="${1:-plain}"
shift
n="${1:-1}"
shift
#echo run engine=${engine} scenario=${scenario} n=${n} "$@"
java -classpath "../${engine}/bin:../${engine}/lib"'/*:../shared/
    bin:bin' runner.Run ${engine} ${scenario} ${n} "$@"
```

Listing B.9: Script ‘run.sh’

B.4.2 Script `one.sh`

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
engine="${1:-dummy}"
shift
n="${1:-1}"
shift
#echo run engine=${engine} scenario=${scenario} n=${n} "$@"
for scenario in `ls scenarios`
do
  ${here}/run.sh ${engine} ${scenario} ${n} "$@"
done
```

Listing B.10: Script 'one.sh'

B.4.3 Script `all.sh`

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
n="${1:-1}"
shift
for engine in `find .. -name 'EngineFactory.java' | awk 'FS="/" {
    print $2 }'`
do
if [ ! -f "../${engine}/SKIP" ]
then
  ${here}/one.sh ${engine} ${n} "$@"
fi
done
```

Listing B.11: Script 'all.sh'

B.4.4 Script `fulldata.sh` Wave 1

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
echo stamp,engine,scenario,n,time,status
for (( n=1; n <= 10000; n += 1 ))
do
 ${here}/all.sh ${n} "$@"
done
```

Listing B.12: Script 'fulldata1.sh'

B.4.5 Script `fulldata.sh` Wave 2

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
echo stamp,engine,scenario,n,time,status
${here}/all.sh 1 "$@"
${here}/all.sh 10 "$@"
${here}/all.sh 100 "$@"
${here}/all.sh 1000 "$@"
${here}/all.sh 10000 "$@"
```

Listing B.13: Script ‘fulldata2.sh’

B.4.6 Script `fulldata.sh` Wave 3

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
echo stamp,engine,scenario,n,time,status
${here}/all.sh 1 "$@"
${here}/all.sh 10 "$@"
${here}/all.sh 20 "$@"
${here}/all.sh 30 "$@"
${here}/all.sh 40 "$@"
${here}/all.sh 50 "$@"
${here}/all.sh 60 "$@"
${here}/all.sh 70 "$@"
${here}/all.sh 80 "$@"
${here}/all.sh 90 "$@"

for (( n=100; n <= 10000; n += 100 ))
do
 ${here}/all.sh ${n} "$@"
done
```

Listing B.14: Script ‘fulldata3.sh’

B.4.7 Script `fulldata.sh` Wave 4

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
echo stamp,engine,scenario,n,time,status
${here}/all.sh 1 "$@"
${here}/all.sh 10 "$@"
${here}/all.sh 20 "$@"
${here}/all.sh 30 "$@"
${here}/all.sh 40 "$@"
${here}/all.sh 50 "$@"
${here}/all.sh 60 "$@"
${here}/all.sh 70 "$@"
${here}/all.sh 80 "$@"
```

```
${here}/all.sh 90 "$@"

${here}/all.sh 100 "$@"
${here}/all.sh 200 "$@"
${here}/all.sh 300 "$@"
${here}/all.sh 400 "$@"
${here}/all.sh 500 "$@"
${here}/all.sh 600 "$@"
${here}/all.sh 700 "$@"
${here}/all.sh 800 "$@"
${here}/all.sh 900 "$@"

${here}/all.sh 1000 "$@"
${here}/all.sh 2000 "$@"
${here}/all.sh 3000 "$@"
${here}/all.sh 4000 "$@"
${here}/all.sh 5000 "$@"
${here}/all.sh 6000 "$@"
${here}/all.sh 7000 "$@"
${here}/all.sh 8000 "$@"
${here}/all.sh 9000 "$@"

${here}/all.sh 10000 "$@"
```

Listing B.15: Script ‘fulldata4.sh’

B.4.8 Script `fulldata.sh` Modified for *Solomon*

```
#!/bin/bash
here="$(dirname "$(realpath "$0")")"
echo stamp,engine,scenario,n,time,status
for (( n=1; n <= 10000; n += 1 ))
do
 ${here}/one.sh 'solomon' ${n} "$@"
done
```

Listing B.16: Modified ‘fulldata1.sh’

B.5 Code Improvements

B.5.1 Original *Hapax* Driver

```
for (String key : ContextUtils.iterable(context)) {
    Object value = context.getObject(key);
    putContext(key, value);
}
```

Listing B.17: Original *Hapax* Context Loop

B.5.2 Improved *Hapax* Driver

```
        dict = TemplateDictionary.create();
        for (String key : ContextUtils.iterable(context)) {
            Object value = context.getObject(key);
            putContext(key, value);
        }
```

Listing B.18: Updated *Hapax* Context Loop

B.5.3 Original *Stringtemplate* Driver

```
public String expand(Context context, String templateName) {
    String template = templates.get(templateName);
    ST engine = new ST(template);
    for (String key : ContextUtils.iterable(context)) {
        Object value = context.getObject(key);
        engine.add(key, value);
    }
    return engine.render();
}
```

Listing B.19: Original *Stringtemplate* expand method

B.5.4 Improved *Stringtemplate* Driver

```
public String expand(Context context, String templateName) throws
    IOException {
    StringTemplate template = group.getInstanceOf(templateName);
    for (String key : ContextUtils.iterable(context)) {
        Object value = context.getObject(key);
        template.setAttribute(key, value);
    }
    return template.toString();
}
```

Listing B.20: Updated *Stringtemplate* expand method

B.5.5 Example TDD tests for *GILT* template generator

```
package test;

import static org.junit.jupiter.api.Assertions.*;

import java.io.IOException;

import org.junit.jupiter.api.Test;
```

```
import com.efsol.source.MemoryTractSource;
import com.efsol.source.WritableTractSource;
import com.efsol.templates.TemplateCompiler;
import com.efsol.templates.TemplateDriver;
import com.efsol.tract.Tract;

import shared.DriverFactory;
import test.helper.FakeSolomonDriverFactory;

public class TemplateGeneratorTest {
  DriverFactory factory = new FakeSolomonDriverFactory();
  TemplateDriver driver;
  WritableTractSource templates;

  @Test
  void testEmpty() throws IOException {
    build("empty", "");
    assertGenerated("empty", "");
  }

  @Test
  void testBoilerplate() throws IOException {
    build("boilerplate", "hello there");
    assertGenerated("boilerplate", "hello there");
  }

  @Test
  void testLiteral() throws IOException {
    build("literal", "¬{\"hello there\"}");
    assertGenerated("literal", "hello there");
  }

  @Test
  void testDirectLookup() throws IOException {
    build("lookup", "(*¬*){lookup \"wibble\"}");
    assertGenerated("lookup", "${wibble}");
  }

  @Test
  void testDirectInclude() throws IOException {
    build("include", "(*¬*){template \"wibble\"}");
    assertGenerated("include", "${*wibble}");
  }

  @Test
  void testUnrelatedBlock() throws IOException {
    build("main", "hello(*¬*){start \"wibble\"}secret¬{end \"wibble
        \"} there");
    assertGenerated("main", "hello there");
```

```
  assertGenerated("wibble", "secret");
}

@Test
void testBlockReference() throws IOException {
  build("main", "hello ¬{block \"name\"} ¬{block \"name\"}¬{start
      \"name\"}Laa¬{end \"name\"}");
  assertGenerated("main", "hello ${*name} ${*name}");
  assertGenerated("name", "Laa");
}

@Test
void testIfThenElseWithLiterals() throws IOException {
  build("main", "¬{if lookup \"wibble\" then \"correct\" else \"
      incorrect\"}");
  assertGenerated("main", "${wibble?'correct':'incorrect'}");
}

@Test
void testIfThenElseWithLookups() throws IOException {
  build("main", "¬{if lookup \"wibble\" then lookup \"correct\"
      else lookup \"incorrect\"}");
  assertGenerated("main", "${wibble?correct:incorrect}");
}

@Test
void testIfThenElseWithTemplates() throws IOException {
  build("main", "¬{if lookup \"wibble\" then template \"correct\"
       else template \"incorrect\"}");
  assertGenerated("main", "${wibble?*correct:*incorrect}");
}

@Test
void testIfThenWithBlock() throws IOException {
  build("main", "¬{if lookup \"wibble\" then block \"correct\"}¬{
      start \"correct\"}secret¬{end \"correct\"}");
  assertGenerated("correct", "secret");
  assertGenerated("main", "${wibble?*correct}");
}

@Test
void testIfElseWithBlock() throws IOException {
  build("main", "¬{if lookup \"wibble\" else block \"incorrect\"}
      ¬{start \"incorrect\"}herring¬{end \"incorrect\"}");
  assertGenerated("incorrect", "herring");
  assertGenerated("main", "${wibble:*incorrect}");
}

@Test
void testIfThenElseWithBlocks() throws IOException {
```

```
  build("main", "¬{if lookup \"wibble\" then block \"correct\"
      else block \"incorrect\"}¬{start \"correct\"}secret¬{end \"
      correct\"}¬{start \"incorrect\"}herring¬{end \"incorrect
      \"}");
  assertGenerated("correct", "secret");
  assertGenerated("incorrect", "herring");
  assertGenerated("main", "${wibble?*correct:*incorrect}");
}

@Test
void testCallWitnNoParameters() throws IOException {
  build("main", "¬{lookup \"person\" method \"getName\"}");
  assertGenerated("main", "${person.getName()}");
}

@Test
void testCallWitnOneLiteralParameter() throws IOException {
  build("main", "¬{lookup \"person\" method \"get\" parameter \"
      id\" end}");
  assertGenerated("main", "${person.get(id)}");
}

@Test
void testCallWitnOneLookupParameter() throws IOException {
  build("main", "¬{lookup \"person\" method \"get\" parameter
      lookup \"id\" end}");
  assertGenerated("main", "${person.get(${id})}");
}

@Test
void testLoopWithALlDefaults() throws IOException {
  build("main", "¬{foreach lookup \"people\"}");
  assertGenerated("main", "${people}");
}

@Test
void testLoopWithDoTemplate() throws IOException {
  build("main", "¬{foreach lookup \"people\" do template \"person
      \"}");
  assertGenerated("main", "${people*person}");
}

@Test
void testLoopWithDoBlock() throws IOException {
  build("main", "¬{foreach lookup \"people\" do block \"person\"}
      ¬{start \"person\"}¬{lookup \"this\"}¬{end \"person\"}");
  assertGenerated("main", "${people*person}");
  assertGenerated("person", "${this}");
}
```

```
  @Test
  void testLoopWithDoTemplateAndSeparator() throws IOException {
    build("main", "¬{foreach lookup \"people\" do template \"person
        \" separator \",\"}");
    assertGenerated("main", "${people*person/',’}");
  }


  private void assertGenerated(String rootname, String expected)
      throws IOException {
    Tract tract = templates.getTract(rootname);
    assertNotNull(tract);
    assertEquals(expected, tract.getText());
  }

  private void build(String rootname, String input) throws
      IOException {
    templates = new MemoryTractSource();
    driver = factory.create();
    new TemplateCompiler(driver).compile(rootname, input, templates
        );
  }
}
```

Listing B.21: Example TDD tests for *GILT* template generator

Appendix C

Template engine information sources

https://en.wikipedia.org/wiki/Comparison_of_web_template_engines
https://handwiki.org/wiki/Comparison_of_web_template_engines
https://mvnrepository.com/search?q=template+engine

Appendix D

SQL Creation Script for the LOG Database

The following SQL creation script is run when the LOG database is created or to clear all data and reset the system.

Note that the session table is named `session2`. There was a table named `session` in an earlier version of the LOG database but this did not contain enough columns so a new table was created. The old `session` table is no longer used.

```
DROP TABLE IF EXISTS log;
CREATE TABLE log (
    t timestamptz,
    v double precision,
    i double precision,
    PRIMARY KEY(t)
);

DROP TABLE IF EXISTS session2;
CREATE TABLE session2 (
    scenario varchar(32),
    session varchar(32),
    status char,
    start timestamp with time zone,
    stop timestamp with time zone,
    base_start timestamp with time zone,
    base_stop timestamp with time zone,
    avg_baseline decimal,
    avg_active decimal,
    total_active decimal,
    extra_active decimal,
    description text,
    PRIMARY KEY(scenario,session)
);

GRANT ALL ON log TO logger;
GRANT ALL ON session2 TO logger;
```

Appendix E

GILT: A Generic Intermediate Language for Templates

E.1 Introduction and Scope

The diversity of existing templating software implementations and template languages means that templates and their associated template engines are not easily interchangeable during the development of software applications. Committing to a template engine and its associated template language is almost always a key architectural decision for a project. Changing that decision later would not only require the developers to make changes to the code that invokes the template engine, but also require potentially complex changes to every template used for every page or document, which might in turn require assistance from non-developers such as copywriters or graphic designers and then require extensive re-testing to ensure that the visible results are the same.

During the performance comparison of the template engine components in Chapter 4 it quickly became apparent that the wide variety of template languages added extra complexity to the process. Even where the template languages were relatively similar, such as *Trimou* and *Mustachej*, there were still details that meant that they could not use identical templates. Manually creating and testing an individual template for each combination of scenario and template engine is possible with a small set of scenarios and template engines, but it does not scale easily to larger numbers of either. A single new template engine would require the definition of additional templates for every possible scenario. A single new scenario would require new templates for that scenario for every supported template engine.

What was needed was a way to reduce this combinatorial increase. Ideally, a new template engine should only require the addition of a single driver for that template engine, and a new scenario should only require the creation of a single new template.

The literature and software searches in Section 2.14 and Section 4.1.1 revealed a lack of existing solutions to the problems described above. The software search revealed a continuing proliferation of disparate template languages and implementations with no sign of consensus. The small amount of literature that attempted to actively compare template engines did so by either limiting the comparison to engines with compatible template languages or manually creating templates for each supported template language (Laakso and Niemi, 2008)(Zoio, 2005).

This appendix addresses the lack of software available to enable interchangeability of template engines by providing an intermediate representation in which to express templates and an extendable software tool to automatically translate templates expressed in this intermediate representation into a range of template languages for different template engines. The *GILT* system developed and evaluated in this appendix is then used to enable research on the energy usage of template engines in context in Section 6.4.

The intermediate representation and template translation software can be used by software development teams both to compare candidate template engines during system design and to enable changing template engines during later development. This can help avoid a potentially costly commitment to a particular template technology.

The research in this appendix contributes to current knowledge in the following ways:

- It defines and evaluates an intermediate format for expressing templates in a way that can be translated into a range of template languages.
- It develops and evaluates an extensible tool to translate templates expressed in the intermediate language to template languages defined by *plugins* that extend the tool.
- It develops and verifies such plugins for the template engines that are evaluated in context in Section 6.4.

E.2 Software Development Methodology

To ensure that the software produced for this research was correct and appropriate, it was important to select a suitable development methodology. As this research was exploratory in nature, it was decided not to use a rigid development methodology with a fixed plan and sequential phases. Instead, an iterative, incremental, design and development approach was needed. Typically, such development methodologies are described as "agile", as clarified in the *Agile Manifesto* (Beck et al., 2001).

In particular, the agile methodology chosen for this project was Test-Driven Development (TDD) (Beck, 2000b)(Koskela, 2007). TDD is an incremental software development methodology in which development progresses through a series of cycles, with each cycle increasing the demonstrable and verified capabilities of the system. In more rigid, sequential, software methodologies, it is common to spend significant amounts of time gathering requirements, planning architectures, and documenting designs before coding even begins. Such projects commonly treat testing and verification as a stage to be performed after coding has completed, leading to even longer delays and the potential of significant re-work if errors are discovered. The aim of TDD is to eliminate (or greatly reduce) the delay between the conception of a software development project and useful software.

The stages of each TDD development cycle are as follows:

1. Select a new feature or improvement from a pool of requirements (sometimes also called "stories")
2. Encode the desired behaviour of the new feature as one or more executable developer tests. These tests form the dynamic acceptance criteria for the new feature.
3. Add the new tests to the existing collection of such tests, which specify the complete behaviour of the system so far.
4. Run the full suite of tests. All existing tests should still pass. The new tests should fail because the new feature has not yet been added.
5. Implement the new feature until all the tests pass. Run the full suite of tests frequently to ensure that the changes for the new feature have not caused regressions in any previous features.

6. Perform refactoring on the complete codebase to keep it as simple and maintainable as possible, for example, by removing duplication, while still passing the full suite of tests.

The intent of TDD is that each development cycle should be short, with new developer tests added and changes implemented several times per developer per day. Each developer test should be as short as possible, testing just one aspect of the desired system. Software projects developed using TDD can easily accumulate hundreds, or even thousands of such tests, leading to very high levels of test coverage and subsequent confidence in the functioning of the system.

A subset of the TDD test suite for the *GILT* software is given as an example in Listing B.21.

At the end of each cycle, the application will have provably gained a new feature or improvement together with an associated set of automated regression tests which can be run at any point to prove that this feature and all previous features are still working correctly. Inclusion of the refactoring step helps keep the code maintainable and increases the speed and accuracy of future changes.

A key benefit of the TDD methodology is the reduction in the need for post-delivery testing. The system has already been comprehensively validated as a correct implementation of the accumulated acceptance criteria during development. There is still a possibility that some of the acceptance criteria or their encoding as developer tests may have been incorrect or incomplete, therefore leading to an incorrect implementation. The incremental delivery nature of TDD supports the checking and validation of these kinds of "real world" problem by users many times during the development process. If any such errors are discovered during development, the process to correct them is exactly the same as the regular development cycle: add or correct developer tests to specify the required changes; develop the code until the full test suite passes again; refactor the code as necessary; and deliver the next iteration of the software.

The intermediate representation and software developed in this appendix was designed to support the testing of template engines in context in Section 6.4 and therefore implicitly received further testing during the energy measurements in that chapter. If the *GILT* software were to generate an incorrect template for any of the experiments in Section 6.4 that template would fail to produce the correct output when expanded and therefore fail that experiment.

E.3 Requirements for an Intermediate Representation

The challenges with template engines and template engines described above imply a range of requirements for an intermediate representation capable of representing the differences between template languages in a way that can be used to generate correct templates for any of the template engines in the cohort of this study. The differences between the template languages in this cohort are explored in detail in Section 4.2.

E.3.1 Key features of an Intermediate Representation

E.3.1.1 Support for different character encodings

One function of an intermediate representation is to express *boilerplate* text in a way that may be transferred to the output template without modification. This implies that the template representation should support any character encoding required for output templates.

E.3.1.2 Support for different file formats, file naming schemes, and configurations

As seen in Section 4.2, template languages and template engines use a variety of file formats, naming schemes and configuration methods. To correctly generate the template and configuration files for a range of template engines, an intermediate representation and its associated conversion tools should support any file formats, file naming schemes, and configurations required by the output templates.

E.3.1.3 Support for arbitrary placeholder delimiters, including different delimiter sets for different uses within the same template

As seen in Section 4.2, template languages use a variety of placeholder delimiters and delimiter combinations. To correctly generate templates for a template language, the intermediate representation and its associated conversion tools should support the generation of the delimiter combinations required by the output templates.

E.3.1.4 Support for internal and external control structures

As seen in Section 4.2, template languages implement a variety of internal and external control structures. To correctly generate templates for a template language, the intermediate representation and its associated conversion tools should support the generation of all the forms of control structures required by the output templates.

E.3.1.5 Support for templates with single or multiple files

As seen in Section 4.2, template languages also vary in the way they support templates in single files or split across multiple files. To correctly generate templates for a template language, the intermediate representation and its associated conversion tools should support the generation of templates in single and multiple files.

E.3.2 Goals for an Intermediate Representation

E.3.2.1 Plain text files

For the same reasons that drove the representations of all the template engines in this cohort, it seems that such a representation should be in the form of text files. This allows for easy storage, editing, and sharing using generic text manipulation tools. Such text files will need to contain an expression in a form of template *intermediate language* which is not only a template language itself, but also contains all the information required to transform a document into a template that conforms to any of the supported template languages.

E.3.2.2 Contain everything needed to generate any of the candidate template formats

For an intermediate representation to be useful for the generation of differing template languages, the intermediate representation needs to contain all the information required for the superset of the supported template languages and features.

It is important to note that this goal of containing everything needed to generate

any of the candidate template formats was interpreted in a loose rather than strict sense. The aim of the implemented intermediate language was not that it should immediately support *every* possible feature of *every* template language, but rather that the initial intermediate language should be a basis for further research, able to represent a range of common template scenarios in a way which could be transformed into a correct template for all template languages which support such a scenario.

E.3.2.3 Maximise the readability of the language

During the design of the intermediate representation it was envisaged that templates in the intermediate language would largely be created and managed by hand. This process will be more pleasant and efficient if the format chosen for the intermediate representation is easier to read and understand. When facing choices during development of the intermediate language, solutions which improve the readability of the intermediate representation should be preferred.

E.3.2.4 Minimise the fragility of the language

Fragility of a template language is discussed in Section 4.2.10. In the same way that fragility is a disadvantage for any template language, it is also a disadvantage for an intermediate language used to generate templates. When facing choices during development of the intermediate language, solutions which decrease the fragility of the intermediate representation should be preferred.

E.3.3 Desirable Goals for an Intermediate Representation

E.3.3.1 Minimise energy usage while generating a template from a definition

Although it was envisaged that template generation would occur less often than the use of the generated templates, the template generation process itself still requires computing resources and their associated energy to operate. In accordance with the general objectives of this research, a desirable goal for an intermediate language is that it supports the minimising of energy use during the generation of templates.

E.3.3.2 Minimise the complexity of parsing

An intermediate language which is simpler to parse will, in principle, require fewer resources and energy to process. Extra energy is used, for example, during multiple-pass parsing, during the use of back-tracking, or when maintaining memory-intensive data-structures.

E.3.3.3 Maximise the speed of generating a template from a definition

Other things being equal, a process that completes more quickly will use less energy and resources overall, so an intermediate language which is quicker to process is also a desirable goal.

E.3.3.4 Support comfortable editing on a range of different keyboard layouts

As pointed out by Erz (2023), different keyboard layouts prioritise different characters, requiring multiple-key patterns for characters considered less common in that language. A desirable goal for an intermediate language is that it minimises the use of such difficult-to-access characters.

E.3.3.5 Minimise the opportunity for conflicts between boilerplate text and template placeholders or directives

One aspect of an easy to read template representation is a clear and unambiguous distinction between boilerplate text and the template placeholders and directives which produce the dynamic aspects of the generated templates. Choosing character sequences to introduce placeholders and directives which are rare in common forms of boilerplate text reduces the need to escape such characters in boilerplate text.

E.4 The Intermediate Language

To address the requirements and goals listed above, an intermediate template language was designed, a compiler was implemented for the new language, and the implementation was tested by generating and then checking templates for

each of the supported template languages. The characteristics of the language and a formal BNF specification for the intermediate language are given below.

E.4.1 Delimiters and Placeholders

The initial approach to selecting delimiters to indicate placeholders and control structures was to select a starting character sequence. A character sequence was desired, which was not found in any of the candidate template languages and which was also assumed to be uncommon in popular document formats in the hope that it would rarely need to be escaped in boilerplate text. An examination of the "UK English" keyboard used for development provided the ¬ character, which met these requirements. In common with most other template languages, an additional character was added to this character to make a two-character start delimiter sequence `¬{`. The character sequence to end a placeholder or control structure was less critical, as it only had to avoid conflicts with characters found within such a placeholder or control structure directive. Following the approach taken by several of the other template languages, a single matching "closing" character `}` was chosen for this purpose.

Initial development and testing of intermediate language, parsing, and template generation proceeded using these delimiters. However, as noted in Section 4.2.9, such delimiter character sequences may not meet the requirement of "comfortable editing" on a wide range of international keyboard layouts. The choice of the ¬ character was also arbitrary and not based on rigorous research, so there would always remain the possibility of template contents, which might conflict with this delimiter choice and therefore require prolific escaping for some documents. While writing this dissertation it also became apparent that ¬ is not always a comfortable character to work with in LaTeX, either.

During the design of the intermediate language, clarity and explicitness were prioritised over brevity. In most, but not all, template languages, a placeholder containing just a name is assumed to imply the lookup of that name in the context and render it as a textual value in place of the placeholder. In the interests of regularity and readability, and to support a wider range of names, this behaviour was made explicit, requiring a `lookup` keyword and quoting the name.

A typical value placeholder in the intermediate language might look like the following:

```
¬{lookup "customerid"}
```

Listing E.1: Intermediate language value placeholder

and produce output for *Solomon* such as:

```
${customerid}
```

Listing E.2: Value placeholder output for Solomon

or for Jangod:

```
{{customerid}}
```

Listing E.3: Value placeholder output for Jangod

Note that, as discussed above, for some template languages such as *JTE*, the presence of such a placeholder will also drive the generation of an appropriate section in a template preamble. However, in the initial implementation of the intermediate language and template generator, there are some limitations to this. The *JTE* preamble requires the specification of a type with each predefined context value. In the initial intermediate language, there is no way to specify the type of a context value, so all values are assumed to be text of type `java.lang.String`. Although this is a common type for context values, it is not the only possibility.

As mentioned in Section 4.2.7, some template languages support access to fields or methods of objects retrieved from the template context. In most of the template languages in this cohort which support these features, this is represented by placing a . after the name of the context value, and following that with the name of the field, method or *JavaBeans* accessor. This specific syntax, although popular, cannot be assumed to be present in all cases, so the intermediate language provides separate, more explicit syntax for such cases.

To access a field of a context value, an additional clause is added to `lookup` directive. This clause consists of the keyword `field` followed by the quoted name of the field to be accessed. For example, if there is an object in the template context with the name "customer", the value of its "id" field could be retrieved and rendered using something like:

```
¬{lookup "customer" field "id"}
```

Listing E.4: Intermediate language field placeholder

In template engines which support *JavaBeans*, the same field syntax can be used to access *JavaBeans* properties, although within the object they are implemented as methods. For other methods, a different syntax is provided. Method access is also represented by following the `lookup` directive with at least one additional clause, but can support greater complexity. Method access is indicated by a keyword `method` followed by the quoted name of the method to be invoked. If no further clauses are provided, this represents the invocation of a single method which

takes no parameters. The result of invoking the method will be rendered in the output document as the result of the placeholder. As an example, if the context object named "customer" provides a method `fullName` which returns a textual representation of the full name of the customer as described in Section 4.2.7, then this might be represented in the intermediate language as:

```
¬{lookup "customer" method "fullName"}
```

Listing E.5: Intermediate language method placeholder

Although such simple methods are fairly common in data objects placed in a template context, they are not the only type of method supported by the underlying programming language. In the general case, methods may require parameters. In the initial intermediate language, any such parameters are listed as clauses following the `method` clause. Each parameter is introduced using the keyword `parameter` followed by the value of the parameter to be supplied when the method is invoked in the generated template. As an example, if a product has prices in multiple currencies and provides a method to return the price in a specified currency, this might be represented as:

```
¬{lookup "product" method "price" parameter "GBP"}
```

Listing E.6: Intermediate language method with parameter

In a template language which supports method calls with parameters, such as *Solomon*, this would result in the generated template below.

```
${product.price("GBP")}
```

Listing E.7: Method with parameter output for Solomon

During development of the intermediate language, method access was originally provided using a separate control directive indicated by a `call` keyword, but this separation was discovered to be unnecessary, as all the information required was already present for correct generation of method calls for template languages which support it when using a regular `lookup` placeholder with additional clauses to specify a method name and optional parameters.

If parameters are supplied and the target template language does not support method parameters, then an error will be produced during template generation. If parameters are supplied and either the method to be invoked does not accept parameters, or the number or type of the supplied parameters is incorrect, this cannot be detected during template generation, so no error will be produced until the generated template is expanded.

In the initial implementation of the intermediate language, field and method access is limited to one "level" deep. It is not possible, for example, to access a

field of an object which itself is a field of an object in the template context. There is also a limitation in method call parameters that no explicit type information is available. All supplied literal parameters are treated as text strings. If a particular method requires a parameter of a different type, then it cannot be supplied as a literal parameter value. In principle, typed method parameters could be provided from other sources, such as named context values or the result of other method calls, but the initial design of the intermediate language does not support these options.

The use of an explicit `lookup` keyword allows the intermediate language parser (and thus a human reader) to clearly distinguish between this kind of simple value placeholder and a more complex placeholder such as the inclusion of a sub-template. In the intermediate language, a template inclusion placeholder would be expressed as something like:

```
¬{template "customer"}
```

Listing E.8: Intermediate language template include

This directive would represent the action of locating a template named "customer", expanding it with the current context, and placing the resulting text into the output document.

Control directives such as `lookup` and `template` are stackable in the initial version of the intermediate language, which therefore supports multiple levels of indirection, so a placeholder such as:

```
¬{lookup lookup "customerid"}
```

Listing E.9: Intermediate language indirect value lookup

would represent the action of the template engine fetching a value from the context named "customerid", then using that value as the name of another value to look up from the context. This final value would then be rendered to a textual form and placed in the output document. A potentially more useful variant of this might be as shown below.

```
¬{template lookup "preference"}
```

Listing E.10: Intermediate language indirect template include

This would represent the action of looking up a value from the context named "preference", locating a template named the same as that value, expanding it and including the resulting text in the output document.

Most template engines in this cohort do not support these kinds of operations, however.

E.4.2 Control Structures and Named Blocks

As was observed in Section 4.2.3, templates to be generated from the intermediate language have a mixture of internal and external control structures. It was observed that, in general, template languages with more "wordy" control structures were easier to read, at least for a fluent speaker of English, so the language was implemented with English-like control directives. To simplify parsing of the intermediate language, these control structures were designed as internal structures, so that any text outside the placeholder delimiters could be passed through to the destination template as boilerplate text.

Internal control structures have potential problems with nested control structures and other syntax clashes, and the template generator needed to be able to generate both single- and multiple-file template languages from a single definition. To address this, it was decided to define the "body" of control structures such as loops and if/then conditionals as named "blocks" within the intermediate template definition. Each block is introduced by a `start` control directive and terminated by an `end` directive. These named blocks can appear anywhere in the definition file, and can be re-used by multiple different control structures if required. Nesting conflicts are not an issue. All blocks belong to the same conceptual namespace and there is no need to physically locate one block inside another, although the initial syntax does not prevent such nesting.

As an example, the conditional expression introduced in Section 4.2.3 to show whether a stock item is available or unavailable could be defined as given below.

```
¬{if lookup "stock" then block "in stock" else block "no stock"}
¬{start "in stock"}Available¬{end "in stock"}
¬{start "no stock"}Unavailable¬{end "no stock"}
```

Listing E.11: Intermediate language conditional blocks

The representation has several key features. It starts with a keyword, in this case `if`, to indicate the type of control structure, followed by a value clause indicating how to obtain the discriminating value. In this case the discriminating value is obtained by looking up the name "stock" in the template context. This kind of value expression will probably be the most common, but the intermediate language supports any kind of value expression at this point, including an indirect lookup as described above. Following the value clause are two optional action clauses, introduced by the `then` and `else` keywords. The `then` action will be performed if the discriminating value is *true*, while the `else` action will be performed if the discriminating value is *false*. the exact semantics of *true* and *false* depend on the template engine which interprets the generated template. Both the `then` action and the `else` action may be present or omitted.

An `if` directive with neither is valid but arguably pointless.

Although support for more complex comparisons and arithmetic in discriminating expressions is rare in template languages, some template languages do support a notion of equality. This allows a conditional expression to use a non-Boolean value as a discriminator by comparing it for equality with a supplied value. To support this feature, where present, the intermediate language allows an `is` keyword and a value following the discriminator value expression as shown below.

```
¬{if lookup "stock" is "0" then block "no stock" else block "in
    stock"}
¬{start "in stock"}Available¬{end "in stock"}
¬{start "no stock"}Unavailable¬{end "no stock"}
```

Listing E.12: Intermediate language conditional with equals

As with other such specialist facilities, the use of this feature will cause an error if the destination template language does not support equality checking in a conditional expression.

This style of conditional expression would be able to generate both a single-file representation with external control structures such as *Jangod* or a multi-file representation with internal control structures such as *Solomon*. This representation is noticeably longer than either of those representations, however.

The repetition of the block name in the `end` directive is there so that blocks can be nested, if preferred. If block nesting were to be explicitly prohibited, then this repetition might no longer be necessary.

The action clause in this kind of directive is not limited to rendering a named block. The intermediate language also supports the rendering of a named context value using a `lookup` clause, expanding and rendering separate template using a `template` clause, or the the generation of literal text. In situations such as the available stock scenario above, where the text contains no characters which might conflict with the intermediate language, the `if` directive can be simplified by using quoted literal text rather than named blocks, as shown below.

```
¬{if lookup "stock" then "Available" else "Unavailable"}
```

Listing E.13: Intermediate language conditional literals

Iterating through the contents of a collection has a similar style. An iteration directive starts with an identifying keyword, in this case `foreach` followed by a value clause indicating how to locate the collection to be iterated. Following the value clause, an action clause is introduced by a `do` keyword indicating what should be done with each element of the collection. In most cases, this action clause will be a named block as describe above. In addition to the `do` action clause there is

an optional additional action clause, introduced by the keyword `separator` which indicates the action to be performed between the rendering of the elements in the list. For simple cases this might just be a quoted literal, but it could also be a context value lookup, a template, or a named block. An example might be:

```
¬{foreach lookup "prices" do template "price" separator ","}
```

Listing E.14: Intermediate language loop example

E.4.3 Symbols and References

There is an additional category of placeholder that is in some ways similar to a `lookup` placeholder but serves a different purpose. Most template languages support iterating through a collection of items, but to do that they need some way to represent a reference to the "loop index" which contains the current item of the loop. The syntax for this varies widely between template engines. Some template engines "push" the loop index into the template context and use a regular context value placeholder to retrieve the value inside the repeated template block. Other template languages have invented a separate syntax just for this purpose. Some template languages allow a template author to specify the name to be used for this loop index (see the `with` keyword in Control Structures, above), while others have a fixed syntax which must always be used. Listing E.15 and Listing E.16 show template loop fragments, for *Freemarker* and *Hapax*, respectively, generated as part Section 6.4 *Measuring Template Engines in Context*.

```
<#list tags as this><meta property="article:tag" content="${(this)
    !}">
```

Listing E.15: Loop index reference example in *Freemarker* syntax

```
<meta property="article:tag" content="{{<>}}">
```

Listing E.16: Loop index reference example in *Hapax* syntax

Although the template generator can understand the different needs of each supported template language, that is not necessarily true for the person who writes the contents of the named block. If the block author were to, for example, assume that the loop index is always available in a certain context value, and use a `lookup` placeholder, the result would be an intermediate template which could only be used to generate templates for template engines which honoured that expectation, and would not make sense when used with other template languages. This would defeat the primary point of the intermediate language, which is to be able to generate templates for all supported template languages from a single intermediate template.

As the implementation of this concept varies so much, it is not possible to simulate this with any of the other placeholders or control structures already present in the intermediate language. To make use of this loop index and potentially other such symbolic concepts, the intermediate language contains a special kind of placeholder, introduced with a `reference` keyword. A `reference` placeholder is used in a template block. The reference placeholder used to generate Listing E.15 and Listing E.16 is given in Listing E.17.

```
<meta property="article:tag" content="¬{reference "this"}">
```

Listing E.17: Intermediate language symbolic reference

The use of a reference placeholder allows a name to be defined for the loop index using the `with` keyword, and referenced in the block containing the loop body, in a way that can be rendered as appropriate to the destination target language. For example, an intermediate template could be rendered in multiple ways.

```
¬{foreach lookup "family" with "name" do block "person"}
¬{start "person"}hello ¬{reference "name"} ¬{end "person"}
```

Listing E.18: Intermediate language symbolic name

In *velocity*, which does support named loop index variables, this might be rendered as:

```
#foreach($name in $family)hello $name #end
```

Listing E.19: Symbolic reference in Velocity syntax

In *Trimou*, however, which does not support named loop index variables, this might be rendered as:

```
{{#family}}hello {{.}} {{/family}}
```

Listing E.20: Symbolic reference in Trimou syntax

As mentioned above, the `with` keyword is optional. If no name for the loop index is specified the template generator will use the default name `this`. In this case, any reference placeholders within the loop block do not need a name. So, for example, the intermediate template fragment below:

```
¬{foreach lookup "family" do block "person"}
¬{start "person"}hello ¬{reference} ¬{end "in stock"}
```

Listing E.21: Intermediate language anonymous reference

would generate the velocity template shown below.

```
#foreach($this in $family)hello $this #end
```

Listing E.22: Anonymous reference in Velocity syntax

The `with` and `reference` syntax is always available if the default name `this` conflicts with another name or context value, or with a keyword in a generated language.

E.5 BNF Grammar for the Intermediate Language

⟨*template*⟩ ::= ⟨*boilerplate*⟩ [⟨*template*⟩]
| ⟨*placeholder*⟩ [⟨*template*⟩]

⟨*boilerplate*⟩ ::= *any sequence of characters not containing* '`¬{`'

⟨*placeholder*⟩ ::= '`¬{`' ⟨*action*⟩ '`}`'

⟨*action*⟩ ::= '`if`' ⟨*if-clause*⟩ ['`then`' ⟨*replacement*⟩] ['`else`' ⟨*replacement*⟩]
| '`foreach`' ⟨*value*⟩ ['`with`' ⟨*literal*⟩] [⟨*do-clause*⟩ [⟨*sep-clause*⟩] | ⟨*sep-clause*⟩ [⟨*do-clause*⟩]]
| '`start`' ⟨*literal*⟩
| '`end`' ⟨*literal*⟩
| ⟨*replacement*⟩

⟨*replacement*⟩ ::= '`block`' ⟨*literal*⟩ | '`template`' ⟨*value*⟩ | ⟨*value*⟩

⟨*value*⟩ ::= '`lookup`' ⟨*value*⟩ ['`field`' ⟨*literal*⟩ | '`method`' ⟨*literal*⟩ [⟨*param-clause*⟩] end]
| '`reference`' ⟨*literal*⟩
| ⟨*literal*⟩

⟨*if-clause*⟩ ::= ⟨*value*⟩ ['`is`' ⟨*value*⟩]

⟨*do-clause*⟩ ::= '`do`' ⟨*replacement*⟩

⟨*sep-clause*⟩ ::= '`separate`' ⟨*replacement*⟩

⟨*param-clause*⟩ ::= '`parameter`' value [⟨*param-clause*⟩]

⟨*literal*⟩ ::= '`"`' *any sequence of characters not containing an unescaped quote character* '`"`'

E.6 The Intermediate Language Compiler

The intermediate language has been designed to be relatively simple and predictable to parse. All language constructs are enclosed within clearly-delimited placeholders. No parsing, other than to recognise or escape the start of a placeholder block, is required outside such blocks, so boilerplate text text can be efficiently passed on to the generated template with little parsing overhead. The language syntax does not support nesting of placeholders, so there is no need for lexical state beyond entering and leaving a placeholder, quoting and escaping.

Keywords were chosen to be both meaningful (to an English-speaking template author) and to each have a distinct first character. All non-keyword text in a placeholder expression must be quoted. This enables keywords to be unambiguously recognised, regardless of where they appear in a placeholder expression. This decreases compiler complexity by greatly reducing the need for "look-ahead" and "roll-back" when parsing and provides context for more informative error messages in the case of invalid placeholder expressions.

For efficiency, the intermediate language has been designed so that no tokenisation (beyond the separation of boilerplate text from placeholder contents performed during the lexical analysis phase) is required. Parsing is performed directly at the character level as described in Section E.6.4 according to the BNF in Section E.5. Each keyword node type is uniquely specified by the combination of the parsing state and the initial character of the keyword name (given in Table E.1). The language has no variables, and literals are directly compiled into `Text` nodes when an initial *quote* (`"`) character is detected in a placeholder.

The language design and the streamlined compilation process were chosen to align with the goal of minimising the complexity of parsing described in Section E.3.3.2.

E.6.1 Lexical Analysis

The initial task of a compiler for the intermediate language is to distinguish placeholders from the surrounding boilerplate text. In a traditional programming language compiler this would be broadly equivalent to a lexical analysis phase. As described above and in the formal description in Section E.5, placeholders in the initial intermediate language are indicated by a two character starting sequence `¬{` and a single character ending sequence `}`. The aspect of the compiler dedicated to recognising placeholders is conceptually implemented as a character stream state machine with three states:

- "OUTSIDE" indicating that the character currently being processed is part of the boilerplate text
- "PILOT" indicating that an initial `¬` has been detected
- "INSIDE" indicating that the full starting sequence has been detected and the character is therefore inside a placeholder

The state machine starts in the OUTSIDE state, and transitions to PILOT on receiving a ¬ character. All other characters received in OUTSIDE state are considered to be boilerplate text and the state machine remains in the OUTSIDE state. The PILOT state has two possible transitions. A { character completes the starting sequence and the state machine moves to the INSIDE state. Any other character indicates that the previously received ¬ was not part of a placeholder starting sequence, so the state machine returns to the OUTSIDE state. When in the INSIDE state, all characters other than } are considered part of the placeholder, so the state remains at INSIDE. Receipt of a } character transitions the state machine back to the OUTSIDE state.

This state machine is complicated slightly by the treatment of special characters. Although probably uncommon, it is possible for the closing } character to appear in a quoted string within a placeholder. In this case it should not be recognised as the closing character of the placeholder, but as part of the quoted text. For this reason, the state machine has an additional state "QUOTED". When the state machine is in the INSIDE state, receipt of a quote character " transitions the state machine into the QUOTED state until another " character moves it back to the INSIDE state. Similarly, there may be occasions in which significant characters ({ following a ¬ in the OUTSIDE state and " in the QUOTED state) need to be *escaped* to enable them to be treated as regular text. This is supported in the intermediate language using the *backslash* character (\). This introduces two further states to the state machine "OUTSIDE-ESCAPED" and "QUOTED-ESCAPED". When in the OUTSIDE state or the PILOT state, receipt of a \ character transitions to the OUTSIDE-ESCAPED state. In the OUTSIDE-ESCAPED state, any subsequent character transitions back to the OUTSIDE state. When in the QUOTED state, the process is similar. Receipt of a \ character transitions to the QUOTED-ESCAPED state and in the QUOTED-ESCAPED state, any subsequent character transitions back to the QUOTED state. There is no need for escaping in the INSIDE state as all characters are significant and distinct in placeholders. The complete state machine is shown in Figure E.6.1.

E.6.2 Nodes and Node Types

Although an intermediate template document is, from one viewpoint, just a sequence of characters, this is not a very useful abstraction for understanding the intent of the template and converting it to another form. The *GILT* software performs template generation in two phases. The first phase is to compile the input document into a symbolic object model for later use. This compiled model is then available for the second phase in which it can be traversed (in conjunction

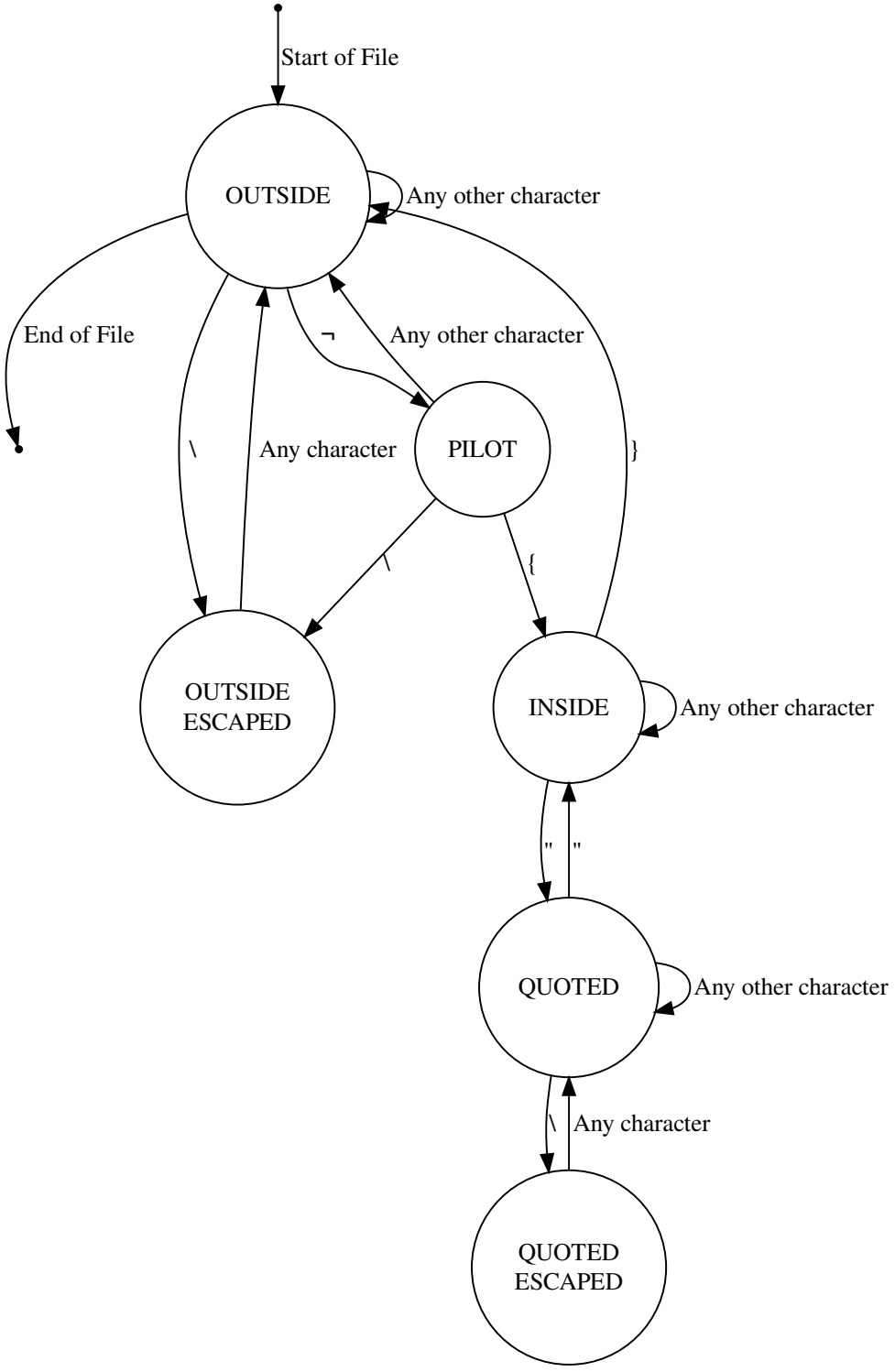


Figure E.6.1: Placeholder state machine

with one or more provided template engine drivers) to generate output templates in supported template languages.

For efficiency, the compiled model is accessed directly in memory by the template

engine drivers rather than writing it out in any form of object file, as this would then require re-reading and processing before use when generating templates. The *GILT* software runs as a single process so it can*appear* as if it is performing as an interpreter - processing template text and directives as they are read. However, it uses distinct compilation and template generation phases internally to deal with the non-linear nature of templates containing optional or re-usable blocks.

The compiled model is represented by a broad-based parse tree structure of "nodes". Each node is either a leaf node (such as some boilerplate text) or a subtree that represents a more complex part of the input document (such as a placeholder or a directive). The Nodes have been chosen, wherever possible, to align with the key concepts of template languages. Each node, once compiled, should contain enough information to enable it to be rendered into an equivalent concept in the target template language.

Compiled nodes may include other nodes "within" them. For example, a `Conditional` node requires a *discriminator* value, which will often be a `Lookup` node. The action or actions associated with the `Conditional` node might be other nodes such as a `Text` node representing literal text, a `Block Reference` node with the name of a block to render, a `Template Inclusion` node, and so on.

The roots of the parse tree are:

- A single sequence of nodes representing the compiled template. This level includes nodes such as `Text` nodes containing boilerplate text, `Lookup` nodes representing values to be looked up from the template context, `Conditional` nodes representing decisions, etc. The nodes in this main sequence may refer to the named blocks mentioned below.
- Zero or more named blocks delimited by `Block Start` and `Block End` nodes. Each of these blocks contains a sequence of nodes, which may also refer to any of the named blocks.

As an example, compiling the *GILT* template fragment shown in Listing E.23 would produce the parse tree shown in Figure E.6.2.

```
¬{if lookup "stock" is "0" then "out of stock" else block "in stock
    "}
¬{start "in stock"}¬{lookup "stock"} units available¬{end "in stock
    "}
```

Listing E.23: *GILT* template example for comparison with generated parse tree

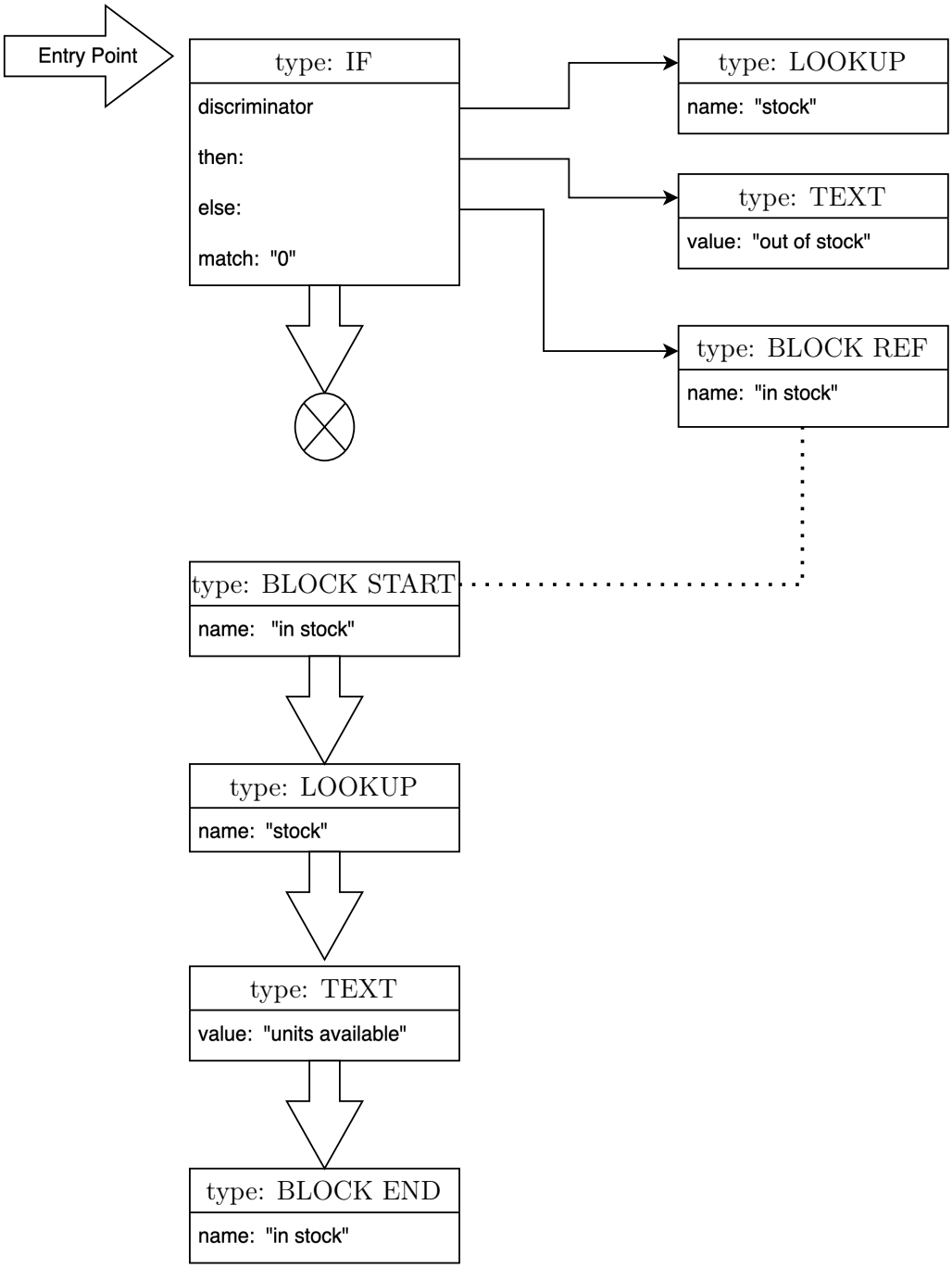


Figure E.6.2: Parse tree generated by compiling the *GILT* template fragment in Listing E.23

Every node implements a single Java interface, shown below.

```
public interface Node {
  static final String VALUE = "value";

  String getType();
  Map<String,Node> getContent();

  default Node get(String key) { return getContent().get(key); }
  default Node getValue() { return get(VALUE); }
  default String getText() { return null; }
}
```

Listing E.24: Parser Node interface

This allows each node to contain a collection of other named nodes, with one given priority as the *value* of this node to simplify code for the common case of boilerplate text and simple lookup or template inclusion nodes. Each node also has a *type* to enable the code in the template generator to decide what to do with the node and how to render it to the destination template language. The template generation code knows nothing about the individual classes which implement this interface, but deals exclusively with the Node interface and its methods.

The various node types produced by the compiler have a lot in common. For ease of implementation, some of these similarities have been gathered into an abstract helper class `AbstractNode`. All the implementing classes extend from this class in order to make use of its facilities. The `AbstractNode` class provides an internal data structure to store the inner nodes as well as methods which the individual Node implementations can use to add inner nodes. In addition, the `AbstractNode` class provides methods for diagnostic output which were used when testing the intermediate language compiler.

The node types produced by this compiler are as follows:

Text node A text node represents a block of plain text. This text may be boilerplate text, found between placeholders or at the start or end of the document, or it may be literal text found within the definition of a placeholder. In either case, the intention is the same. When generating a template in the target template language, a text template will be rendered as text. A text node has one internal value, the text itself.

Block Start node A block start node represents the start of a named block. A block start node has one internal value, the name of the block. A block start node is always a top-level node

Block End node A block end node represents the end of a named block. A block end node has one internal value, the name of the block. A block end node is always a top-level node.

Block Reference node A block reference node represents the use of a named block. A block reference node has one internal value, the name of the block. Although a block reference node can be a top-level node, it is more usual to encounter it as the action associated with another node such as a conditional node or a loop node.

Lookup node A lookup node represents the fetching of a value from the template context. A lookup node has one mandatory internal value, the name of the value to look up. This node can also contain some optional internal values: a field to access, a method to invoke, and a collection of parameters for that method. In the initial implementation of the intermediate language compiler, field and method access are exclusive, so a lookup node can contain a field name, a method name, or neither. Method parameters can only be present if a method name has been specified. The context value name, field name, and method name are all simple text values, but the collection of method parameters can include multiple parameters and those in turn can be literal values or other nodes. The collection of method parameters is represented as an array node (see below).

Template Inclusion node A template inclusion node represents the inclusion of one template by another. The single internal value is a Node representing how to find the name of the template to include. In the initial version of the intermediate language and the compiler this internal value can be either a literal text node or a lookup node. More complex configurations such as including a template whose name is given in a file are not supported.

Conditional node A conditional node represents a decision within the template. A conditional node has one mandatory internal value, a node representing the discriminating value. Although the intermediate language allows this to be a literal, that would usually be a pointless exercise, as the result of the conditional expression would always be the same. For most uses this internal value will be a lookup node representing the context value on which to make the decision.

A conditional node can also have a combination of three optional values. If a value was specified using the `is` keyword then that value will be stored as an internal

value to be matched. If an action was specified using the `then` keyword then that value will be stored as an action to be performed if the discriminating expression is *true*. If an action was specified using the `else` keyword then that value will be stored as an action to be performed if the discriminating expression is *false*. The initial implementation of the compiler does not prevent the declaration of a conditional placeholder with neither a `then` nor an `else` action. If neither are present, then the conditional node will perform no action and produce no output when the destination template is expanded. The nodes representing the actions will usually be block reference nodes, template inclusion nodes, or literal values.

Loop node A loop node represents an iteration placeholder. A loop node has one mandatory internal value, a node representing the collection to be iterated through. While this could potentially be a literal, the initial intermediate language has no syntax to specify multi-value literals, so that is not likely to be a valid type to be iterated in the destination template language. In most cases the node representing the collection will be a lookup node which specifies how to retrieve the collection from the template context. The action to be performed for each element of the collection, specified using the `do` keyword, is theoretically optional, although a loop with no action would generate no output when the destination template is expanded. The node representing this action will usually be a block reference node, a template inclusion node, or a literal value. Two extra internal values are also optional. If a symbol is specified using the `with` keyword it will be stored for use when the destination template is generated. If an element separator is specified using the `separator` it will be also be stored as an internal value. The separator value can be any of the action node types used for `do`, above.

Method Call node A method call node represents a method call placeholder. Although this type of node was supported in the initial intermediate language, and thus the initial compiler and generator implementations, it was later subsumed into the lookup node, In the initial implementation it had two mandatory internal values, a lookup node representing the context value on which to invoke the method and a literal value containing the name of the method. As with the method call option of the lookup node, there was also the option to include a collection of parameters. If present, they were stored using an array node.

Array node An array node is an internal node which cannot appear as a top-level node. An array node is created whenever another node needs to contain an unknown number of internal values. Specifically, in the initial implementation of the template language model it is used to contain the list of nodes representing

method parameters in both lookup nodes and method call nodes.

Reference node As discussed above, for the intermediate language to express the notion of a current loop element in a way which is independent of the destination template language, it requires a symbolic way to indicate when the current loop element is to be mentioned during template generation. This is the purpose of the reference node. A reference node is produced when a placeholder which starts with the `reference` keyword is parsed. A reference node has one internal value containing the name of the symbol to be referenced. If no symbol name is specified in the reference placeholder, the default value of `this` is used.

E.6.3 Character Buffering

As discussed above, the aim of the intermediate language compiler is to convert an incoming template into a tree of nodes representing the parsed components of the template. Each node typically represents a block of characters from the input template. Each chunk of boilerplate text is represented as a single node, regardless of the length of the chunk. Likewise, each placeholder is represented as a single node with internal values indicating the type of placeholder and the combination of directives and values within it. In both these cases, the content of the node to be constructed is not known until the end is reached. The end of a chunk of boilerplate text is indicated either by the end of the input or by the start of a placeholder. The end of a placeholder is indicated by the terminating `}` character.

The intermediate language compiler includes a buffer to collect the text which will ultimately be converted to a node of one sort or another. From the start of a chunk of boilerplate text, or the start of a placeholder, incoming characters are accumulated into a buffer. At the end, the contents of the buffer are parsed to a node. Accumulated boilerplate text is directly converted to a text node. Accumulated placeholder text is passed to a separate `PlaceholderParser` object which processes the contents of the buffer according to the grammar of placeholder contents. When the node has been created, the buffer is cleared ready to begin collecting characters for the next node.

Adding text to the buffer normally happens on the "any other character" transitions when the state of the state machine (INSIDE or OUTSIDE) remains the same. Special characters such as placeholder delimiters are not usually

added to the buffer, as they serve to indicate the start or end of a node. However, here are a few situations in which special characters are added to the buffer. Any *escaped* characters are added to the buffer, although the escape character (\) is not. Quote characters (") in placeholders change the state for the purposes of parsing, but are added to the buffer along with the rest of the placeholder characters. The most complex case is if a "pilot" character (¬) is encountered but it is not followed by the `{` that would complete the start of a placeholder (or there is a `{`, but it is escaped). In this case, both the pilot character and the current character need to be added to the buffer. This is the only case in which the compiler needs to "undo" a decision while parsing.

E.6.4 Parsing of Placeholders

Once the enclosing ¬{ and } have been removed, each placeholder consists of a sequence of characters which need to be parsed according to the placeholder grammar to determine which specific node or nodes to create. A formal specification for the placeholder grammar is given in Section E.5. Within a placeholder, unquoted whitespace characters are not significant and are removed as the placeholder contents are parsed. In the following discussion of the parsing process, such whitespace will be ignored.

As mentioned above, the keywords in the intermediate language have been carefully chosen so that they all have distinct starting letters. The top-level "action" symbols that can start a placeholder and the node classes that result are shown in the table below. A placeholder that starts with any character not in the table is invalid. A parse error will be produced and parsing will stop.

The placeholder parser examines the initial character of the placeholder and calls an appropriate parse method for each placeholder type. This parse method is responsible for parsing the rest of the placeholder and returning a `Node` object representing it. Each method (except for quoted literals which effectively have a single character keyword) begins with a call to a method `expect` which is given the full text of the keyword, for example by calling `expect("lookup")`. This method steps through the incoming character stream checking that the specified characters are present. If the keyword ends prematurely, or contains an unexpected character, then a parse error will be produced and parsing will stop. Once the initial keyword has been verified, the code for the parse method depends on the specific nature of that type of placeholder. The sequence of methods called to parse the remainder of the placeholder follow the grammar.

As a simple example, a block start placeholder is defined in the grammar

Character	Keyword	Description	Node class
		Top-Level Keywords	
i	`if`	conditional	`IfNode`
f	`foreach`	loop	`LoopNode`
s	`start`	block start	`BlockStartNode`
e	`end`	block end	`BlockEndNode`
b	`block`	block reference	`BlockNode`
t	`template`	template inclusion	`TemplateNode`
l	`lookup`	value lookup	`LookupNode`
r	`reference`	symbol reference	`ReferenceNode`
"		*quoted literal*	`TextNode`
		Context-Specific Keywords	
d	`do`	loop body	*various*
e	`else`	action if false	*various*
i	`is`	conditional match value	`TextNode`
f	`field`	lookup field name	`TextNode`
m	`method`	lookup method name	`TextNode`
p	`parameter`	method parameter	*various*
s	`separator`	loop separator	*various*
t	`then`	action if true	*various*
w	`with`	loop variable name	`TextNode`

Table E.1: Keyword Initial Characters

specification as the keyword `start` followed by a literal, so in the method for a block start, the keyword is verified, then the `literal` method is called to parse a literal value. All the parse methods return a `Node` object, including ones which cannot be used at the top-level of a placeholder, so in this case calling the `literal` method results in a `TextNode` object representing the literal text. As the literal value is the only extra node in this case, as soon as the `TextNode` object is returned from the `literal` method it is used as the value when constructing a new `BlockStartNode` which is then returned from the parse method.

Block end and block reference placeholders also require a single literal value following the initial keyword, so the structure of their parse methods is similar to the block start method. Other parse methods are more complex, however, as they have options which need to be correctly recognised and processed. In such cases the process is similar to the process for the "action" symbol used at the top-level of a placeholder. The next character of the input stream is examined and checked against a list of valid options. If a valid initial character is

encountered, then parsing is handed off to the appropriate parse method.

For example, the `conditional` method is used to parse placeholders which start with `if`. The placeholder grammar indicates that the next item must be a "value", so the `value` method is called to return a `Node` object representing that value. After the value, there are several optional items. The next item can be either `is`, `then`, `else`, or nothing at all. A Node variable is prepared in advance for each option and set to `null`, indicating that the particular option has not been provided. If the placeholder ends at this point, then the resulting `IfNode` object will have no internal values for the value to match, the action on true, or the action on false. If the next character in the placeholder is one of `i`, `t`, or `e`, then the process continues with an appropriate parse method, which follows the same pattern as all the parse methods. Call `expect` to verify the full keyword, then combine the mandatory and optional following values into a `Node` object which is then returned. Each returned value is placed into the variable prepared for that purpose. When the placeholder ends, the collected variables are combined into an `IfNode` which is returned from the method.

The `Node` objects resulting from top-level placeholders are then added to the sequence of nodes which represent the template as a whole, alongside any `TextNode` objects resulting from boilerplate text. This sequence of nodes is what is passed to the template generator for rendering into a destination template.

The parsing approach described above correctly implements the placeholder grammar, but that does not mean that all possible combinations of nodes and sub-nodes are useful. For example, as can be seen from the table of initial characters, a placeholder which starts with a double-quote character represents literal text and results in the creation of a `TextNode` object. While this is legitimate according to the placeholder grammar, it is generally not very useful, as the resulting node is indistinguishable to the generator from a section of boilerplate equivalent to the quoted text. Likewise, an `ifNode` with none of the optional values, although valid, represents a template action that examines a value but does nothing with it, so it will not result in any output when the generated destination template is expanded. Depending on the syntax and semantics of the different destination template languages, such pointless constructs may not even be able to generate valid templates.

E.7 The Template Generator

The job of the template generator is to take a template expressed in the intermediate language and convert it, where possible, to a similarly-functioning

template in a destination template language. Conversion is not always possible, however. The intermediate language is designed to support a set of common template language features but these are not supported in all template languages.

There is only one intermediate language, so the template generator creates an instance of the intermediate language compiler which it uses to compile a supplied intermediate template to produce a parse tree of `Node` objects representing the various chunks of boilerplate text and placeholders from the original template. To produce a set of destination templates, the template generator passes the compiled parse tree to a template language driver that understands how to walk the supplied parse tree and calls an appropriate method for each type of `Node` to render that `Node` in a specific destination template language.

A sequence diagram illustrating the interaction between the template generator, *GILT* compiler, the template engine plugin, and the template store during template generation is given in Figure E.7.1. The rendering methods that must be supported by a template driver are shown in Figure E.7.3 and are listed in more detail in Listing E.27.

E.7.1 Template Storage and the Tract class

Different template engines have different requirements for template storage. Some template engines expect a template to be provided as a text string in memory, some require that every template exists in a file, sometimes with a specific filename, location or file type. The Template generator cannot know about the vagaries of every possible template engine, so an abstraction is required to isolate the generation process from such details.

The template generator treats every template as an object which implements the `Tract` interface. The Tract interface provides a small set of methods to access the content (in this case the text of the template) and a collection of name/value properties. In this use of the `Tract` concept, the properties depend on the requirements of the specific template engine. Some template engines will not need properties in addition to the text of the template, while others may require details of, for example, how and where the template should be stored.

When Tract objects need to be retrieved from storage, there is another abstraction `TractSource`. `TractSource` is also an interface which provides a method to retrieve a `Tract` by name, and also some general purpose methods to enquire about the `TractSource` itself, such as whether it is empty or where it is located. By default, a `TractSource` is read-only, but a further interface `WritableTractSource` extends this

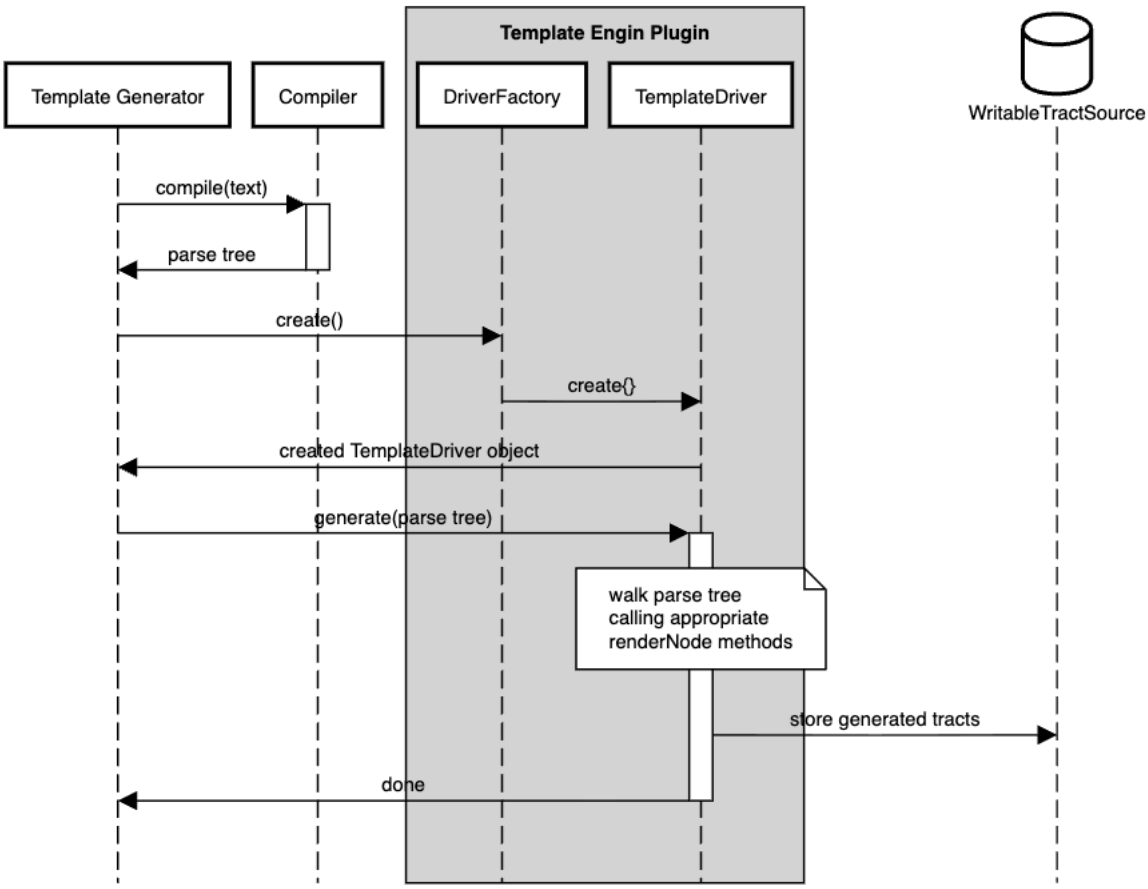


Figure E.7.1: Sequence diagram showing the interaction between the template generator, *GILT* compiler, template engine plugin and tract store during template generation

to support placing a `Tract` into the store as well as removing a stored `Tract` or clearing the store.

`TractSource` (and `WritableTractSource`) implementations can be created to represent the details of how and where templates are stored. The template generator can use the `TractSource` methods on a provided store object with no knowledge of the underlying implementation.

A class diagram illustrating the relationships between the `Context`, `Tract`, and `TractSource` interfaces and their associated concrete classes is given in Figure E.7.2.

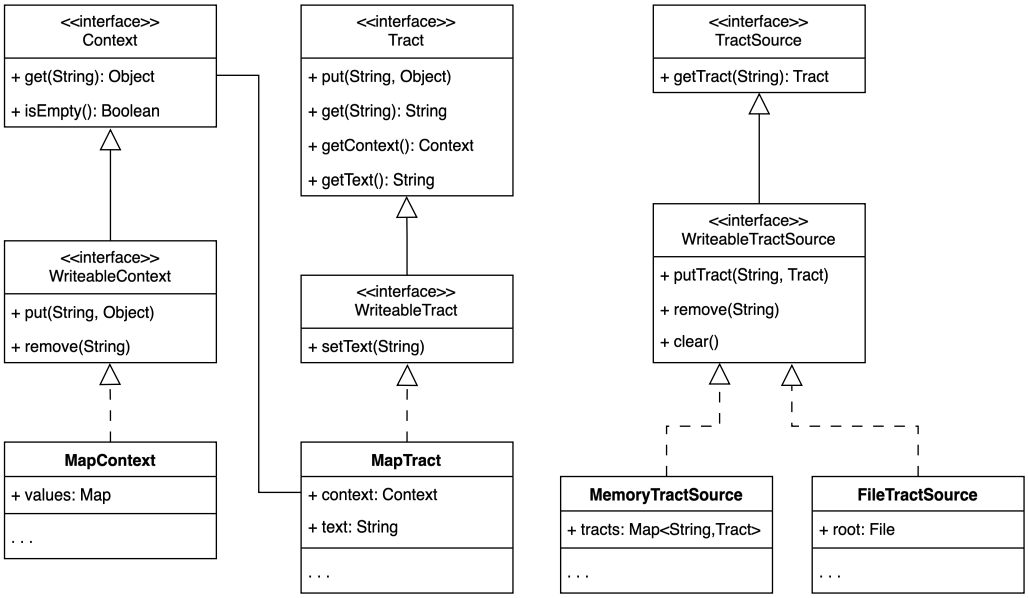


Figure E.7.2: Class diagram illustrating the relationships between the `Context`, `Tract`, and `TractSource` class families

E.7.2 Drivers and Dynamic Loading

Using the same justification as the design of the dynamic loading mechanism for template engine drivers during template engine comparison (as described in Section 4.7.2), each template language driver is implemented as a separate dynamically-loaded, "plugin". Each plugin contains a class with the same fully-qualified name: `plugin.DriverFactory`. This class provides the entry point to the plugin, a `create()` method which returns a template-engine-specific object implementing the `TemplateDriver` interface. This driver object provides methods for rendering the different types of `Node` objects described in Section E.6.2 in a suitable way for a specific template language.

A UML class diagram illustrating the `TemplateDriver` interfaces, classes, and plugins is shown in Figure E.7.3.

The `TemplateDriver`interface implemented by template language drivers contains just two methods, as shown in Listing E.25. As an example of a plugin implementation, the code for this class in the *Trimou* plugin is shown in Listing E.26.

```
package shared;
import java.io.IOException;
import com.efsol.templates.TemplateDriver;

public interface DriverFactory {
  TemplateDriver create() throws IOException;
  String getName();
}
```

Listing E.25: DriverFactory interface

```
package plugin;
import java.io.IOException;
import com.efsol.templates.TemplateDriver;

public class DriverFactory implements shared.DriverFactory {
    @Override
    public TemplateDriver create() throws IOException {
        return new TrimouTemplateDriver();
    }

    @Override
    public String getName() {
        return "trimou";
    }
}
```

Listing E.26: *Trimou* DriverFactory implementation

When generating templates for a particular template language, the name of the plugin to load is specified as a command-line parameter to a script which builds a Java *classpath* containing the template generator and the specified plugin classes.

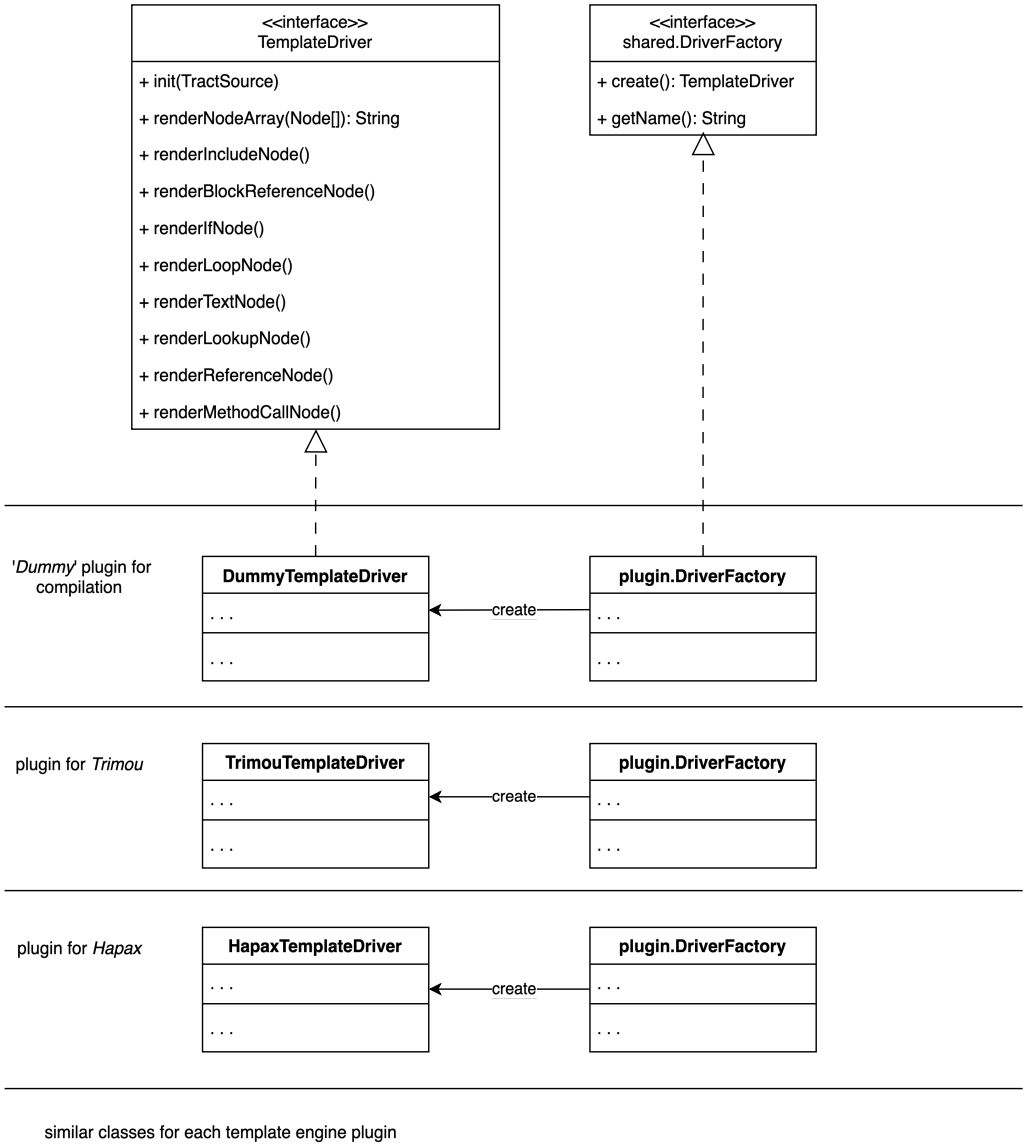


Figure E.7.3: Class diagram for the `TemplateDriver` interfaces, classes, and plugins

E.8 Template Language Drivers

The aim of the design of the intermediate language is that, after parsing the intermediate template, the resulting data structure contains all the information required to generate an equivalent template in each of the supported template languages. Subject, of course, to the capabilities of the individual template languages. In order to generate the destination templates, however, a mapping is required between the collection of symbolic language nodes held in the data structure, and the textual expression in a particular target template language. For this research, this mapping has been defined in the form of a range of software *drivers*, written in Java. A separate driver has been created for each of the template engines in the cohort. All the template language drivers implement the same Java interface definition, which provides methods to render specific node types from the symbolic model.

E.8.1 Driver Interface and Shared Code

In conjunction with the initial intermediate language and its compiler, a Java interface was defined to provide a set of methods which all template language drivers must implement. The code for the interface is given below.

```
public interface TemplateDriver {
  void init(Map<String, Node[]> blocks, WritableTractSource
      templates);
  String renderNodeArray(Node[] nodes) throws IOException;

  void renderIncludeNode(String templateName) throws IOException;
  void renderBlockReferenceNode(String blockName) throws
      IOException;
  void renderIfNode(Node key, Node match, Node iftrue, Node iffalse
      ) throws IOException;
  void renderLoopNode(Node target, Node with, Node body, Node sep)
      throws IOException;
  void renderTextNode(String text) throws IOException;
  void renderLookupNode(Node target, Node field, Node method, Node
      parameters) throws IOException;
  void renderMethodCallNode(Node target, Node method, Node[]
      parameters) throws IOException;
  void renderReferenceNode(Node symbol) throws IOException;

  String quote(String text);
}
```

Listing E.27: TemplateDriver interface

In this interface, there are three groups of methods.

The first group consists of methods related to the whole template generation process.

The `init` method begins the generation process for a new output template. The parameters are a Map of nodes representing the named blocks from the intermediate template and a "place" where the templates are stored. The "place" is a virtual location which provides a basic set of operations and can be implemented in different ways, such as a file store, an in-memory store, or even a remote store in a database or server if required.

The first parameter to this method is used during template generation whenever a control structure in the intermediate template refers to a named block. The contents of the block can be fetched from the supplied Map by name as an array of parse nodes and rendered in whatever manner and to whatever location the particular template language driver requires. The second parameter will be used for two situations. The first is if a `template` directive is encountered, but the destination language does not support template inclusion. In this case, the template generator will need to render the contents of the included template directly into the output template. The second situation in which the template location will be needed is if the destination template language requires the generation of multiple template files rather than including the block contents in-line in the main template. This location is where generated sub-templates will be placed.

The `renderNodeArray` method is used when rendering either a whole template or a sub-template such as a named block. This method is the main entry point into the driver from the template generator code. This is a separate method from the rest of the "render" methods as, depending on the destination template language and whether it is a "top level" template, the result may be included in the generated template, or it may be stored to the template "place" for direct use or for later inclusion when another template is expanded. The `renderNodeArray` method returns the result as a text string that can then be stored or included as required.

The second group of methods are the core methods of the template language driver for each different template language. Each of these methods handles one of the types of nodes produced during parsing and allows the driver to render that type of node in an appropriate manner for that template language. The method names largely follow the node names and the syntax of the intermediate template language.

The `renderIncludeNode` method allows the template language driver to adapt to

the destination syntax for template inclusion and is triggered by the presence of a parse node resulting from a `template` directive.

The `renderBlockReferenceNode` method allows the template language driver to adapt to the destination syntax for sub-templates and is triggered by the presence of a parse node resulting from a `block` directive.

The `renderIfNode` method allows the template language driver to adapt to the destination syntax for conditional control structures and is triggered by the presence of a parse node resulting from an `if` directive.

The `renderLoopNode` method allows the template language driver to adapt to the destination syntax for loop control structures and is triggered by the presence of a parse node resulting from a `loop` directive.

The `renderTextNode` method is triggered by the presence of a parse node resulting from boilerplate text. Note that the parsing process also produces text nodes for literal text within a placeholder but these are handled within the render method for the appropriate placeholder directive.

The `renderLookupNode` method allows the template language driver to adapt to the destination syntax for context value access and is triggered by the presence of a parse node resulting from a `lookup` directive.

The final group of methods enable drivers to adapt to different situations during the processing of nodes.

The `quote` method allows the template language driver to generate quoted and/or escaped text according to the rules for the destination template language.

E.8.2 Commonalities Between Template Languages

While it would be possible to create a template language driver completely from scratch, based solely on the `TemplateDriver` interface, the resulting code would likely have a lot of similarities to other drivers. This is for three main reasons. The first reason is that there is always a certain amount of general processing which will be needed in all implementations. For example, the code to step through the nodes in an array during processing of the `renderNodeArray` method is very likely to be the same for most, if not all, template language driver implementations. The second reason is that, even in areas where template languages differ, they are not entirely distinct from each other. Some template languages will have similar

syntax to others for some operations, or the structure of their syntax will be similar enough that portions of the code to generate the resulting template will be the same. For example in this cohort of template engines, *Handlebars*, *Hapax*, *Jangod*, *Mustachej*, *Pebble*, and *Trimou* all use the same basic syntax for context value placeholders. In this case, the code for value placeholders in all these different driver implementations will probably be functionally identical. A third reason for similarity of code is that the intermediate language is capable of expressing some relatively complex template directives and expressions which only a few template languages support. In these situations, the code for most of the template drivers will consist of an error message indicating that the destination template language cannot support the required construct. If most template languages do not support a particular combination of features, then they could all make use of the same default implementation which produces such an error message.

In cases where there is some similarity between template languages, it can make sense to share code. In the initial implementation of the template generator, all the template language drivers share a base implementation of the `TemplateDriver` interface named `CommonTemplateDriver`. This class defines a default implementation for all of the "render" methods as well as a selection of helper methods to perform operations common to several template language driver implementations.

One of the key methods provided by this class is `renderNode` which examines a supplied node and calls whichever of the other "render" methods is appropriate. This method understands the structure of each of the node types and is able to recognise each node by its `type` field and extract their options and parameters and construct a correct call to one of the specific method to render the node. This method will issue a template generation error if an unknown node type is encountered, but this should never happen unless the parser code is altered without the corresponding changes to `CommonTemplateDriver`. In addition to this basic functionality, the `renderNode` method, along with most methods in the `CommonTemplateDriver` class, provides switchable diagnostics to assist a driver author in verifying and debugging driver behaviour.

Some method implementations will be similar for almost all template language drivers. An example of this is the `renderTextNode` method. All the template languages in this cohort render boilerplate text as-is, so the default implementation is very simple - it just adds the supplied text to the output template. There is a slight complication with this method, though. the specification of the initial intermediate language allows any characters which do not introduce an intermediate language placeholder in the boilerplate text. There is a chance, therefore, that boilerplate text may conflict with the placeholder syntax of the destination template language. In such cases,

individual drivers can override this method to apply language-specific escaping rules to the supplied text before passing it on to the default method for rendering.

In addition to these key rendering methods, the `CommonTemplateDriver` class also provides a range of overridable methods for common functions such as quoting or escaping text, or appending characters to the generated text in a variety of situations. For example, some template languages require the names of context values included templates to be quoted while others do not. Likewise, some template languages support literal text in place of context values, but these need to be quoted, and perhaps in a different way to template or context value names. The parsed node list resulting from parsing an intermediate template in the intermediate language contains a range of types of nodes including the various kinds of template directives and boilerplate text found outside placeholders. The code to determine when and how to quote or escape the textual values of these nodes could end up repeated in many drivers, so helper methods are provided which deal with common cases. The aim is to provide enough support that drivers only need to include code which is specific to that particular template language.

In the initial implementation of the template generator there is still work to be done in minimising the code requirements for each driver. There are areas where functionally identical code is included in all template drivers. For example, consider the implementation of the `renderBlockReferenceNode` method from the *Velocity* driver shown in Listing E.28. This implementation contains no code specific to the *Velocity* template language, and similar implementations are found in the drivers for other template languages. This code should therefore be moved to somewhere accessible to all drivers. Examining the template languages in the cohort under consideration, it appears that most use some form of in-line representation for code blocks, so this implementation is also a candidate for a default method implementation. This would allow most drivers to just use the default, and only drivers for template languages (such as *Stringtree* and *Solomon*) which use a different approach will need to implement this method.

```
@Override
public void renderBlockReferenceNode(String blockName) throws
    IOException {
    Node[] block = blocks.get(blockName);
    if (null == block) {
        throw new TemplateCompileError("attempt to reference
            unknown block '" + blockName + "'");
    }
    renderNodeArray(block);
}
```

Listing E.28: *Velocity* implementation of renderBlockReferenceNode

E.9 Testing Template Generation

This template generation software consists of three key components: the compiler for the intermediate language; the template generator, and the individual drivers for the different template languages.

Each of these components was developed and tested separately using the *Test-Driven Development* process(Beck, 2000b)(Koskela, 2007) as introduced in Section E.2. This process requires *developer tests*, also known as *unit tests*, to be created and run with the code during development. These tests act to specify the required software behaviour as it is developed and also to prevent accidental regressions during development.

The developer tests for these components were automated using the JUnit [1] test libraries. Additional software was also written to provide "test stubs" for any parts of the system that had not yet been developed. An example of such a test stub was a "fake" template driver. This could be used in place of any of the other template drivers during the testing of other parts of the software. This template driver generated its own output designed for easy recognition and checking. The use of a test stub for this purpose rather than a real driver meant that any later changes to real drivers would not cause unrelated tests to fail.

The use of *GILT* software in further research was only considered once the code passed all the developer tests described in the following subsections.

E.9.1 Testing the Intermediate Language Compiler

The compiler for the intermediate language is a relatively complex piece of software which needed to evolve with the design of the intermediate language itself. Tests for the compiler form three rough groups: syntax tests, placeholder tests, and grammar tests. Syntax tests include testing for correct recognition of empty files, boilerplate text without placeholders, and placeholders without boilerplate text. This group of tests also includes testing for correct error output in the case of incorrect intermediate language syntax such as incomplete placeholders or quoted text.

Placeholder tests for the intermediate language compiler were more numerous, as there were more combinations and possibilities to be considered, even though

[1] `https://junit.org/junit5/`

they only concentrated on the parsing of the contents of a single placeholder. All the placeholder tests were constructed in a similar style. An object of the class `PlaceholderParser` was created and provided with a text string containing the textual contents of a placeholder to be parsed, then the `parse` method of the object was called to return a parsed `Node` object. If no errors or exceptions were produced, the contents of the Node were then examined to check that the placeholder had been correctly parsed. Each type of placeholder had a set of tests covering valid and invalid content. Tests for valid content included checking the presence and absence of optional directives and values as well as checking with different types of values such as "lookup" values and quoted literal values.

Grammar tests include tests for the correct arrangement of placeholders and boilerplate text, for example, checking that each block start has a matching end. These were mostly tested as part of the template generation process, using the fake template driver and checking both that no errors were produced and that the correct output template was produced.

E.9.2 Testing the Template Generation Process

The tests for the template generation process built on the intermediate language compiler tests. Having determined that the parsing process was generating the correct nodes from both boilerplate text and placeholders, the next step was to ensure that the sequence of nodes representing the parsed template was being analysed correctly and passed to the appropriate driver methods for rendering. All the template generation tests were constructed in the same style, and a helper method was created to contain common code to minimise the size of each test. Before running any of the tests a template driver factory object for the fake template driver was created. At the start of each test a memory-based template storage "place" was created and a template driver was created using the template driver factory which was already available. When the driver had been created, a `TemplateCompiler` object was created and its `compile` method called, passing in the details of the template storage and the text containing a complete template to be processed. To complete the test, the generated template or templates were then retrieved from the template storage and compared with the expected result.

E.9.3 Testing Real Template Drivers

Having determined that the parsing and template generation process was functioning correctly, the next step was to write and test the drivers for the rest of the template languages. This testing took place in several phases. The first

phase was to code the driver and ensure that there were no compilation errors or run time problems when it was called by the template generator. The next phase was to perform a basic test using a standard, if relatively simple, input intermediate template which includes a range of text and placeholder types. Each driver test began by creating and initialising the template generation system, the driver, a template context, and the template engine being tested. With everything set up, the basic intermediate template was passed to the template generator to produce a destination template which was then expanded to produce a final document. This document was then checked to ensure that the resulting document contents were correct.

Beyond that, testing of each template driver needed to be specific to the template language being generated. Not only did the form and content of the generated template vary between template drivers, but each template language supported a different subset of the possible features available in the intermediate language. For these advanced features, the test suite for each driver contained a different set of tests, ensuring both that supported language features were correctly processed and that unsupported features produced a reasonable error message.

E.10 Discussion and Conclusions

This research explored the possibility of creating an intermediate language to represent common template features in a way which could be used to generate templates for a range of template engines. On the whole, this exercise was a success, with the *GILT* language capable of expressing all the required features, and the tooling capable of generating equivalent templates for all the supported template engines, subject to their capabilities.

The initial version of the intermediate language still had some problems, though, including an arguably poor choice of delimiter syntax, a simplistic notion of value types, and potentially ambiguous interpretation of complex scenarios involving method parameters. Improvements to address these and other issues will be considered for a future version of the language and tooling.

The language and tooling were, however, able to be used for the performance and energy comparisons of more complex template scenarios as discussed in Chapter 6.

Appendix F

GILT Input and Output Examples

F.1 GILT Input Documents used for Section 6.4

```
<!DOCTYPE html>
<html><head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
  <meta charset="utf-8">
<!-- Google Analytics -->
<script async="" src="assets/analytics.js"></script><script type="
    text/javascript">
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||
    function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.
    createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.
    insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.
    js','ga');
ga('create', 'UA-31529903-2', 'auto');
ga('send', 'pageview');
</script>
<!-- End Google Analytics -->
<!-- Global site tag (gtag.js) - Google Analytics
<script async src="https://www.googletagmanager.com/gtag/js?id=UA
    -31529903-2"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());
  gtag('config', 'UA-31529903-2');
</script>
 -->
  <title>¬{lookup "site_title"}</title>
  <meta name="viewport" content="width=device-width, initial-scale
      =1, maximum-scale=1">
  <meta property="og:type" content="website">
<meta property="og:title" content="¬{lookup "page_title"}">
<meta property="og:url" content="¬{lookup "base_url"}/index.html">
<meta property="og:site_name" content="¬{lookup "site_title"}">
```

```
<meta property="og:locale" content="en_US">
<meta property="article:author" content="¬{lookup "author"}">
¬{foreach lookup "tags" do block "site_tag"}¬{start "site_tag"}<meta
    property="article:tag" content="¬{reference "this"}">
¬{end "site_tag"}<meta name="twitter:card" content="summary">
    <link rel="alternate" href="¬{lookup "base_url"}/atom.xml"
       title="¬{lookup "site_title"}" type="application/atom+xml">
    <link href="assets/css.css" rel="stylesheet" type="text/css">
  <link rel="stylesheet" href="assets/bootstrap.min.css" integrity
     ="sha384-1q8mTJOASx8j1Au+
     a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7" crossorigin
     ="anonymous">
  <link rel="stylesheet" href="assets/font-awesome.min.css"
     integrity="sha384-XdYbMnZ/QjLh6iI4ogqCTaIjrFk87ip+ekIjefZch0Y+
     PvJ8CDYtEs1ipDmPorQ+" crossorigin="anonymous">
<link rel="stylesheet" href="assets/styles.css">
 <meta name="generator" content="Hexo 5.4.1"><script type="text/
    javascript" async="" src="assets/count.js"></script><script src
    ="assets/count-data.js"></script></head>
<body>
  <nav class="navbar navbar-inverse">
  <div class="container">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-
         toggle="collapse" data-target="#main-menu-navbar" aria-
         expanded="false">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
    </div>
    <!-- Collect the nav links, forms, and other content for
       toggling -->
    <div class="collapse navbar-collapse" id="main-menu-navbar">
      <ul class="nav navbar-nav">
          <li><a class="active" href="¬{lookup "base_url"}/index.
             html">Home</a></li>
          <li><a class="" href="¬{lookup "base_url"}/about/">About
             Me</a></li>
          <li><a class="" href="¬{lookup "base_url"}/site/">This
             Site</a></li>
          <li><a class="" href="¬{lookup "base_url"}/archives/">
             Archives</a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right">
        <li><div class="collapse navbar-collapse" id="
           navbarSupportedContent">
          <form action="//google.com/search" method="GET" accept-
             charset="UTF-8" id="search-form" class="form-inline ml
             -auto">
            <div class="md-form my-0">
              <input class="form-control" input="" type="search"
                 name="q" placeholder="Search" aria-label="Search">
              <input type="hidden" name="sitesearch" value="¬{
                 lookup "base_url"}">
            </div>
          </form>
        </div>
        <!-- <li></li><a id="nav-search-btn" class="fa fa-search"
```

```
            title="Search"></a>
        <div id="search-form-wrap">
            <form action="//google.com/search" method="get" accept-
                charset="UTF-8" class="search-form"><input type="
                search" name="q" class="search-form-input"
                placeholder="Search"><button type="submit" class="
                search-form-submit">&#xF002;</button><input type="
                hidden" name="sitesearch" value="¬{lookup "base_url
                "}"></form>
        </div> -->
          </li><li><a href="¬{lookup "base_url"}/atom.xml" title="
              rss_feed&quot;"><i class="fa fa-rss"></i></a></li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
  <div class="container">
    <div class="blog-header">
  <h1 class="blog-title">¬{lookup "site_title"}</h1>
</div>
    <div class="row">
        <div class="col-sm-8 blog-main">
¬{template "t2021_01_05_hosting"}
¬{template "t2020_11_23_what_is_sustainability"}
¬{template "t2020_10_23_tomm_flowers_presentation"}
¬{template "t2020_10_13_tommy_flowers_autumn_2020"}
¬{template "t2020_09_13_make_a_difference"}
¬{template "t2020_09_04_a_new_venture"}
        </div>
        <div class="col-sm-3 col-sm-offset-1 blog-sidebar">
¬{if lookup "newsletter_signup" then block "signup"}¬{start "signup
    "}
    <div class="widget-wrap">
      <h3 class="widget-title">Newsletter</h3>
      <div class="widget newsletter">
        For extra news and updates, please sign up to my newsletter
            .
        <form method="post" action="https://sendfox.com/form/1kgjo9
            /1wqgwj" class="sendfox-form" id="1wqgwj" data-async="
            true" data-recaptcha="false">
          <p><input type="text" placeholder="First Name" name="
              first_name" required=""></p>
          <p><input type="text" placeholder="Last Name" name="
              last_name" required=""></p>
          <p><input type="email" placeholder="Email" name="email"
              required=""></p>
          <p><label><input type="checkbox" name="gdpr" value="1"
              required=""> I agree to receive email updates and
              information.</label></p>
          <!-- no botz please -->
          <div style="position: absolute; left: -5000px;" aria-
              hidden="true"><input type="text" name="a_password"
              tabindex="-1" autocomplete="off"></div>
          <p><button type="submit">Subscribe</button></p>
        </form>
        <script src="assets/form.js"></script>
        <span class="smallprint">you can unsusbscribe at any time</
            span>
      </div>
    </div>
¬{end "signup"}
```

```
¬{template "tags_module"}
¬{template "tagcloud_module"}
¬{template "archives_module"}
¬{template "recent_module"}
        </div>
    </div>
  </div>
  <footer class="blog-footer">
  <div class="container">
    <div id="footer-info" class="inner">
      © ¬{lookup "copyright_year"} ¬{lookup "author"}<br>
      Proudly hosted by <a href="https://www.greengeeks.com/track/
          efficacy" target="_blank">Green Geeks Hosting</a><br>
      Proudly powered by  <a href="http://hexo.io/" target="_blank
          ">Hexo Static Site Generator</a>
    </div>
  </div>
</footer>
¬{template "disqus"}
<script src="assets/jquery.min.js" integrity="sha384-8
    gBf6Y4YYq7Jx97PIqmTwLPin4hxIzQw5aDmUg/DDhul9fFpbbLcLh3nTIIDJKhx"
     crossorigin="anonymous"></script>
<script src="assets/bootstrap.min.js" integrity="sha384-0
    mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
     crossorigin="anonymous"></script>
<script src="assets/script.js"></script>
</body></html>
```

Listing F.1: index.tpl

```
  <div class="sidebar-module">
    <h4>Archives</h4>
    <ul class="sidebar-module-list"><li class="sidebar-module-list-
        item"><a class="sidebar-module-list-link" href="¬{lookup "
        base_url"}/archives/2021/01/">January 2021</a><span class="
        sidebar-module-list-count">1</span></li><li class="sidebar-
        module-list-item"><a class="sidebar-module-list-link" href="
        ¬{lookup "base_url"}/archives/2020/11/">November 2020</a><
        span class="sidebar-module-list-count">1</span></li><li
        class="sidebar-module-list-item"><a class="sidebar-module-
        list-link" href="¬{lookup "base_url"}/archives/2020/10/">
        October 2020</a><span class="sidebar-module-list-count">2</
        span></li><li class="sidebar-module-list-item"><a class="
        sidebar-module-list-link" href="¬{lookup "base_url"}/
        archives/2020/09/">September 2020</a><span class="sidebar-
        module-list-count">2</span></li></ul>
  </div>
```

Listing F.2: archives_module.tpl

```
<script>
  var disqus_shortname = 'the-green-programmer';
  (function(){
    var dsq = document.createElement('script');
    dsq.type = 'text/javascript';
    dsq.async = true;
    dsq.src = '//' + disqus_shortname + '.disqus.com/count.js';
    (document.getElementsByTagName('head')[0] || document.
        getElementsByTagName('body')[0]).appendChild(dsq);
  })();
```

```
</script>
```

Listing F.3: disqus.tpl

```
    <div class="widget-wrap">
      <h3 class="widget-title">Newsletter</h3>
      <div class="widget newsletter">
        For extra news and updates, please sign up to my newsletter
            .
        <form method="post" action="https://sendfox.com/form/1kgjo9
            /1wqgwj" class="sendfox-form" id="1wqgwj" data-async="
            true" data-recaptcha="false">
          <p><input type="text" placeholder="First Name" name="
              first_name" required=""></p>
          <p><input type="text" placeholder="Last Name" name="
              last_name" required=""></p>
          <p><input type="email" placeholder="Email" name="email"
              required=""></p>
          <p><label><input type="checkbox" name="gdpr" value="1"
              required=""> I agree to receive email updates and
              information.</label></p>
          <!-- no botz please -->
          <div style="position: absolute; left: -5000px;" aria-
              hidden="true"><input type="text" name="a_password"
              tabindex="-1" autocomplete="off"></div>
          <p><button type="submit">Subscribe</button></p>
        </form>
        <script src="assets/form.js"></script>
        <span class="smallprint">you can unsusbscribe at any time</
            span>
      </div>
    </div>
```

Listing F.4: newsletter_widget.tpl

```
  <div class="sidebar-module">
    <h4>Recent Posts</h4>
    <ul class="sidebar-module-list">
        <li>
          <a href="¬{lookup "base_url"}/2021/01/05/hosting/">What
              can we do for a sustainable web?</a>
        </li>
        <li>
          <a href="¬{lookup "base_url"}/2020/11/23/what-is-
              sustainability/">What does sustainability mean in
              software development?</a>
        </li>
        <li>
          <a href="¬{lookup "base_url"}/2020/10/23/tomm-flowers-
              presentation/">Tommy Flowers 2020 presentation</a>
        </li>
        <li>
          <a href="¬{lookup "base_url"}/2020/10/13/tommy-flowers-
              autumn-2020/">The Tommy Flowers Network Autumn 2020
              Conference</a>
        </li>
        <li>
          <a href="¬{lookup "base_url"}/2020/09/13/make-a-
              difference/">Why do I think we can make a difference
              ?</a>
        </li>
```

```
    </ul>
  </div>
```

Listing F.5: recent_module.tpl

```
    <article id="post-a-new-venture" class="article article-type-
        post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="¬{lookup "base_url
          "}/2020/09/04/a-new-venture/">A New Venture</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="¬{lookup "base_url"}/2020/09/04/a-new-venture/" class="
      article-date"><time datetime="2020-09-04T12:00:25.000Z"
      itemprop="datePublished">2020-09-04</time></a>
</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p>A first blog post is always a little bit uncomfortable,
            but we all have to start somewhere.</p>
<p>This is a blog in which I intend to explore the strange and
    often counterintuitive world of "<em>green</em>" (a.k.a "<em>
    sustainable</em>") programming and software development in
    general.</p>
<p>I feel as if there is a lot to say, so I hope I manage to get it
     written down. And I hope you will be along for the journey!</p>
    </div>
    <footer class="article-footer">
      <a data-url="¬{lookup "base_url"}/2020/09/04/a-new-venture/"
          data-id="cl14uzq6s00062amb92xpa24b" class="article-share-
          link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="¬{lookup "base_url"}/2020/09/04/a-new-venture/#
            disqus_thread" class="article-comment-link">0 Comments</
            a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ¬{lookup "base_url"}/tags/Admin/" rel="tag">Admin</a></li></ul
      >
    </footer>
  </div>
</article>
```

Listing F.6: t2020_09_04_a_new_venture.tpl

```
    <article id="post-make-a-difference" class="article article-
        type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="¬{lookup "base_url
          "}/2020/09/13/make-a-difference/">Why do I think we can
          make a difference?</a>
    </h1>
  </header>

  <div class="article-meta">
```

```
    <div class="article-datetime">
  <a href="¬{lookup "base_url"}/2020/09/13/make-a-difference/"
     class="article-date"><time datetime="2020-09-13T08:58:24.000Z"
      itemprop="datePublished">2020-09-13</time></a>
</div>
    <div class="article-author">Frank Carver</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="¬{lookup "base_url"}/images/jefferson-santos-9
            SoCnyQmkzI-unsplash.jpg" title="Photo credit: Jefferson
            Santos - Unsplash" class="image-link" rel="article4"><
            img src="assets/jefferson-santos-9SoCnyQmkzI-unsplash.
            jpg" alt="Photo credit: Jefferson Santos - Unsplash"
            title="Programmer"></a><span class="caption">Photo
            credit: Jefferson Santos - Unsplash</span></p>
<p>In 2019, <a target="_blank" rel="noopener" href="https://www.kth
    .se/en">KTH Royal Institute of Technology</a> in Sweden
    estimated that <strong>the internet uses over 10% of the world's
     electricity</strong>. This figure is still growing, too.
    Enforced isolation driven by the COVID19 pandemic has seen
    internet usage shoot up.</p>
<p>Naturally, there has been a lot of research on this topic, and I
     plan
 to explore some of it on this blog in future posts. However, most
     of it
 has concentrated on the <em>hardware</em> - the cables, routers,
computers, and data centres which comprise the physical parts of
    the
system. These are the bits that consume all this energy, after all,
     and
if their consumption can be improved it seems reasonable to assume
    that
the overall energy consumption of the internet as a system will
decrease.</p>
<p>Unfortunately, behind this level of obviousness are some things
    which get a lot less airtime:</p>
          <p class="article-more-link">
            <a class="btn btn-primary" href="¬{lookup "base_url
                "}/2020/09/13/make-a-difference/#more">Read More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/09/13/make-a-
          difference/" data-id="cl14uzq6z00092amb5t738df5" class="
          article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/09/13/make-a-
            difference/#disqus_thread" class="article-comment-link
            ">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ¬{lookup "base_url"}/tags/Why/" rel="tag">Why</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.7: t2020_09_13_make_a_difference.tpl

```
    <article id="post-tommy-flowers-autumn-2020" class="article
```

```
       article-type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="¬{lookup "base_url
          "}/2020/10/13/tommy-flowers-autumn-2020/">The Tommy
          Flowers Network Autumn 2020 Conference</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="¬{lookup "base_url"}/2020/10/13/tommy-flowers-autumn
      -2020/" class="article-date"><time datetime="2020-10-13T16
      :06:14.000Z" itemprop="datePublished">2020-10-13</time></a>
</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="¬{lookup "base_url"}/images/tfnConf.jpg" title
            ="Photo credit: Tommy Flowers Network" class="image-link
            " rel="article3"><img src="assets/tfnConf.jpg" alt="
            Photo credit: Tommy Flowers Network" title="Conference
            "></a><span class="caption">Photo credit: Tommy Flowers
            Network</span></p>
<p>The <a target="_blank" rel="noopener" href="http://
    tommyflowersnetwork.blogspot.com/" title="The Tommy Flowers
    Network">Tommy Flowers Network</a>
 is a partnership between industry and academia to bring together a
unique combination of theoretical, practical, and commercial
    approaches.
 Named after the GPO engineer <a target="_blank" rel="noopener"
     href="https://en.wikipedia.org/wiki/Tommy_Flowers">Tommy
     Flowers BSc DSc MBE</a>, who worked on the code breaking
     machines at <a target="_blank" rel="noopener" href="https://en.
     wikipedia.org/wiki/Bletchley_Park">Bletchley Park</a>
 during the Second World War, the Network aims to collaborate on
     solving
 some of the biggest social and technological problems of our age
     .</p>
<p>The Tommy Flowers Network is currently sponsored by <a target="
    _blank" rel="noopener" href="https://www.bt.com/">British
    Telecom</a> (BT) and hosts regular conferences at BT's <a target
    ="_blank" rel="noopener" href="https://atadastral.co.uk/">
    Adastral Park</a>
 research site. In 2020 the Autumn conference runs from 12th-16th
October and is being held fully online. I have been given the
opportunity to present my work at this conference on Wednesday 14
October in the form of a five-minute <em>"Lightning Talk"</em>.</p>
<p>I recommend you check out <a target="_blank" rel="noopener" href
    ="http://tommyflowersnetwork.blogspot.com/" title="The Tommy
    Flowers Network">Tommy Flowers Network</a> and the <a target="
    _blank" rel="noopener" href="http://tommyflowersnetwork.blogspot
    .com/2020/07/virtual-conference-lets-get-physical.html">Autumn
    2020 conference</a>.
 Even if you don't have a chance to attend live, I understand that
     the
presentations will be shared on their website after the event.</p>
<p>If you want a flavour of my talk ahead of time, feel free to
    download <a href="¬{lookup "base_url"}/downloads/2020-10-13-tfn.
    pptx">the PowerPoint slide deck</a> for the presentation.</p>
    </div>
    <footer class="article-footer">
```

```
      <a data-url="https://greenprogrammer.net/2020/10/13/tommy-
          flowers-autumn-2020/" data-id="cl14uzq73000b2ambfd18fffu"
          class="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/10/13/tommy-
            flowers-autumn-2020/#disqus_thread" class="article-
            comment-link">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ¬{lookup "base_url"}/tags/Conferences/" rel="tag">Conferences
      </a></li><li class="article-tag-list-item"><a class="article-
      tag-list-link" href="¬{lookup "base_url"}/tags/Presentations/"
       rel="tag">Presentations</a></li></ul>
    </footer>
  </div>

</article>
```

Listing F.8: t2020_10_13_tommy_flowers_autumn_2020.tpl

```
    <article id="post-tomm-flowers-presentation" class="article
        article-type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="¬{lookup "base_url
          "}/2020/10/23/tomm-flowers-presentation/">Tommy Flowers
          2020 presentation</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="¬{lookup "base_url"}/2020/10/23/tomm-flowers-
      presentation/" class="article-date"><time datetime="2020-10-23
      T15:27:15.000Z" itemprop="datePublished">2020-10-23</time></a>
</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p>At the recent <a target="_blank" rel="noopener" href="
            http://tommyflowersnetwork.blogspot.com/" title="The
            Tommy Flowers Network">Tommy Flowers Network</a>
 conference, I presented a five-minute session on my research as
     part of
 their "Lightning Talks". Despite several practice runs and
     trimming the
 presentation to fit the time slot, I still managed to hit the time
limit before I got to my final two slides.</p>
<p>After staying with the conference to watch the remaining
presentations for that day, I decided to re-record a version of the
     same
 presentation that I am calling the <em>Director's Cut</em>. In
     this one I take a bit more time to explain each step, with the
     result that it take a bit over ten minutes.</p>
<p>This is still pretty short, so even if you did see the rushed
    live
presentation, I encourage you to take a look at this one, to get
    the
full experience.</p>
<style>.embed-container {
    position: relative;
```

```
    padding-bottom: 56.25%;
    height: 0;
    overflow: hidden;
    max-width: 100%;
  }
  .embed-container iframe, .embed-container object, .embed-
     container embed {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
  }
  </style>
<div class="embed-container"><iframe src="assets/qn6IwZJvH50.html"
    allowfullscreen="" allow="accelerometer; autoplay; encrypted-
    media; gyroscope; picture-in-picture" frameborder="20"></iframe
    ></div>
<br>
          <p class="article-more-link">
            <a class="btn btn-primary" href="¬{lookup "base_url
                "}/2020/10/23/tomm-flowers-presentation/#more">Read
                More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/10/23/tomm-
          flowers-presentation/" data-id="cl14uzq71000a2amb948v63nz"
           class="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/10/23/tomm-
            flowers-presentation/#disqus_thread" class="article-
            comment-link">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ¬{lookup "base_url"}/tags/Conferences/" rel="tag">Conferences
      </a></li><li class="article-tag-list-item"><a class="article-
      tag-list-link" href="¬{lookup "base_url"}/tags/Presentations/"
       rel="tag">Presentations</a></li><li class="article-tag-list-
      item"><a class="article-tag-list-link" href="¬{lookup "
      base_url"}/tags/Video/" rel="tag">Video</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.9: t2020_10_23_tomm_flowers_presentation.tpl

```
    <article id="post-what-is-sustainability" class="article
       article-type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="¬{lookup "base_url
          "}/2020/11/23/what-is-sustainability/">What does
          sustainability mean in software development?</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="¬{lookup "base_url"}/2020/11/23/what-is-sustainability/"
      class="article-date"><time datetime="2020-11-23T14:58:24.000Z
```

```
      " itemprop="datePublished">2020-11-23</time></a>
</div>
    <div class="article-author">Frank Carver</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="¬{lookup "base_url"}/images/
            image1170x530cropped.jpg" title="Photo credit: The
            United Nations" class="image-link" rel="article1"><img
            src="assets/image1170x530cropped.jpg" alt="Photo credit:
             The United Nations" title="UN Sustainability Goals"></a
            ><span class="caption">Photo credit: The United Nations
            </span></p>
<p><em>Sustainability</em> is a complex term that is used in a lot
    of
different contexts, and has many different assumptions and
    associations
toed to it. In my research, <em>sustainability</em> is a vital
    concept
which underlies the research, and is found in many of the key
    papers in
the research literature. So there is no avoiding the question: what
     <em>does</em> sustainability mean in software development?</p>
          <p class="article-more-link">
            <a class="btn btn-primary" href="¬{lookup "base_url
                "}/2020/11/23/what-is-sustainability/#more">Read
                More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/11/23/what-is-
          sustainability/" data-id="cl14uzq7h000t2ambhjbx7uzj" class
          ="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/11/23/what-is-
            sustainability/#disqus_thread" class="article-comment-
            link">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ¬{lookup "base_url"}/tags/Why/" rel="tag">Why</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.10: t2020_11_23_what_is_sustainability.tpl

```
    <article id="post-hosting" class="article article-type-post"
        itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="¬{lookup "base_url
          "}/2021/01/05/hosting/">What can we do for a sustainable
          web?</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="¬{lookup "base_url"}/2021/01/05/hosting/" class="article
      -date"><time datetime="2021-01-05T17:22:06.000Z" itemprop="
      datePublished">2021-01-05</time></a>
```

```
</div>
    <div class="article-author">Frank Carver</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="¬{lookup "base_url"}/images/
            IMG_20201128_143813208_trees-cropped.jpg" title="Photo
            credit: Frank Carver (c) 2020" class="image-link" rel="
            article0"><img src="assets/IMG_20201128_143813208_trees-
            cropped.jpg" alt="Photo credit: Frank Carver (c) 2020"
            title="Trees"></a><span class="caption">Photo credit:
            Frank Carver (c) 2020</span></p>
<p>As I have mentioned before on this blog, I am currently
    undertaking
research for a PhD in sustainable software. I don't have full
    results
from my experimentation so far, but the more I look into this area,
     and
the more that I see of the huge impact the internet has on the
environment, the more I want to start making changes <strong>right
    now</strong>.</p>
<p>This post covers some of the things I am doing, and some of the
    things <em>you</em> can do too, to help make our contribution to
     the internet "greener".</p>
          <p class="article-more-link">
            <a class="btn btn-primary" href="¬{lookup "base_url
                "}/2021/01/05/hosting/#more">Read More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2021/01/05/hosting/"
           data-id="cl14uzq6u00072amba8riggsv" class="article-share-
          link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2021/01/05/hosting/#
            disqus_thread" class="article-comment-link">0 Comments</
            a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ¬{lookup "base_url"}/tags/Hosting/" rel="tag">Hosting</a></li
      ><li class="article-tag-list-item"><a class="article-tag-list-
      link" href="¬{lookup "base_url"}/tags/How/" rel="tag">How</a
      ></li></ul>
    </footer>
  </div>
</article>
```

Listing F.11: t2021_01_05_hosting.tpl

```
  <div class="sidebar-module">
    <h4>Tag Cloud</h4>
    <p class="tagcloud">
      <a href="¬{lookup "base_url"}/tags/Admin/" style="font-size:
          10px;">Admin</a> <a href="¬{lookup "base_url"}/tags/
          Conferences/" style="font-size: 20px;">Conferences</a> <a
          href="¬{lookup "base_url"}/tags/Hosting/" style="font-size
          : 10px;">Hosting</a> <a href="¬{lookup "base_url"}/tags/
          How/" style="font-size: 10px;">How</a> <a href="¬{lookup "
          base_url"}/tags/Presentations/" style="font-size: 20px;">
          Presentations</a> <a href="¬{lookup "base_url"}/tags/Video
```

```
          /" style="font-size: 10px;">Video</a> <a href="¬{lookup "
          base_url"}/tags/Why/" style="font-size: 20px;">Why</a>
    </p>
  </div>
```

Listing F.12: tagcloud_module.tpl

```
  <div class="sidebar-module">
    <h4>Tags</h4>
    <ul class="sidebar-module-list" itemprop="keywords"><li class="
        sidebar-module-list-item"><a class="sidebar-module-list-link
        " href="¬{lookup "base_url"}/tags/Admin/" rel="tag">Admin</a
        ><span class="sidebar-module-list-count">1</span></li><li
        class="sidebar-module-list-item"><a class="sidebar-module-
        list-link" href="¬{lookup "base_url"}/tags/Conferences/" rel
        ="tag">Conferences</a><span class="sidebar-module-list-count
        ">2</span></li><li class="sidebar-module-list-item"><a class
        ="sidebar-module-list-link" href="¬{lookup "base_url"}/tags/
        Hosting/" rel="tag">Hosting</a><span class="sidebar-module-
        list-count">1</span></li><li class="sidebar-module-list-item
        "><a class="sidebar-module-list-link" href="¬{lookup "
        base_url"}/tags/How/" rel="tag">How</a><span class="sidebar-
        module-list-count">1</span></li><li class="sidebar-module-
        list-item"><a class="sidebar-module-list-link" href="¬{
        lookup "base_url"}/tags/Presentations/" rel="tag">
        Presentations</a><span class="sidebar-module-list-count">2</
        span></li><li class="sidebar-module-list-item"><a class="
        sidebar-module-list-link" href="¬{lookup "base_url"}/tags/
        Video/" rel="tag">Video</a><span class="sidebar-module-list-
        count">1</span></li><li class="sidebar-module-list-item"><a
        class="sidebar-module-list-link" href="¬{lookup "base_url"}/
        tags/Why/" rel="tag">Why</a><span class="sidebar-module-list
        -count">2</span></li></ul>
  </div>
```

Listing F.13: tags_module.tpl

F.2 Freemarker Output Documents produced for Section 6.4

```
<!DOCTYPE html>
<html><head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
  <meta charset="utf-8">
<!-- Google Analytics -->
<script async="" src="assets/analytics.js"></script><script type="
    text/javascript">
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||
    function(){
(i[r].q=i[r].q||[]).push(arguments)},i[r].1=1*new Date();a=s.
    createElement(o),
m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.
    insertBefore(a,m)
})(window,document,'script','//www.google-analytics.com/analytics.
    js','ga');
ga('create', 'UA-31529903-2', 'auto');
ga('send', 'pageview');
</script>
```

```
<!-- End Google Analytics -->
<!-- Global site tag (gtag.js) - Google Analytics
<script async src="https://www.googletagmanager.com/gtag/js?id=UA
    -31529903-2"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());
  gtag('config', 'UA-31529903-2');
</script>
 -->
  <title>${(site_title)!}</title>
  <meta name="viewport" content="width=device-width, initial-scale
      =1, maximum-scale=1">
  <meta property="og:type" content="website">
<meta property="og:title" content="${(page_title)!}">
<meta property="og:url" content="${(base_url)!}/index.html">
<meta property="og:site_name" content="${(site_title)!}">
<meta property="og:locale" content="en_US">
<meta property="article:author" content="${(author)!}">
<#list tags as this><meta property="article:tag" content="${(this)
    !}">
</#list><meta name="twitter:card" content="summary">
    <link rel="alternate" href="${(base_url)!}/atom.xml" title="${(
        site_title)!}" type="application/atom+xml">
    <link href="assets/css.css" rel="stylesheet" type="text/css">
  <link rel="stylesheet" href="assets/bootstrap.min.css" integrity
      ="sha384-1q8mTJOASx8j1Au+
      a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7" crossorigin
      ="anonymous">
  <link rel="stylesheet" href="assets/font-awesome.min.css"
      integrity="sha384-XdYbMnZ/QjLh6iI4ogqCTaIjrFk87ip+ekIjefZch0Y+
      PvJ8CDYtEs1ipDmPorQ+" crossorigin="anonymous">
<link rel="stylesheet" href="assets/styles.css">
 <meta name="generator" content="Hexo 5.4.1"><script type="text/
     javascript" async="" src="assets/count.js"></script><script src
     ="assets/count-data.js"></script></head>
<body>
  <nav class="navbar navbar-inverse">
  <div class="container">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-
          toggle="collapse" data-target="#main-menu-navbar" aria-
          expanded="false">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
    </div>
    <!-- Collect the nav links, forms, and other content for
        toggling -->
    <div class="collapse navbar-collapse" id="main-menu-navbar">
      <ul class="nav navbar-nav">
          <li><a class="active" href="${(base_url)!}/index.html">
              Home</a></li>
          <li><a class="" href="${(base_url)!}/about/">About Me</a
              ></li>
          <li><a class="" href="${(base_url)!}/site/">This Site</a
              ></li>
```

```
          <li><a class="" href="${(base_url)!}/archives/">Archives
              </a></li>
      </ul>
      <ul class="nav navbar-nav navbar-right">
        <li><div class="collapse navbar-collapse" id="
            navbarSupportedContent">
          <form action="//google.com/search" method="GET" accept-
              charset="UTF-8" id="search-form" class="form-inline ml
              -auto">
            <div class="md-form my-0">
              <input class="form-control" input="" type="search"
                  name="q" placeholder="Search" aria-label="Search">
              <input type="hidden" name="sitesearch" value="${(
                  base_url)!}">
            </div>
          </form>
        </div>
        <!-- <li></li><a id="nav-search-btn" class="fa fa-search"
            title="Search"></a>
        <div id="search-form-wrap">
            <form action="//google.com/search" method="get" accept-
                charset="UTF-8" class="search-form"><input type="
                search" name="q" class="search-form-input"
                placeholder="Search"><button type="submit" class="
                search-form-submit">&#xF002;</button><input type="
                hidden" name="sitesearch" value="${(base_url)!}"></
                form>
        </div> -->
          </li><li><a href="${(base_url)!}/atom.xml" title="
              rss_feed&quot;"><i class="fa fa-rss"></i></a></li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
  <div class="container">
    <div class="blog-header">
  <h1 class="blog-title">${(site_title)!}</h1>
</div>
    <div class="row">
        <div class="col-sm-8 blog-main">
<#include "t2021_01_05_hosting">
<#include "t2020_11_23_what_is_sustainability">
<#include "t2020_10_23_tomm_flowers_presentation">
<#include "t2020_10_13_tommy_flowers_autumn_2020">
<#include "t2020_09_13_make_a_difference">
<#include "t2020_09_04_a_new_venture">
        </div>
        <div class="col-sm-3 col-sm-offset-1 blog-sidebar">
<#if newsletter_signup??>    <div class="widget-wrap">
      <h3 class="widget-title">Newsletter</h3>
      <div class="widget newsletter">
        For extra news and updates, please sign up to my newsletter
            .
        <form method="post" action="https://sendfox.com/form/1kgjo9
            /1wqgwj" class="sendfox-form" id="1wqgwj" data-async="
            true" data-recaptcha="false">
          <p><input type="text" placeholder="First Name" name="
              first_name" required=""></p>
          <p><input type="text" placeholder="Last Name" name="
              last_name" required=""></p>
          <p><input type="email" placeholder="Email" name="email"
```

```
            required=""></p>
          <p><label><input type="checkbox" name="gdpr" value="1"
              required=""> I agree to receive email updates and
              information.</label></p>
          <!-- no botz please -->
          <div style="position: absolute; left: -5000px;" aria-
              hidden="true"><input type="text" name="a_password"
              tabindex="-1" autocomplete="off"></div>
          <p><button type="submit">Subscribe</button></p>
        </form>
        <script src="assets/form.js"></script>
        <span class="smallprint">you can unsusbscribe at any time</
            span>
      </div>
    </div>
</#if>
<#include "tags_module">
<#include "tagcloud_module">
<#include "archives_module">
<#include "recent_module">
        </div>
    </div>
  </div>
  <footer class="blog-footer">
  <div class="container">
    <div id="footer-info" class="inner">
      © ${(copyright_year)!} ${(author)!}<br>
      Proudly hosted by <a href="https://www.greengeeks.com/track/
          efficacy" target="_blank">Green Geeks Hosting</a><br>
      Proudly powered by  <a href="http://hexo.io/" target="_blank
          ">Hexo Static Site Generator</a>
    </div>
  </div>
</footer>
<#include "disqus">
<script src="assets/jquery.min.js" integrity="sha384-8
    gBf6Y4YYq7Jx97PIqmTwLPin4hxIzQw5aDmUg/DDhul9fFpbbLcLh3nTIIDJKhx"
     crossorigin="anonymous"></script>
<script src="assets/bootstrap.min.js" integrity="sha384-0
    mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
     crossorigin="anonymous"></script>
<script src="assets/script.js"></script>
</body></html>
```

Listing F.14: index.tpl

```
  <div class="sidebar-module">
    <h4>Archives</h4>
    <ul class="sidebar-module-list"><li class="sidebar-module-list-
        item"><a class="sidebar-module-list-link" href="${(base_url)
        !}/archives/2021/01/">January 2021</a><span class="sidebar-
        module-list-count">1</span></li><li class="sidebar-module-
        list-item"><a class="sidebar-module-list-link" href="${(
        base_url)!}/archives/2020/11/">November 2020</a><span class
        ="sidebar-module-list-count">1</span></li><li class="sidebar
        -module-list-item"><a class="sidebar-module-list-link" href
        ="${(base_url)!}/archives/2020/10/">October 2020</a><span
        class="sidebar-module-list-count">2</span></li><li class="
        sidebar-module-list-item"><a class="sidebar-module-list-link
        " href="${(base_url)!}/archives/2020/09/">September 2020</a
        ><span class="sidebar-module-list-count">2</span></li></ul>
```

```
  </div>
```

Listing F.15: archives_module.tpl

```
<script>
  var disqus_shortname = 'the-green-programmer';
  (function(){
    var dsq = document.createElement('script');
    dsq.type = 'text/javascript';
    dsq.async = true;
    dsq.src = '//' + disqus_shortname + '.disqus.com/count.js';
    (document.getElementsByTagName('head')[0] || document.
        getElementsByTagName('body')[0]).appendChild(dsq);
  })();
</script>
```

Listing F.16: disqus.tpl

```
    <div class="widget-wrap">
      <h3 class="widget-title">Newsletter</h3>
      <div class="widget newsletter">
        For extra news and updates, please sign up to my newsletter
            .
        <form method="post" action="https://sendfox.com/form/1kgjo9
            /1wqgwj" class="sendfox-form" id="1wqgwj" data-async="
            true" data-recaptcha="false">
          <p><input type="text" placeholder="First Name" name="
              first_name" required=""></p>
          <p><input type="text" placeholder="Last Name" name="
              last_name" required=""></p>
          <p><input type="email" placeholder="Email" name="email"
              required=""></p>
          <p><label><input type="checkbox" name="gdpr" value="1"
              required=""> I agree to receive email updates and
              information.</label></p>
          <!-- no botz please -->
          <div style="position: absolute; left: -5000px;" aria-
              hidden="true"><input type="text" name="a_password"
              tabindex="-1" autocomplete="off"></div>
          <p><button type="submit">Subscribe</button></p>
        </form>
        <script src="assets/form.js"></script>
        <span class="smallprint">you can unsusbscribe at any time</
            span>
      </div>
    </div>
```

Listing F.17: newletter_widget.tpl

```
  <div class="sidebar-module">
    <h4>Recent Posts</h4>
    <ul class="sidebar-module-list">
        <li>
          <a href="${(base_url)!}/2021/01/05/hosting/">What can we
              do for a sustainable web?</a>
        </li>
        <li>
          <a href="${(base_url)!}/2020/11/23/what-is-sustainability
              /">What does sustainability mean in software
              development?</a>
        </li>
```

```
        <li>
          <a href="${(base_url)!}/2020/10/23/tomm-flowers-
              presentation/">Tommy Flowers 2020 presentation</a>
        </li>
        <li>
          <a href="${(base_url)!}/2020/10/13/tommy-flowers-autumn
              -2020/">The Tommy Flowers Network Autumn 2020
              Conference</a>
        </li>
        <li>
          <a href="${(base_url)!}/2020/09/13/make-a-difference/">
              Why do I think we can make a difference?</a>
        </li>
    </ul>
  </div>
```

Listing F.18: recent_module.tpl

```
    <article id="post-a-new-venture" class="article article-type-
        post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="${(base_url)!}/2020/09/04/a-
          new-venture/">A New Venture</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="${(base_url)!}/2020/09/04/a-new-venture/" class="article
      -date"><time datetime="2020-09-04T12:00:25.000Z" itemprop="
      datePublished">2020-09-04</time></a>
</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p>A first blog post is always a little bit uncomfortable,
            but we all have to start somewhere.</p>
<p>This is a blog in which I intend to explore the strange and
    often counterintuitive world of "<em>green</em>" (a.k.a "<em>
    sustainable</em>") programming and software development in
    general.</p>
<p>I feel as if there is a lot to say, so I hope I manage to get it
     written down. And I hope you will be along for the journey!</p>
    </div>
    <footer class="article-footer">
      <a data-url="${(base_url)!}/2020/09/04/a-new-venture/" data-
          id="cl14uzq6s00062amb92xpa24b" class="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="${(base_url)!}/2020/09/04/a-new-venture/#
            disqus_thread" class="article-comment-link">0 Comments</
            a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ${(base_url)!}/tags/Admin/" rel="tag">Admin</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.19: t2020_09_04_a_new_venture.tpl

```
    <article id="post-make-a-difference" class="article article-
       type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="${(base_url)!}/2020/09/13/make
         -a-difference/">Why do I think we can make a difference?</
         a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="${(base_url)!}/2020/09/13/make-a-difference/" class="
     article-date"><time datetime="2020-09-13T08:58:24.000Z"
     itemprop="datePublished">2020-09-13</time></a>
</div>
    <div class="article-author">Frank Carver</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="${(base_url)!}/images/jefferson-santos-9
           SoCnyQmkzI-unsplash.jpg" title="Photo credit: Jefferson
           Santos - Unsplash" class="image-link" rel="article4"><
           img src="assets/jefferson-santos-9SoCnyQmkzI-unsplash.
           jpg" alt="Photo credit: Jefferson Santos - Unsplash"
           title="Programmer"></a><span class="caption">Photo
           credit: Jefferson Santos - Unsplash</span></p>
<p>In 2019, <a target="_blank" rel="noopener" href="https://www.kth
   .se/en">KTH Royal Institute of Technology</a> in Sweden
   estimated that <strong>the internet uses over 10% of the world's
    electricity</strong>. This figure is still growing, too.
   Enforced isolation driven by the COVID19 pandemic has seen
   internet usage shoot up.</p>
<p>Naturally, there has been a lot of research on this topic, and I
    plan
 to explore some of it on this blog in future posts. However, most
    of it
 has concentrated on the <em>hardware</em> - the cables, routers,
computers, and data centres which comprise the physical parts of
   the
system. These are the bits that consume all this energy, after all,
    and
if their consumption can be improved it seems reasonable to assume
   that
the overall energy consumption of the internet as a system will
decrease.</p>
<p>Unfortunately, behind this level of obviousness are some things
   which get a lot less airtime:</p>
        <p class="article-more-link">
          <a class="btn btn-primary" href="${(base_url)
             !}/2020/09/13/make-a-difference/#more">Read More</a>
        </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/09/13/make-a-
         difference/" data-id="cl14uzq6z00092amb5t738df5" class="
         article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/09/13/make-a-
           difference/#disqus_thread" class="article-comment-link
           ">0 Comments</a>
```

```
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ${(base_url)!}/tags/Why/" rel="tag">Why</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.20: t2020_09_13_make_a_difference.tpl

```
    <article id="post-tommy-flowers-autumn-2020" class="article
        article-type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="${(base_url)!}/2020/10/13/
          tommy-flowers-autumn-2020/">The Tommy Flowers Network
          Autumn 2020 Conference</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="${(base_url)!}/2020/10/13/tommy-flowers-autumn-2020/"
      class="article-date"><time datetime="2020-10-13T16:06:14.000Z"
       itemprop="datePublished">2020-10-13</time></a>
</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="${(base_url)!}/images/tfnConf.jpg" title="Photo
             credit: Tommy Flowers Network" class="image-link" rel="
            article3"><img src="assets/tfnConf.jpg" alt="Photo
            credit: Tommy Flowers Network" title="Conference"></a><
            span class="caption">Photo credit: Tommy Flowers Network
            </span></p>
<p>The <a target="_blank" rel="noopener" href="http://
    tommyflowersnetwork.blogspot.com/" title="The Tommy Flowers
    Network">Tommy Flowers Network</a>
 is a partnership between industry and academia to bring together a
unique combination of theoretical, practical, and commercial
    approaches.
 Named after the GPO engineer <a target="_blank" rel="noopener"
     href="https://en.wikipedia.org/wiki/Tommy_Flowers">Tommy
     Flowers BSc DSc MBE</a>, who worked on the code breaking
     machines at <a target="_blank" rel="noopener" href="https://en.
     wikipedia.org/wiki/Bletchley_Park">Bletchley Park</a>
 during the Second World War, the Network aims to collaborate on
     solving
 some of the biggest social and technological problems of our age
     .</p>
<p>The Tommy Flowers Network is currently sponsored by <a target="
    _blank" rel="noopener" href="https://www.bt.com/">British
    Telecom</a> (BT) and hosts regular conferences at BT's <a target
    ="_blank" rel="noopener" href="https://atadastral.co.uk/">
    Adastral Park</a>
 research site. In 2020 the Autumn conference runs from 12th-16th
October and is being held fully online. I have been given the
opportunity to present my work at this conference on Wednesday 14
October in the form of a five-minute <em>"Lightning Talk"</em>.</p>
<p>I recommend you check out <a target="_blank" rel="noopener" href
    ="http://tommyflowersnetwork.blogspot.com/" title="The Tommy
    Flowers Network">Tommy Flowers Network</a> and the <a target="
    _blank" rel="noopener" href="http://tommyflowersnetwork.blogspot
```

```
    .com/2020/07/virtual-conference-lets-get-physical.html">Autumn
    2020 conference</a>.
 Even if you don't have a chance to attend live, I understand that
     the
presentations will be shared on their website after the event.</p>
<p>If you want a flavour of my talk ahead of time, feel free to
    download <a href="${(base_url)!}/downloads/2020-10-13-tfn.pptx">
    the PowerPoint slide deck</a> for the presentation.</p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/10/13/tommy-
          flowers-autumn-2020/" data-id="cl14uzq73000b2ambfd18fffu"
          class="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/10/13/tommy-
            flowers-autumn-2020/#disqus_thread" class="article-
            comment-link">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ${(base_url)!}/tags/Conferences/" rel="tag">Conferences</a></
      li><li class="article-tag-list-item"><a class="article-tag-
      list-link" href="${(base_url)!}/tags/Presentations/" rel="tag
      ">Presentations</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.21: t2020_10_13_tommy_flowers_autumn_2020.tpl

```
    <article id="post-tomm-flowers-presentation" class="article
        article-type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="${(base_url)!}/2020/10/23/tomm
          -flowers-presentation/">Tommy Flowers 2020 presentation</a
          >
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="${(base_url)!}/2020/10/23/tomm-flowers-presentation/"
      class="article-date"><time datetime="2020-10-23T15:27:15.000Z"
       itemprop="datePublished">2020-10-23</time></a>
</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p>At the recent <a target="_blank" rel="noopener" href="
            http://tommyflowersnetwork.blogspot.com/" title="The
            Tommy Flowers Network">Tommy Flowers Network</a>
 conference, I presented a five-minute session on my research as
     part of
 their "Lightning Talks". Despite several practice runs and
     trimming the
 presentation to fit the time slot, I still managed to hit the time
limit before I got to my final two slides.</p>
<p>After staying with the conference to watch the remaining
presentations for that day, I decided to re-record a version of the
     same
 presentation that I am calling the <em>Director's Cut</em>. In
```

```
    this one I take a bit more time to explain each step, with the
    result that it take a bit over ten minutes.</p>
<p>This is still pretty short, so even if you did see the rushed
   live
presentation, I encourage you to take a look at this one, to get
   the
full experience.</p>
<style>.embed-container {
    position: relative;
    padding-bottom: 56.25%;
    height: 0;
    overflow: hidden;
    max-width: 100%;
  }
  .embed-container iframe, .embed-container object, .embed-
     container embed {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
  }
  </style>
<div class="embed-container"><iframe src="assets/qn6IwZJvH50.html"
   allowfullscreen="" allow="accelerometer; autoplay; encrypted-
   media; gyroscope; picture-in-picture" frameborder="20"></iframe
   ></div>
<br>
          <p class="article-more-link">
            <a class="btn btn-primary" href="${(base_url)
                !}/2020/10/23/tomm-flowers-presentation/#more">Read
                More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/10/23/tomm-
          flowers-presentation/" data-id="cl14uzq71000a2amb948v63nz"
           class="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/10/23/tomm-
            flowers-presentation/#disqus_thread" class="article-
            comment-link">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ${(base_url)!}/tags/Conferences/" rel="tag">Conferences</a></
      li><li class="article-tag-list-item"><a class="article-tag-
      list-link" href="${(base_url)!}/tags/Presentations/" rel="tag
      ">Presentations</a></li><li class="article-tag-list-item"><a
      class="article-tag-list-link" href="${(base_url)!}/tags/Video
      /" rel="tag">Video</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.22: t2020_10_23_tomm_flowers_presentation.tpl

```
    <article id="post-what-is-sustainability" class="article
       article-type-post" itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
```

```
      <a class="article-title" href="${(base_url)!}/2020/11/23/what
          -is-sustainability/">What does sustainability mean in
          software development?</a>
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="${(base_url)!}/2020/11/23/what-is-sustainability/" class
      ="article-date"><time datetime="2020-11-23T14:58:24.000Z"
      itemprop="datePublished">2020-11-23</time></a>
</div>
    <div class="article-author">Frank Carver</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="${(base_url)!}/images/image1170x530cropped.jpg"
             title="Photo credit: The United Nations" class="image-
            link" rel="article1"><img src="assets/
            image1170x530cropped.jpg" alt="Photo credit: The United
            Nations" title="UN Sustainability Goals"></a><span class
            ="caption">Photo credit: The United Nations</span></p>
<p><em>Sustainability</em> is a complex term that is used in a lot
    of
different contexts, and has many different assumptions and
    associations
toed to it. In my research, <em>sustainability</em> is a vital
    concept
which underlies the research, and is found in many of the key
    papers in
the research literature. So there is no avoiding the question: what
     <em>does</em> sustainability mean in software development?</p>
          <p class="article-more-link">
            <a class="btn btn-primary" href="${(base_url)
                !}/2020/11/23/what-is-sustainability/#more">Read
                More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2020/11/23/what-is-
          sustainability/" data-id="cl14uzq7h000t2ambhjbx7uzj" class
          ="article-share-link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2020/11/23/what-is-
            sustainability/#disqus_thread" class="article-comment-
            link">0 Comments</a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
      article-tag-list-item"><a class="article-tag-list-link" href="
      ${(base_url)!}/tags/Why/" rel="tag">Why</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.23: t2020_11_23_what_is_sustainability.tpl

```
    <article id="post-hosting" class="article article-type-post"
        itemscope="" itemprop="blogPost">
  <header class="article-header">
    <h1 itemprop="name">
      <a class="article-title" href="${(base_url)!}/2021/01/05/
          hosting/">What can we do for a sustainable web?</a>
```

```
    </h1>
  </header>
  <div class="article-meta">
    <div class="article-datetime">
  <a href="${(base_url)!}/2021/01/05/hosting/" class="article-date
     "><time datetime="2021-01-05T17:22:06.000Z" itemprop="
     datePublished">2021-01-05</time></a>
</div>
    <div class="article-author">Frank Carver</div>
  </div>
  <div class="article-inner">
    <div class="article-entry" itemprop="articleBody">
        <p><a href="${(base_url)!}/images/
            IMG_20201128_143813208_trees-cropped.jpg" title="Photo
            credit: Frank Carver (c) 2020" class="image-link" rel="
            article0"><img src="assets/IMG_20201128_143813208_trees-
            cropped.jpg" alt="Photo credit: Frank Carver (c) 2020"
            title="Trees"></a><span class="caption">Photo credit:
            Frank Carver (c) 2020</span></p>
<p>As I have mentioned before on this blog, I am currently
    undertaking
research for a PhD in sustainable software. I don't have full
    results
from my experimentation so far, but the more I look into this area,
     and
the more that I see of the huge impact the internet has on the
environment, the more I want to start making changes <strong>right
    now</strong>.</p>
<p>This post covers some of the things I am doing, and some of the
    things <em>you</em> can do too, to help make our contribution to
     the internet "greener".</p>
          <p class="article-more-link">
            <a class="btn btn-primary" href="${(base_url)
                !}/2021/01/05/hosting/#more">Read More</a>
          </p>
    </div>
    <footer class="article-footer">
      <a data-url="https://greenprogrammer.net/2021/01/05/hosting/"
          data-id="cl14uzq6u00072amba8riggsv" class="article-share-
         link">
        <i class="fa fa-share"></i> Share
      </a>
        <a href="https://greenprogrammer.net/2021/01/05/hosting/#
            disqus_thread" class="article-comment-link">0 Comments</
            a>
  <ul class="article-tag-list" itemprop="keywords"><li class="
     article-tag-list-item"><a class="article-tag-list-link" href="
     ${(base_url)!}/tags/Hosting/" rel="tag">Hosting</a></li><li
     class="article-tag-list-item"><a class="article-tag-list-link"
      href="${(base_url)!}/tags/How/" rel="tag">How</a></li></ul>
    </footer>
  </div>
</article>
```

Listing F.24: t2021_01_05_hosting.tpl

```
  <div class="sidebar-module">
    <h4>Tag Cloud</h4>
    <p class="tagcloud">
      <a href="${(base_url)!}/tags/Admin/" style="font-size: 10px
         ;">Admin</a> <a href="${(base_url)!}/tags/Conferences/"
```

```
        style="font-size: 20px;">Conferences</a> <a href="${(
        base_url)!}/tags/Hosting/" style="font-size: 10px;">
        Hosting</a> <a href="${(base_url)!}/tags/How/" style="font
        -size: 10px;">How</a> <a href="${(base_url)!}/tags/
        Presentations/" style="font-size: 20px;">Presentations</a>
         <a href="${(base_url)!}/tags/Video/" style="font-size: 10
        px;">Video</a> <a href="${(base_url)!}/tags/Why/" style="
        font-size: 20px;">Why</a>
  </p>
</div>
```

Listing F.25: tagcloud_module.tpl

```
<div class="sidebar-module">
  <h4>Tags</h4>
  <ul class="sidebar-module-list" itemprop="keywords"><li class="
      sidebar-module-list-item"><a class="sidebar-module-list-link
      " href="${(base_url)!}/tags/Admin/" rel="tag">Admin</a><span
       class="sidebar-module-list-count">1</span></li><li class="
      sidebar-module-list-item"><a class="sidebar-module-list-link
      " href="${(base_url)!}/tags/Conferences/" rel="tag">
      Conferences</a><span class="sidebar-module-list-count">2</
      span></li><li class="sidebar-module-list-item"><a class="
      sidebar-module-list-link" href="${(base_url)!}/tags/Hosting
      /" rel="tag">Hosting</a><span class="sidebar-module-list-
      count">1</span></li><li class="sidebar-module-list-item"><a
      class="sidebar-module-list-link" href="${(base_url)!}/tags/
      How/" rel="tag">How</a><span class="sidebar-module-list-
      count">1</span></li><li class="sidebar-module-list-item"><a
      class="sidebar-module-list-link" href="${(base_url)!}/tags/
      Presentations/" rel="tag">Presentations</a><span class="
      sidebar-module-list-count">2</span></li><li class="sidebar-
      module-list-item"><a class="sidebar-module-list-link" href="
      ${(base_url)!}/tags/Video/" rel="tag">Video</a><span class="
      sidebar-module-list-count">1</span></li><li class="sidebar-
      module-list-item"><a class="sidebar-module-list-link" href="
      ${(base_url)!}/tags/Why/" rel="tag">Why</a><span class="
      sidebar-module-list-count">2</span></li></ul>
</div>
```

Listing F.26: tags_module.tpl

Glossary

artificial intelligence The application of software and data systems to process information and reason in a manner based on living intelligence. Machine learning based on neural networks is an approach to achieve artificial intelligence.

bare metal A term used when software runs on real rather than virtualised computer hardware. In embedded systems, the term 'bare metal' is also used when software runs without the need for an operating system.

black box A black box component is one which is used in whole and without the ability to modify the way it works. Black box component reuse is a process in which a component is reused without modification.

boilerplate Blocks of predefined text in a template which are included as-is in the output document when the template is rendered by a template engine.

classpath A classpath is an path-style environment variable containing a sequence of places for the Java Virtual Machine to look for class files when loading an application. The entries in the classpath may be directories in a file system, individual class files, or *JAR* archive files.

client A role in a distributed system which sends network requests or messages to a server.

client-server A distributed system in which computing elements have one of two roles: client or server.

client-side processing Code which executes on a client device, rather than requiring a request to a server.

cloud computing Any software and data systems which use resources in the cloud.

compiler A software tool which processes an input document through stages such as lexical analysis and parsing to produce a object model for subsequent use by a linker, loader, or interpreter..

computer science A field within computing which includes both theoretical disciplines such as algorithms, theory of computation, and information theory, and applied disciplines such as the design and implementation of hardware and software.

computer systems engineering A field within computing which applies engineering principles and computer programming expertise to develop, test, and maintain computer systems.

computing The term introduced by Denning et al. (1989) to include all aspects of the specification, architecture, design, construction, evaluation, maintenance, and management of solutions and products which use computer technology.

computing and sustainability One of the names given to the intersection of the disciplines of sustainability and computing.

computing sustainability One of the names given to the intersection of the disciplines of sustainability and computing.

consolidating A strategy for improving the operating costs of a computer system by sharing computing hardware between multiple systems.

containerisation An alternative to virtualisation in which multiple isolated software systems can share aspects of a single underlying operating system. Containerised applications are typically smaller than virtualised applications because they do not need to duplicate existing operating system features.

continuous integration A software technique in which software consisting of multiple collaborating parts is assembled and tested by an automated process. This process is performed whenever any of the component parts of the system are changed, to ensure that changes to individual parts do not affect the behaviour or performance of the larger system.

cross-compile Cross-compilation is a software development technique involving creating software on one machine, typically one set up as a development workstation, and using software tools to generate code which will run on a different device. Cross-compilation is commonly done when developing software for embedded systems.

CSV Comma-Separated Values, a textual data format consisting of one data record per line, with fields within each record separated by comma (`,`) characters. When used with data values which may contain comma characters, the enhanced QCSV format (which wraps data values in quotation marks) can be used..

dark web Information and services which are accessed using web technology but not listed in search data or linked from public pages. This makes such information largely invisible to general web users.

data science A field within computing concerned with representation, storage and manipulation of data.

datacenter A single building or collection of buildings providing shared support for a large amount of computing resources.

design pattern A description of a common software design approach. Each design pattern is phrased in terms of four elements: the pattern name, the problem it addresses, the solution it provides, and the consequences of adopting that solution (Gamma et al., 1994).

developer test A test created and used during software development to specify the behaviour of the software being developed. Developer tests are a key part of Test-Driven Development and are typically automated and run many times during software development. Also known as a unit test..

distributed system A computer system comprising more than one separate software or hardware node which communicate over a network.

dynamic scaling The process of automatically starting up and shutting down virtual servers or application containers to meet increases or decreases in demand.

eager evaluation A software technique in which calculation or processing of derived values is performed as soon as the input values are available. The main advantage of this approach is that derived values are already available when required, which can improve the responsiveness of a software application. The opposite approach to eager evaluation is lazy evaluation.

efficiency In computing, efficiency is commonly used in the sense that an efficient system is one which achieves an objective using minimal resources. The resources in question may include such things as time, energy, disk space, computer memory or any other scarce or costly resource. Where a specific

resource is considered, the term can add a modifier, for example energy-efficiency or memory-efficiency.

embedded A software system is described as 'embedded' when it is designed for a specific hardware configuration. Often the hardware and software for an embedded system are designed together.

embedded system A software system is described as 'embedded' when it is tied to, and often developed alongside, specific computer hardware. Embedded systems often run on bare metal without the need for an operating system, or with a custom or real time operating system.

functional requirements Requirements that a system must meet in order to be considered fit for purpose. Functional requirements can usually be phrased in a way which allows for a yes/no or pass/fail answer.

global warming Also known as 'climate change'. An observed increase in global average temperature over time.

green computing One of the names given to the intersection of the disciplines of sustainability and computing.

greenhouse effect A process in which infra-red radiation is reflected back to a planet surface by the composition of the atmosphere rather than escaping into space. This capturing of heat energy leads to an overall increase in temperature.

greenhouse gases A range of gases, including carbon dioxide and methane, which contribute to the greenhouse effect when released into the atmosphere.

HTTP Hypertext Transfer Protocol, the basic protocol used to transfer requests ans responses between web browsers and web servers.

HTTPS Hypertext Transfer Protocol Secure, an extension of the HTTP protocol in which requests and responses are encrypted.

hypervisor A form of operating system which acts to manage access to the bare metal of a computing system from multiple virtual machines.

ICT Information and Communication Technology, a collective term for the electronic and computer systems which enable the internet and the World-Wide Web as well as stand-alone computerised devices and the internet of things.

informatics A field within computing concerned with representation and transformation of information.

information science A field within computing concerned with analysis, collection, classification, manipulation, storage, retrieval, movement, dissemination, and protection of information.

internet A massive distributed system which enables a wide range of services including email, social media, digital telephony, and the World-Wide Web.

internet of things The addition of sensors and other non-interactive devices to the internet to gather, interpret and transfer information as well as operate internet-connected equipment.

interpreter A software tool which reads and processes an input document representing a program or other series of instructions and executed those instructions immediately. Note that the input document for an interpreter may be textual, requiring parsing, or it may be an object file or other representation of a object model..

intranet A section of the World-Wide Web protected behind a login or other security approach and therefore inaccessible to external web browsers.

JavaBeans A feature of the Java programming language that allows some object methods to be accessed as if they are fields. To be accessible as a JavaBean, a class must have a zero-argument constructor and methods with specific name patterns. See `https://docs.oracle.com/javase/8/docs/technotes/guides/beans/index.html`.

lazy evaluation A software technique in which calculation or processing of derived values is deferred until the values are used. The main advantage of this approach is that derived values never need to be calculated if they are never used. This can be particularly significant if a calculation involves a relatively slow process such as requesting information from a remote system. The opposite approach to lazy evaluation is eager evaluation.

linker A software tool which takes one or more object models such as programs or libraries and combines them into a single object model for use by a loader or interpreter..

loader A software tool which transforms a generic object model into instructions and data for a specific computer system..

Model View Controller A software design pattern that divides an application into three key areas: The *model*, which represents the

information and state contained in the application; the *view*, which presents output to and interprets input and commands from a user; and the *controller*, which converts input and commands into actions on the model and produces output to be displayed by the view..

non-functional requirements Aims that a system should meet in order to be considered fit for purpose. Non-functional requirements are often vague or aspirational and can be difficult to measure.

object file An output file generated by a compiler containing an external representation of a object model for subsequent use by a linker, loader, or interpreter..

object model A symbolic version of an input document produced by a compiler..

operating system A software system which provides services to applications, isolating them from the specifics of the underlying computing hardware. Current common operating systems include Windows, MacOS, and Linux.

page context A collection of named data items made available to a template engine to provide values for placeholder value expressions in a template..

performance In computing, performance is usually used to refer to speed. In software, performance relates to the speed of the software in performing a given task. In hardware, performance is usually related to the speed at which the hardware can operate. Performance is inversely related to the time taken to perform a task.

placeholder An indication in a template of where, and potentially how, to include data provided when the template is rendered by a template engine.

power usage effectiveness (PUE) An indication of the proportion of consumed energy used above that needed for the desired work.

programming language A language used to instruct computing devices on what to do and how to do it. programming languages are usually textual but may also be graphical (e.g. Scratch[1]) or symbolic (e.g. APL (Falkoff and Iverson, 1978)).

QCSV Quoted Comma-Separated Values, a textual data format consisting of one data record per line, with fields within each record wrapped in quotation marks and separated by comma (`,`) characters. When used with data values which cannot contain comma characters, the simpler CSV format (which avoids the need to wrap data values in quotation marks) can be used..

[1] `https://scratch.mit.edu/`

real time A term used to describe software optimised for immediate response to events and stimuli. Common operating systems such as Windows or Linux are not classed as 'real time' as they use process scheduling algorithms which can cause delays or interruptions to application software.

refactoring A process of changing the implementation of a system without changing its behaviour. Typically this is done to simplify or otherwise improve the maintainability of the system ready for future requirements or additional features. Refactoring is possible on any software system, but is considerably easier and safer if the system has a robust suite of conformance tests which can catch any accidental errors..

Remote Code Execution A software system security threat characterised by an ability to execute code on one system from another. Often abbreviated to RCE..

server A role in a distributed system which receives and responds to network requests or messages from a client.

Server-Side Template Injection A software system security threat characterised by an ability to remotely inject dangerous values into the page context of a trusted template. Often abbreviated to SSTI..

side-effects In a template language, a side-effect is anything which happens as a result of a template placeholder expression other than the production of text for the output document. Examples might be modifying the operation of other placeholder expressions, controlling the operation of the template engine as a whole, or executing code on the underlying computer system.

soft requirements An alternative name for non-functional requirements..

software development A field within computing which includes conceiving a goal, evaluating feasibility, analysing requirements, design, implementation, testing and release management.

software engineering A field within computing which applies engineering principles and computer programming expertise to develop, test, and maintain software applications.

speed In computing, speed is often used interchangeably with performance. Speed is conceptually easy to measure for tasks that have a defined start and end - a faster task completes in less time. Speed is less easy to measure for services or applications that run until manually stopped.

strategy pattern A software design pattern in which alternate implementations of a solution can be treated interchangeably by sharing a common abstraction.

sustainability The ability of something to endure. Used in this dissertation to refer to the specific sustainability of human life and our supportive ecosystems on Earth.

sustainability ledger A metaphorical 'balance sheet' of forces and actions which improve and/or worsen global sustainability.

sustainable computing One of the names given to the intersection of the disciplines of sustainability and computing.

template A specification of an output document consisting of boilerplate text and placeholders.

template engine Also sometimes known as a template processor, *templating engine*, or *templater*, this is a software application or library which combines a generic template with dynamic data to create a composite output document.

template language The symbols, grammar and linguistic elements which are allowable in a template for a particular template engine. Typically a template language specifies the syntax for boilerplate and placeholders within the template document.

template processor Another name for a template engine. For consistency, this thesis always uses the term template engine.

Test-Driven Development Often abbreviated to *TDD*. A software development process which uses automated developer tests to specify and verify software behaviour. In this process, developer tests are created before the software to be tested is written, and development is not considered complete until all such tests pass..

the cloud An abstraction model which treats multiple processing and data storage systems, typically located in large datacenters, as a single bank of assignable resources.

UN Sustainable Development Goals A set of 17 'goals' aimed at improving the sustainability of humanity and the environment.

unit test A test created and used during software development to specify the behaviour of the software being developed. Developer tests are a key part

of Test-Driven Development and are typically automated and run many times during software development. Also known as a developer test..

upgrading A strategy for improving the performance or operating costs of a computer system by replacing computing hardware with newer, more efficient, components.

virtual machine An installation of an operating system and applications running alongside other virtual machines under the control of a hypervisor.

virtualisation A strategy for consolidating computer systems by running multiple virtual machines under the control of a hypervisor.

virtualised A term used when a computer system executes in a virtual machine rather than on bare metal hardware.

web application A software application which makes use of the World-Wide Web. Web applications are typically distributed systems making use of one or more servers as well as web browsers.

web browser Software which enables users to access (also known as *browse* or *surf*) the information available on the World-Wide Web.

white box A white box component is one which is provided in the form of source code which may be modified or used in part. White box component reuse is a process in which a component is used as a source of code for modification..

World-Wide Web A distributed information system enabled by the internet and accessed using a web browser.

zombie In this context a zombie is a slang term for a computer system or application which continues operation even though it is no longer used by humans or other computer systems.

References

Abdalkareem, Rabe et al. (2017). "Why do developers use trivial packages? an empirical case study on npm". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 385–395.

Adafruit (2023). *Adafruit INA260 High or Low Side Voltage, Current, Power Sensor : ID 4226*. URL: `https://www.adafruit.com/product/4226` (visited on 09/22/2023).

Adelmeyer, Michael et al. (2017). "Rebound effects in cloud computing: towards a conceptual framework". In: *13th International Conference on Wirtschaftsinformatik*. St. Gallen, Switzerland.

Adenowo, Adetokunbo A A and Basirat A Adenowo (2020). "Software Engineering Methodologies: A Review of the Waterfall Model and Object-Oriented Approach". In: *International Journal of Scientific and Engineering Research* 4.7.

Aggarwal, Karan, Abram Hindle, and Eleni Stroulia (2015). "GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption". In: *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, pp. 311–320. ISBN: 1-4673-7532-2.

Ahmad, Raja Wasim et al. (2015). "A Review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues". In: *Journal of Network and Computer Applications* 58, pp. 42–59. ISSN: 1084-8045.

Aho, Alfred V. et al., eds. (2007). *Compilers: principles, techniques, & tools*. 2. ed., Pearson internat. ed. Boston Munich: Pearson Addison-Wesley. ISBN: 978-0-321-48681-3.

Alidoosti, Razieh, Patricia Lago, and Maryam Razavian (2022). *Ethics in Software Engineering: a Systematic Literature Review*. Tech. rep. Vrije Universiteit Amsterdam.

Allen, T F H and Thomas W Hoekstra (1993). "Toward a Definition of Sustainability". In: *Sustainable ecological systems: implementing an ecological approach to land management*. Fort Collins, Colorado.

Amazon (2024). *AWS Well-Architected Framework*. URL: https://aws.amazon.com/architecture/well-architected/ (visited on 03/18/2024).

Ameller, David et al. (2012). "How do software architects consider non-functional requirements: An exploratory study". In: *2012 20th IEEE international requirements engineering conference (RE)*. IEEE, pp. 41–50. ISBN: 1-4673-2785-9.

Andersen, Jacob and Morten Tranberg Hansen (2009). "Energy bucket: A tool for power profiling and debugging of sensor nodes". In: *2009 Third International Conference on Sensor Technologies and Applications*. IEEE, pp. 132–138. ISBN: 0-7695-3669-7.

Andreolini, Mauro and Sara Casolari (2006). "Load prediction models in Web-based systems". In: *Valuetools '06*. Pisa, Italy.

Androutsellis-Theotokis, S et al. (2011). "Open Source Software: A Survey from 10, 000 Feet." In: *Foundations and Trends in Technology, Information and Operations Management* 4.3-4, pp. 187–347.

Anwar, Hina et al. (2020). "Should energy consumption influence the choice of android third-party http libraries?" In: *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 87–97.

Ardi, Calvin, Alefiya Hussain, and Stephen Schwab (2021). "Building Reproducible Video Streaming Traffic Generators". In: *Cyber Security Experimentation and Test Workshop*. Virtual CA USA: ACM, pp. 91–95. ISBN: 978-1-4503-9065-1.

Ardito, Luca and Marco Torchiano (2018). "Creating and evaluating a software power model for linux single board computers". In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp. 1–8. ISBN: 978-1-4503-5732-6.

ARM (2023). *Arm CPU Architecture – Arm®*. URL: https://www.arm.com/architecture/cpu (visited on 09/22/2023).

Arnold, Ken (1999). "The Jini™ Architecture: Dynamic Services in a Flexible Network". In: *DAC 99*. New Orleans, Louisiana.

Arnoldus, B J, M G J van den Brand, and A Serebrenik (2012). "Less is More: Unparser-completeness of Metalanguages for Template Engines". In: *SIGPLAN Notices* 47.3. ISBN: 9781450306898, pp. 137–146. ISSN: 15232867.

Arnoldus, Bastiaan Jeroen (2010). "An Illumination of the Template Enigma: Software Code Generation with Templates". PhD thesis. Technische Universiteit Eindhoven. ISBN: 978-90-386-2418-1.

Arnoldus, Jeroen, Jeanot Bijpost, and Mark G.J. van den Brand (2007). "Repleo: a syntax-safe template engine". In: *International Conference on*

Generative Programming and Component Engineering (GPCE), p. 25. ISBN: 978-1-59593-855-8.

Astudillo-Salinas, Fabian et al. (2016). "Minimizing the power consumption in Raspberry Pi to use as a remote WSN gateway". In: *2016 8th IEEE Latin-American Conference on Communications (LATINCOM)*. Medellin, Colombia: IEEE, pp. 1–5.

Badampudi, Deepika, Claes Wohlin, and Kai Petersen (2016). "Software component decision-making: In-house, OSS, COTS or outsourcing - A systematic literature review". In: *The Journal of Systems and Software* 121, pp. 105–124.

Badampudi, Deepika et al. (2017). "A Decision-making Process-line for Selection of Software Asset Origins and Components". In: *The Journal of Systems & Software* 135.17, pp. 164–1212.

Baek, Dusan, Jae-Hyeon Park, and Jung-Won Lee (2018). "An energy efficiency grading system for mobile applications based on usage patterns". In: *The Journal of Supercomputing* 74, pp. 6502–6515. ISSN: 0920-8542.

Balanza-Martinez, Jose, Patricia Lago, and Roberto Verdecchia (2023). *Tactics for Software Energy Efficiency: A Review*. Tech. rep. Vrije Universiteit Amsterdam.

Ballhausen, Miriam (2019). "Free and open source software licenses explained". In: *Computer* 52.6, pp. 82–86. ISSN: 0018-9162.

Barbier, E B, A Markandya, and D W Pearce (1990). "Environmental Sustainability and Cost-Benefit Analysis". In: *Environment and Planning A* 22, pp. 1259–1266.

Barbosa, Denilson et al. (2002). "ToXgene". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*. ISBN: 1581134975, p. 616.

Bashroush, Rabih, Eoin Woods, and Adel Noureddine (2016). "ICT Energy Demand: what got us here won't get us there!" In: *IEEE Software* 33.2, pp. 18–21.

Basili, Victor R, Lionel C Briand, and Walcélio L Melo (1996). "How reuse influences productivity in object-oriented systems". In: *Communications of the ACM* 39.10, pp. 104–116. ISSN: 0001-0782.

Basmadjian, Robert and Hermann De Meer (2012). "Evaluating and modeling power consumption of multi-core processors". In: *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*. Madrid Spain: ACM, pp. 1–10. ISBN: 978-1-4503-1055-0.

Bauer, Veronika and Lars Heinemann (2012). "Understanding API usage to support informed decision making in software maintenance". In: *2012 16th*

European Conference on Software Maintenance and Reengineering. IEEE, pp. 435–440. ISBN: 0-7695-4666-8.

Beattie, Geoffrey (2010). *Why aren't we saving the planet?: a psychologist's perspective*. Routledge. ISBN: 1-136-88908-6.

Beck, Kent (2000a). *Extreme programming explained: embrace change*. Addison-Wesley Professional. ISBN: 0-201-61641-6.

— (2000b). *Test driven development: By example*. Addison-Wesley Professional. ISBN: 0-13-758523-3.

Beck, Kent et al. (2001). *The agile manifesto*. Tech. rep.

Becker, Christoph (2023). *Insolvent: how to reorient computing for just sustainability*. Cambridge, Massachusetts: The MIT Press. ISBN: 978-0-262-37466-8.

Becker, Christoph et al. (2015). "Sustainability Design and Software: The Karlskrona Manifesto". In: *28th Intl. Conf. on Software Maintenance*. Vol. 2, pp. 467–476. ISBN: 978-1-4799-1934-5.

Behrouz, Reyhaneh Jabbarvand et al. (2015). "Ecodroid: An approach for energy-based ranking of android apps". In: *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. IEEE, pp. 8–14. ISBN: 1-4673-7049-5.

Bekaroo, Girish and Aditya Santokhee (2016). "Power consumption of the Raspberry Pi: A comparative analysis". In: *2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies, EmergiTech 2016*. Institute of Electrical and Electronics Engineers Inc., pp. 361–366. ISBN: 978-1-5090-0706-6.

Belkhir, Lotfi and Ahmed Elmeligi (2018). "Assessing ICT global emissions footprint: Trends to 2040 & recommendations". In: *Journal of Cleaner Production* 177, pp. 448–463. ISSN: 09596526.

Berners-Lee, T (1996). *RFC 1945: Hypertext Transfer Protocol – HTTP/1.0*. URL: `https://www.rfc-editor.org/rfc/rfc1945` (visited on 11/16/2022).

Berners-Lee, Tim et al. (1992). "World-Wide Web: the information universe". In: *Electronic networks* 2.1, pp. 52–58. ISSN: 1066-2243.

Bertoa, Manuel F, José M Troya, and Antonio Vallecillo (2003). "A survey on the quality information provided by software component vendors". In: *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, pp. 25–30.

Betz, Stefanie et al. (2015). "Sustainability debt: A metaphor to support sustainability design decisions". In: *Fourth International Workshop on Requirements Engineering for Sustainable Systems (RE4SuSy)*. Ottawa, Canada.

Betz, Stefanie et al. (2024). "Lessons Learned from Developing a Sustainability Awareness Framework for Software Engineering using Design Science". In:

ACM Transactions on Software Engineering and Methodology, p. 3649597. ISSN: 1049-331X, 1557-7392.

Bissyande, Tegawende F. et al. (2013). "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects". In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. Kyoto, Japan: IEEE, pp. 303–312. ISBN: 978-0-7695-4986-6.

Bjarnason, Elizabeth, Patrik Åberg, and Nauman bin Ali (2023). "Software selection in large-scale software engineering: A model and criteria based on interactive rapid reviews". In: *Empirical Software Engineering* 28.2, p. 51.

Blackburn, Stephen M, Perry Cheng, and Kathryn S McKinley (2004). "Myths and realities: The performance impact of garbage collection". In: *ACM SIGMETRICS Performance Evaluation Review* 32.1, pp. 25–36. ISSN: 0163-5999.

Boehm, Barry and Chris Abts (1999). "COTS integration: Plug and pray?" In: *Computer* 32.1, pp. 135–138. ISSN: 0018-9162.

Bogus, Andre (2008). *Lighttpd*. Packt Publishing Ltd. ISBN: 1-84719-211-4.

BoldGrid (2022). *W3 Total Cache – WordPress plugin*. URL: `https://wordpress.org/plugins/w3-total-cache/` (visited on 11/16/2022).

Brett, Matthew et al. (2001). "Using the talairach atlas with the MNI template". In: *NeuroImage* 13.6, p. 85. ISSN: 10538119.

Bretthauer, David (2001). *Open Source Software: A History*. en. Tech. rep. University of Connecticut.

Brooks, Fred (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley.

Brown, Len et al. (2006). "Linux Laptop Battery Life Measurement Tools, Techniques, and Results". In: *Proceedings of the Linux Symposium*. Vol. 1. Ottawa, Canada, pp. 127–146.

Brundtland, Gro Harlem (1987). *Report of the World Commission on Environment and Development: Our Common Future*. Tech. rep. United Nations World Commission on Environment and Development.

Bunse, Christian and Sebastian Stiemer (2013). "On the energy consumption of design patterns". In: *Softwaretechnik-Trends* 33.2.

Bush, Vannevar (1945). "As We May Think". In: *The Atlantic Monthly*, pp. 101–108.

Caldwell, Nicholas Hugh Mullen (1998). "Knowledge-Based Engineering for the Scanning Electron Microscope". PhD thesis. Trinity College, Cambridge.

Calero, Coral, Manuel F. Bertoa, and Ma Angeles Moraga (2013). "A systematic literature review for software sustainability measures". In: *2013 2nd International Workshop on Green and Sustainable Software, GREENS 2013 - Proceedings*. IEEE, pp. 46–53. ISBN: 978-1-4673-6267-2.

Campbell-Kelly, M. et al. (2023). *Computer: A History of the Information Machine*. Taylor & Francis. ISBN: 978-1-000-87875-2.

Canek, Rodrigo, Pedro Borges, and Chantal Taconet (2022). "Analysis of the Impact of Interaction Patterns and IoT Protocols on Energy Consumption of IoT Consumer Applications". In: *Distributed Applications and Interoperable Systems*. Ed. by David Eyers and Spyros Voulgaris. Vol. 13272. Cham: Springer International Publishing, pp. 131–147.

Cao, Jingcun, Pradeep Chintagunta, and Shibo Li (2023). "From Free to Paid: Monetizing a Non-Advertising-Based App". In: *Journal of Marketing Research* 60.4, pp. 707–727. ISSN: 0022-2437.

Capra, Eugenio, Chiara Francalanci, and Sandra A Slaughter (2012). "Is software "green"? Application development environments and energy efficiency in open source applications". In: *Information and Software Technology* 54.1, pp. 60–71. ISSN: 0950-5849.

Carvalho, Fernando Miguel, Luis Duarte, and Julien Gouesse (2020). "Text Web Templates Considered Harmful". en. In: *Web Information Systems and Technologies*. Ed. by Alessandro Bozzon, Francisco José Domínguez Mayo, and Joaquim Filipe. Vol. 399. Series Title: Lecture Notes in Business Information Processing. Cham: Springer International Publishing, pp. 69–95. ISBN: 978-3-030-61749-3 978-3-030-61750-9. DOI: 10.1007/978-3-030-61750-9_4. URL: http://link.springer.com/10.1007/978-3-030-61750-9_4 (visited on 02/25/2026).

Castillo, Lorena (2023). *Average Website Traffic Statistics*. Tech. rep. Gitnux. URL: https://blog.gitnux.com/average-website-traffic-statistics/ (visited on 10/13/2023).

Chari, Suresh, Josyula R. Rao, and Pankaj Rohatgi (2003). "Template Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 13–28. ISBN: 978-3-540-36400-9.

Chen, FeiFei et al. (2012). "An energy consumption model and analysis tool for cloud computing environments". In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. IEEE, pp. 45–50. ISBN: 1-4673-1832-9.

Chen, Xingru et al. (2022). "Reuse in Contemporary Software Engineering Practices – An Exploratory Case Study in A Medium-sized Company". In: *e-Informatica Software Engineering Journal* 16.1, p. 220110. ISSN: 2084-4840.

Chien, Andrew A (2021). "Driving the Cloud to True Zero Carbon". In: *Communications of the ACM* 64.2.

Chitchyan, Ruzanna et al. (2016). "Sustainability design in requirements engineering: state of practice". In: *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 533–542.

Chowdhury, Shaiful Alam et al. (2015). "A system-call based model of software energy consumption without hardware instrumentation". In: *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, pp. 1–6. ISBN: 1-5090-0172-7.

Christel, Michael G and Kyo C Kang (1992). *Issues in requirements elicitation*. Tech. rep. Carnegie Mellon University, Software Engineering Institute.

Cinelli, Matteo et al. (2021). "The echo chamber effect on social media". In: *Proceedings of the National Academy of Sciences* 118.9, e2023301118. ISSN: 0027-8424.

Cisco (2020). *Cisco Annual Internet Report (2018–2023)*.

Collins, Eliane Figueiredo and Vicente Ferreira De Lucena (2012). "Software Test Automation practices in agile development environment: An industry experience report". In: *2012 7th International Workshop on Automation of Software Test (AST)*. Zurich, Switzerland: IEEE, pp. 57–63.

Colmant, Maxime et al. (2018). "The next 700 CPU power models". In: *Journal of Systems and Software* 144, pp. 382–396. ISSN: 0164-1212.

Condori-Fernandez, Nelly and Patricia Lago (2015). "Can we know upfront how to prioritize quality requirements?" In: *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE)*. IEEE, pp. 33–40. ISBN: 1-5090-0116-6.

— (Mar. 2018). "Characterizing the contribution of quality requirements to software sustainability". In: *Journal of Systems and Software* 137, pp. 289–305. ISSN: 01641212.

Connolly Bree, Déaglán and Mel Ó Cinnéide (2020). "Inheritance versus delegation: which is more energy efficient?" In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 323–329.

Copeland, B.J. (2004). "Colossus: its origins and originators". In: *IEEE Annals of the History of Computing* 26.4, pp. 38–45. ISSN: 1058-6180.

Coroama, Vlad C. and Lorenz M. Hilty (2009). "Energy Consumed vs. Energy Saved by ICT-A Closer Look." In: *EnviroInfo 2009*, pp. 347–355.

Couto, Marco et al. (2020). "On energy debt: managing consumption on evolving software". In: *Proceedings of the 3rd International Conference on Technical Debt*, pp. 62–66.

Cusumano, Michael A and Stanley Smith (1995). *Beyond the Waterfall: Software Development at Microsoft*.

Data, Mahendra, Muhammad Luthfi, and Widhi Yahya (2017). "Optimizing single low-end LAMP server using NGINX reverse proxy caching". In: *2017*

International Conference on Sustainable Information Engineering and Technology (SIET). Malang: IEEE, pp. 21–23.

Davies, Julius et al. (2013). "Software bertillonage: Determining the provenance of software development artifacts". In: *Empirical Software Engineering* 18, pp. 1195–1237. ISSN: 1382-3256.

De Freitas Netto, Sebastião Vieira et al. (Dec. 2020). "Concepts and forms of greenwashing: a systematic review". In: *Environmental Sciences Europe* 32.1, p. 19. ISSN: 2190-4707, 2190-4715.

De La Mora, Fernando López and Sarah Nadi (2018). "Which library should I use? A metric-based comparison of software libraries". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pp. 37–40.

Denning, P.J. et al. (1989). "Computing as a discipline". In: *Computer* 22.2, pp. 63–70. ISSN: 0018-9162.

derBauer (2023). *Intel HEDT is BACK! Insane Overclockable 56 Core Sapphire Rapids CPU*. Video file. URL: `https://www.youtube.com/watch?v=LzvF7xZbcaI` (visited on 09/08/2023).

Dezfouli, Behnam, Immanuel Amirtharaj, and Chia-Chi Chelsey Li (2018). "EMPIOT: An energy measurement platform for wireless IoT devices". In: *Journal of Network and Computer Applications* 121, pp. 135–148. ISSN: 1084-8045.

Dick, Markus and Stefan Naumann (2010). "Enhancing Software Engineering Processes towards Sustainable Software Product Design." In: *EnviroInfo*, pp. 706–715.

Dick, Markus et al. (2013). "Green software engineering with agile methods". In: *2013 2nd International Workshop on Green and Sustainable Software, GREENS 2013 - Proceedings*. IEEE, pp. 78–85. ISBN: 978-1-4673-6267-2.

Dixit, Varun and Davinderjit Kaur (2024). "A Systematic Review for Sustainable Software Development Practice and Paradigm". In: *Journal of Computational Analysis and Applications* 33.6.

Do, Thanh, Suhib Rawshdeh, and Weisong Shi (2009). *ptop: A process-level power profiling tool*.

Dorkal, Michal (2023). *From Runtime Efficiency to Carbon Efficiency - InfoQ*. London. URL: `https://www.infoq.com/presentations/slang/` (visited on 02/09/2024).

Drescher, Florian and Alexis Engelke (2024). "Fast Template-Based Code Generation for MLIR". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. Edinburgh United Kingdom: ACM, pp. 1–12. ISBN: 979-8-4007-0507-6.

Dutta, Prabal et al. (2008). "Energy metering for free: Augmenting switching regulators for real-time monitoring". In: *2008 International Conference on*

Information Processing in Sensor Networks (ipsn 2008). IEEE, pp. 283–294. ISBN: 0-7695-3157-1.

Dzhagaryan, Armen et al. (2016). "An environment for automated measurement of energy consumed by mobile and embedded computing devices". In: *Measurement* 94, pp. 103–118. ISSN: 0263-2241.

EESI (2022). *The Growth in Greenhouse Gas Emissions from Commercial Aviation (2019, revised 2022)*. URL: https://www.eesi.org/papers/view/fact-sheet-the-growth-in-greenhouse-gas-emissions-from-commercial-aviation (visited on 02/04/2024).

Eickhoff, Jens (2011). *Onboard computers, onboard software and satellite operations: an introduction*. Springer Science & Business Media. ISBN: 3-642-25170-6.

Erz, Hendrik (2023). *Coding and Keyboards*. URL: https://www.hendrik-erz.de/post/coding-and-keyboards (visited on 11/17/2023).

Evans Data Corporation (2018). *Global Developer Population and Demographic Study 2018 Vol. 2*. URL: https://evansdata.com/reports/viewRelease.php?reportID=9 (visited on 05/26/2018).

Falk, Sophia and Aimee Van Wynsberghe (2023). "Challenging AI for Sustainability: what ought it mean?" In: *AI and Ethics*. ISSN: 2730-5953, 2730-5961.

Falkoff, Adin D and Kenneth E Iverson (1978). "The Evolution of APL". In: *ACM SIGPLAN Notices* 13, pp. 45–57.

Fan, Ming, Subodha Kumar, and Andrew B. Whinston (2009). "Short-term and long-term competition between providers of shrink-wrap software and software as a service". In: *European Journal of Operational Research* 196.2, pp. 661–671. ISSN: 03772217.

Farkas, Keith I et al. (2000). "Quantifying the energy consumption of a pocket computer and a Java virtual machine". In: *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 252–263.

FastHosts (2023). *Dedicated Servers*. URL: https://www.fasthosts.co.uk/ppc/dedicated-servers (visited on 09/27/2023).

Field, Hayden, Glen Anderson, and Kerstin Eder (2014). "EACOF: A Framework for Providing Energy Transparency to enable Energy-Aware Software Development". In: *SAC '14: Proceedings of the 29th Annual ACM Symposium on Applied Computing*. arXiv: 1406.0117, pp. 1194–1199.

Fielding, Roy Thomas (2000). "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine.

Fitzgerald, Brian (2024). *Information Systems and Software Engineering: The Case for Convergence*.

Fleischman, W (2010). "Electronic voting systems and the therac-25: What have we learned". In: *The "Backwards, Forwards, and Sideways Changes" of ICT, Proceedings of ETHICOMP*, pp. 170–179.

Flinn, Jason and Mahadev Satyanarayanan (1999). "Powerscope: A tool for profiling the energy usage of mobile applications". In: *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, pp. 2–10. ISBN: 0-7695-0025-0.

Fonseca, Alcides, Rick Kazman, and Patricia Lago (2019). "A Manifesto for Energy-Aware Software". In: *IEEE Software* 36.December, pp. 79–82. ISSN: 19374194.

Foster, C et al. (2009). "Pattern of developing the performance template". In: *British Journal of Sports Medicine* 43.10, pp. 765–769. ISSN: 0306-3674, 1473-0480.

Fowler, M. and K. Beck (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley. ISBN: 978-0-201-48567-7.

Fowler, Martin (2012). *Test Pyramid*. URL: `https://martinfowler.com/bliki/TestPyramid.html` (visited on 10/05/2023).

Frakes, William B. and Sadahiro Isoda (1994). "Success factors of systematic reuse". In: *IEEE software* 11.5, pp. 14–19. ISSN: 0740-7459.

Freitag, Charlotte et al. (Sept. 2021). "The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations". In: *Patterns* 2.9, p. 100340. ISSN: 26663899.

Freitas, Artur and Renata Vieira (2014). "An Ontology for Guiding Performance Testing". In: *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. Warsaw, Poland: IEEE, pp. 400–407. ISBN: 978-1-4799-4143-8.

Fritzson, Peter et al. (2009). "Towards a Text Generation Template Language for Modelica". In: *Proceedings 7th Modelica Conference, Como, Italy, Sep. 20-22, 2009 Towards*, pp. 193–207.

Gacek, Cristina, ed. (2002). *Software Reuse: Methods, Techniques, and Tools*. Springer. ISBN: 978-3-540-22335-1.

Gamma, Erich et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley.

Gancarz, Rafal (2023). *The Frugal Architect: AWS Promotes Cost Awareness for Sustainability*. URL: https://www.infoq.com/news/2023/12/frugal-architect-werner-vogels/ (visited on 12/21/2023).

García, Félix et al. (2006). "Towards a consistent terminology for software measurement". In: *Information and Software technology* 48.8, pp. 631–644. ISSN: 0950-5849.

Garwood, G. J. (1997). "Work Manager". In: *BT Technology Journal* 15.1, pp. 58–68. ISSN: 1573-1995.

Ge, Rong et al. (2009). "Powerpack: Energy profiling and analysis of high-performance systems and applications". In: *IEEE Transactions on Parallel and Distributed Systems* 21.5, pp. 658–671. ISSN: 1045-9219.

Gelenbe, Erol (Aug. 2023). "Electricity Consumption by ICT: Facts, trends, and measurements". In: *Ubiquity* 2023.August, pp. 1–15. ISSN: 1530-2180. (Visited on 02/03/2025).

George, Boby and Laurie Williams (2004). "A structured experiment of test-driven development". In: *Information and Software Technology* 46.5, pp. 337–342. ISSN: 09505849.

Georges, Andy, Dries Buytaert, and Lieven Eeckhout (2007). "Statistically Rigorous Java Performance Evaluation General". In: *OOPSLA 07*.

Gibson, David, Kunal Punera, and Andrew Tomkins (2005). "The volume and evolution of web page templates". In: *Special interest tracks and posters of the 14th international conference on World Wide Web*. ISBN: 1-59593-051-5, pp. 830–839.

GitHub (2023). *GitHub: Let's build from here*. URL: https://github.com/ (visited on 08/15/2023).

Giungato, Pasquale et al. (2017). "Current trends in sustainability of bitcoins and related blockchain technology". In: *Sustainability (Switzerland)* 9.12. ISBN: 2071-1050. ISSN: 20711050.

Glass, Robert L and Addison Wesley (2002). *Facts and Fallacies of Software Engineering*. Addison Wesley. ISBN: 0-321-11742-5.

Goetz, Jozef and Antonio Flores Marquez (2023). "Web Framework". In: *International Journal on Engineering, Science & Technology (IJonEST)* 5.4. ISSN: 2642-4088.

Gonzalez-Barahona, Jesus M. (2021). "A Brief History of Free, Open Source Software and Its Communities". In: *Computer* 54.2, pp. 75–79. ISSN: 0018-9162, 1558-0814.

Goodland, Robert (2002). "Sustainability: human, social, economic and environmental". In: *Encyclopedia of global environmental change* 5, pp. 481–491.

Gossart, Cédric (2015). "Rebound effects and ICT: a review of the literature". In: *ICT innovations for sustainability*, pp. 435–448. ISSN: 3319092278.

Gottschalk, Marion et al. (2012). "Removing energy code smells with reengineering services". In: *INFORMATIK 2012*. ISSN: 3885796023.

Grafana (2023). *Grafana: The open observability platform | Grafana Labs*. URL: https://grafana.com/ (visited on 09/22/2023).

Green Software Fioundation (2024). *Green Software Foundation*. URL: https://greensoftware.foundation/ (visited on 03/31/2024).

Gu, Dayong, Clark Verbrugge, and Etienne M Gagnon (2006). "Relative factors in performance analysis of Java virtual machines". In: *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 111–121.

Gu, Qing, Patricia Lago, and Simone Potenza (2012). "Aligning economic impact with environmental benefits: A green strategy model". In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. IEEE, pp. 62–68. ISBN: 1-4673-1832-9.

Gupta, Pooja and Priya Pedamkar (2023). *Types of Computer Software | Top 3 Major Types of Computer Software*. URL: https://www.educba.com/types-of-computer-software/ (visited on 08/10/2023).

Gurumurthi, Sudhanva et al. (2002). "Using complete machine simulation for software power estimation: The softwatt approach". In: *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE, pp. 141–150. ISBN: 0-7695-1525-8.

Gwaka, Leon Tinashe (2022). "Computer Supported Livestock Systems: The Potential of Digital Platforms to Revitalize a Livestock System in Rural Zimbabwe". In: *Proceedings of the ACM on Human-Computer Interaction* 6.CSCW2, pp. 1–28. ISSN: 2573-0142.

Haefliger, Stefan, Georg Von Krogh, and Sebastian Spaeth (2008). "Code reuse in open source software". In: *Management science* 54.1, pp. 180–193. ISSN: 0025-1909.

Hähnel, Marcus et al. (2012). "Measuring energy consumption for short code paths using RAPL". In: *ACM SIGMETRICS Performance Evaluation Review* 40.3, pp. 13–17. ISSN: 0163-5999.

Hao, Shuai et al. (2013). "Estimating mobile application energy consumption using program analysis". In: *2013 35th international conference on software engineering (ICSE)*. IEEE, pp. 92–101. ISBN: 1-4673-3076-0.

Harrison, Anthony (2022). "Manage Risk with a Software Bill of Materials". In: *ITNOW* 64.4, pp. 40–41. ISSN: 1746-5702, 1746-5710.

Hartley, Tim et al. (2022). "Just-In-Time Compilation on ARM—A Closer Look at Call-Site Code Consistency". In: *ACM Transactions on Architecture and Code Optimization* 19.4, pp. 1–23. ISSN: 1544-3566, 1544-3973.

Hartmann, F (2011). “Safe template processing of XML documents”. Publication Title: Doctoral Thesis, Technische Universität Dresden Issue: September. PhD thesis. Technischen Universität Dresden.

Hasan, Samir et al. (2016). “Energy profiles of java collections classes”. In: *Proceedings of the 38th International Conference on Software Engineering*, pp. 225–236.

Hasselbring, Wilhelm (2021). “Benchmarking as Empirical Standard in Software Engineering Research”. In: *EASE 2021,*

Heinze, Carolyn; (2014). “Scope Creep: Inevitable, Preventable, Manageable”. In: *Systems Contractor News* 21.12, pp. 38–39.

Hejderup, Joseph, Arie van Deursen, and Georgios Gousios (2018). “Software ecosystem call graph for dependency management”. In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pp. 101–104.

Hertel, Guido, Sven Niedner, and Stefanie Herrmann (2003). “Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel”. In: *Research policy* 32.7, pp. 1159–1177. ISSN: 0048-7333.

HG Insights (2024). *AWS Market Share 2024: Insights into the Buyer Landscape | HG Insights*. URL: `https://hginsights.com/blog/aws-market-report-buyer-landscape` (visited on 02/05/2025).

Hilty, Lorenz M. (2011). *Information technology and sustainability: Essays on the relationship between information technology and sustainable development*. BoD–Books on Demand. ISBN: 3-8423-9655-4.

Hilty, Lorenz M., Wolfgang Lohmann, and Elaine M Huang (2011). “Sustainability and ICT-an overview of the field”. In: *Notizie di POLITEIA* 27.104, pp. 13–28. ISSN: 1128-2401.

Hilty, Lorenz M. et al. (2006). “Rebound effects of progress in information technology”. In: *Poiesis & Praxis* 4.1, pp. 19–38. ISSN: 1615-6609.

Hindle, Abram (2012a). “Green mining: A methodology of relating software change to power consumption”. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, pp. 78–87. ISBN: 1-4673-1761-6.

— (2012b). “Green mining: Investigating power consumption across versions”. In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, pp. 1301–1304. ISBN: 1-4673-1067-0.

— (2016). “Green software engineering: the curse of methodology”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. IEEE, pp. 46–55. ISBN: 1-5090-1855-7.

Hindle, Abram et al. (2014). "Greenminer: A hardware based mining software repositories software energy consumption framework". In: *Proceedings of the 11th working conference on mining software repositories*, pp. 12–21.

Hislop, Gregory W (2009). "Software Engineering Education: Past, Present, and Future". In: *Software Engineering: Effective Teaching and Learning Approaches and Practices*, pp. 1–13.

Hoda, Rashina et al. (2017). "Systematic literature reviews in agile software development: A tertiary study". In: *Information and Software Technology* 85, pp. 60–70.

Hu, Mao-Lin, Mohammad Yaser Masoomi, and Ali Morsali (2019). "Template strategies with MOFs". In: *Coordination Chemistry Reviews* 387, pp. 415–435. ISSN: 00108545.

Hucka, M and M J Graham (2018). "Software search is not a science, even among scientists: A survey of how scientists and engineers find software". In: *The Journal of Systems & Software* 18.

Hunter, Jason and William Crawford (2001). *Java servlet programming: Help for server side Java developers*. O'Reilly. ISBN: 1-4493-9067-6.

Ibrahim, Mostafa E A, Markus Rupp, and Hossam A H Fahmy (2011). "A precise high-level power consumption model for embedded systems software". In: *EURASIP Journal on Embedded Systems* 2011, pp. 1–14.

Intel (2019). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers*. URL: `https://www.intel.com/content/dam/develop/external/us/en/documents/335592-sdm-vol-4.pdf` (visited on 02/11/2024).

— (2022). *Intel® Xeon® CPU Max Series Product Overview*. URL: `https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html` (visited on 09/25/2023).

— (2023). *Hot Chips 2023 - Sierra Forest Xeon Architecture*. URL: `https://www.intel.com/content/www/us/en/content-details/787431/hot-chips-2023-sierra-forest-xeon-architecture-presentation.html` (visited on 09/26/2023).

Jagroep, Erik et al. (2016a). "A Resource Utilization Score for Software Energy Consumption". In: *ICT4S*.

Jagroep, Erik et al. (2017). "Awakening awareness on energy consumption in software engineering". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Society Track (ICSE-SEIS)*. IEEE, pp. 76–85. ISBN: 1-5386-2673-X.

Jagroep, Erik A. et al. (2016b). "Software energy profiling: Comparing releases of a software product". In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp. 523–532. ISBN: 978-1-4503-4161-5.

Jain, Anil K, Yu Zhong, and Marie-Pierre Dubuisson-Jolly (1998). "Deformable template models: A review". In: *Signal Processing* 71.2, pp. 109–129. ISSN: 01651684.

Java Design Patterns (2023). *Fluent Interface*. URL: https://java-design-patterns.com/patterns/fluentinterface/ (visited on 12/11/2023).

Jelschen, Jan et al. (2012). "Towards applying reengineering services to energy-efficient applications". In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, pp. 353–358. ISBN: 0-7695-4666-8.

Jensen, R. J. and G. Szulanski (2007). "Template Use and the Effectiveness of Knowledge Transfer". In: *Management Science* 53.11. ISBN: 0025-1909, pp. 1716–1730. ISSN: 0025-1909.

Jiang, James et al. (2007a). "Lack of Skill Risks to Organizational Technology Learning and Software Project Performance". In: *, Information Resources Management Journal,* 20.3.

Jiang, Xiaofan et al. (2007b). "Micro power meter for energy monitoring of wireless sensor networks at scale". In: *Proceedings of the 6th international conference on Information processing in sensor networks*, pp. 186–195.

Jin, Chaoqiang et al. (Dec. 2022). "A measurement-based power consumption model of a server by considering inlet air temperature". In: *Energy* 261, p. 125126. ISSN: 03605442.

Joseph, Russ, David Brooks, and Margaret Martonosi (2001). "Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs". In: *Workshop on Complexity Effectice Design WCED, held in conjunction with ISCA*. Vol. 28.

Kalaitzoglou, Georgios, Magiel Bruntink, and Joost Visser (2014). "A Practical Model for Evaluating the Energy Efficiency of Software Applications". In: *ICT4S*, pp. 77–86.

Kang, Ruogu et al. (July 2015). ""My Data Just Goes Everywhere:" User Mental Models of the Internet and Implications for Privacy and Security". In: *2015 Symposium on Usable Privacy and Security*. Ottawa, Canada.

Kansal, Aman and Feng Zhao (2008). "Fine-grained energy profiling for power-aware application design". In: *ACM SIGMETRICS Performance Evaluation Review* 36.2, pp. 26–31. ISSN: 0163-5999.

Kaup, Fabian, Philip Gottschling, and David Hausheer (2014). "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi". In: *39th Annual IEEE Conference on Local Computer Networks*. ISBN: 978-1-4799-3780-6, pp. 236–243.

Kaup, Fabian et al. (2018). "The Progress of the Energy-efficiency of Single-board Computers". In: *Tech. Rep. NetSys-TR-2018-01*.

Kazman, Rick et al. (2018). "Managing Energy Consumption as an Architectural Quality Attribute". In: *IEEE Software* 35.5, pp. 102–107. ISSN: 0740-7459.

Kern, Eva et al. (2013). "Green Software and Green Software Engineering - Definitions, Measurements and Quality Aspects". In: *Proceedings of the First International Conference on Information and Communication Technologies for Sustainability (ICT4S)*, pp. 87–94.

Khomh, Foutse and S Amirhossein Abtahizadeh (2018). "Understanding the impact of cloud patterns on performance and energy consumption". In: *Journal of Systems and Software* 141, pp. 151–170. ISSN: 0164-1212.

Khoshgoftaar, Taghi M (2001). "Empirical Assessment of a Software Metric: The Information Content of Operators". In: *Software Quality Journal* 9, pp. 99–112.

Kim, Yongbeom and Edward A Stohr (1998). "Software reuse: Survey and research directions". In: *Journal of Management Information Systems* 14.4, pp. 113–147. ISSN: 0742-1222.

King, Nigel (1998). "Template analysis." In: *Qualitative methods and analysis in organizational research: A practical guide*. Thousand Oaks, CA: Sage Publications Ltd, pp. 118–134. ISBN: 0-7619-5350-7.

Kitchenham, Barbara and Stuart Charters (2007). *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Keele University.

Knowles, Bran et al. (June 2022). "Our house is on fire: The climate emergency and computing's responsibility." In: *Communications of the ACM* 65.6, pp. 38–40. ISSN: 0001-0782, 1557-7317.

Kolesnikov, A (2006). "Enterprise application development using Jakarta Tapestry". PhD thesis. Glasgow Caledonian University.

Koomey, Jonathan G et al. (2009). "Assessing trends over time in performance, costs, and energy use for servers". In: *Lawrence Berkeley National Laboratory, Stanford University, Microsoft Corporation, and Intel Corporation, Tech. Rep.*

Korolov, Maria (2022). *Arm chips gaining in data centers, but still in single digits*. URL: `https://www.datacenterknowledge.com/arm/arm-chips-gaining-data-centers-still-single-digits` (visited on 09/25/2023).

Koskela, Lasse (2007). *Test Driven - Practical TDD and Acceptance TDD for Java Developers*. Manning. ISBN: 81-7722-779-3.

Küber, Esteban (Oct. 2023). *Efficient Language and Library Use to Reduce Carbon*. URL: `https://www.infoq.com/presentations/rust-java-app/` (visited on 10/12/2023).

Kunder, Maurice De (2008). "Geschatte grootte van het geïndexeerde World Wide Web". PhD thesis. Tilburg University.

Kusuma, Mandahadi, Widyawan, and Ridi Ferdiana (July 2017). "Performance comparison of caching strategy on wordpress multisite". In: *2017 3rd International Conference on Science and Technology - Computer (ICST)*. Yogyakarta, Indonesia: IEEE, pp. 176–181. ISBN: 978-1-5386-1874-5.

Laakso, Tuukka and Joni Niemi (2008). "An Evaluation of AJAX-enabled Java-based Web Application Frameworks". In: *Proceedings of MoMM2008*, pp. 431–437.

Lago, Patricia (2019). "Architecture design decision maps for software sustainability". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. IEEE, pp. 61–64. ISBN: 1-7281-1762-3.

Lakhani, Karim R and Robert G Wolf (2003). "Why hackers do what they do: Understanding motivation and effort in free/open source software projects". In: *Open Source Software Projects (September 2003)*.

Larios Vargas, Enrique et al. (2020). "Selecting Third-Party Libraries: The Practitioners' Perspective". In: *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 245–256.

LaToza, Thomas D, Evelina Shabani, and André van der Hoek (2013). "A study of architectural decision practices". In: *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, pp. 77–80. ISBN: 1-4673-6290-5.

Law, Marcus (2022). *Energy efficiency predictions for data centres in 2023*. URL: `https://datacentremagazine.com/articles/efficiency-to-loom-large-for-data-centre-industry-in-2023` (visited on 07/26/2023).

Lawlis, Patricia K et al. (2001). "A formal process for evaluating COTS software products". In: *Computer* 34.05, pp. 58–63. ISSN: 0018-9162.

Lechelt, Susan, Katerina Gorkovenko, and Chris Speed (2024). "On Disused Connected Devices: Understanding Disuse, 'Holding On' and Barriers to Circularity". In: *ACM Journal on Computing and Sustainable Societies*, p. 3651171. ISSN: 2834-5533.

Lee, Sung Une et al. (Jan. 2024). "A survey of energy concerns for software engineering". In: *Journal of Systems and Software*, p. 111944. ISSN: 01641212.

Leiner, Barry M. et al. (1997). *Brief History of the Internet*.

Lerner, Josh and Jean Tirole (2002). "Some simple economics of open source". In: *The journal of industrial economics* 50.2, pp. 197–234. ISSN: 0022-1821.

Lesaint, D et al. (2003). "Field workforce scheduling". In: *BT Technology Journal* 21.4, p. 4.

Li, Ding et al. (2014). "An empirical study of the energy consumption of android applications". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, pp. 121–130. ISBN: 1-4799-6146-9.

Lilja, David J (2000). *Measuring computer performance: a practitioner's guide*. Cambridge university press. ISBN: 0-521-64670-7.

Lim, Wayne C (1994). "Effects of reuse on quality, productivity, and economics". In: *IEEE software* 11.5, pp. 23–30. ISSN: 0740-7459.

Linders, Ben (Jan. 2023). *Sustainability for Software Companies: Reducing Impact by Deciding What Not to Do*. URL: `https://www.infoq.com/news/2023/01/sustainability-software-impact/` (visited on 01/05/2023).

Litke, Andreas et al. (2005). "Energy consumption analysis of design patterns". In: *Proceedings of the International Conference on Machine Learning and Software Engineering*, pp. 86–90.

Live Counter (2019). *How Big Is The Internet? (In Petabyte)*. URL: `https://live-counter.com/how-big-is-the-internet/` (visited on 04/13/2019).

Mäkinen, Joona (2022). "Creating a Monetization Model for a Free-to-play Mobile Game". PhD thesis.

Mancebo, Javier, Coral Calero, and Félix García (2021). "Does maintainability relate to the energy consumption of software? A case study". In: *Software Quality Journal*. ISSN: 0963-9314.

Mann, Dr Samuel and Lesley Smith (2007). "Computing Education for Sustainability". In: Nelson, New Zealand.

Manner, Jukka (2023). "Black software — the energy unsustainability of software systems in the 21st century". In: *Oxford Open Energy* 2. ISSN: 2752-5082.

Manotas, Irene, Lori Pollock, and James Clause (2014). "Seeds: A software engineer's energy-optimization decision support framework". In: *Proceedings of the 36th International Conference on Software Engineering*, pp. 503–514.

Manotas, Irene et al. (2016). "An empirical study of practitioners' perspectives on green software engineering". In: *Proceedings - International Conference on Software Engineering*. Vol. 14-22-May-. IEEE Computer Society, pp. 237–248. ISBN: 978-1-4503-3900-1.

Marcu, Marius and Dacian Tudor (2011). "Power consumption measurements of virtual machines". In: *2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, pp. 445–449. ISBN: 1-4244-9109-6.

Maslow, Abraham H. (1966). *The psychology of science*. Harper & Row.

Matthews, L., T. Ishikawa, and S. Baker (2004). "The template update problem". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.6, pp. 810–815. ISSN: 0162-8828.

MaxLinear (2018). *MXL7704 Datasheet*.

Maxville, Valerie, J Armarego, and CP Lam (2004). "Learning to select software components". In: *2004 International Conference of Software Engineering and Knowledge Engineering*. Knowledge Systems Institute.

McCandless, David, Pearl Doughty-White, and Miriam Quick (2015). *Information is Beautiful*. URL: https://informationisbeautiful.net/visualizations/million-lines-of-code/ (visited on 01/24/2025).

Mengesha, Temesgen Hagos (2024). "Software Sustainability Consideration in Industry". PhD thesis. Uppsala Universitet.

Meyer, Mathias (2014). "Continuous Integration and Its Tools". In: *IEEE Software* 31.3, pp. 14–16. ISSN: 0740-7459, 1937-4194.

Midha, Vishal and Prashant Palvia (2011). "Factors affecting the success of Open Source Software". In: *The Journal of Systems & Software* 85, pp. 895–905.

Mileva, Yana Momchilova, Valentin Dallmeier, and Andreas Zeller (2010). "Mining API popularity". In: *Testing–Practice and Research Techniques: 5th International Academic and Industrial Conference, TAIC PART 2010, Windsor, UK, September 3-5, 2010. Proceedings*. Springer, pp. 173–180. ISBN: 3-642-15584-7.

Mileva, Yana Momchilova et al. (2009). "Mining trends of library usage". In: *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pp. 57–62.

Milkman, Katherine L, Dolly Chugh, and Max H Bazerman (2009). "How can decision making be improved?" In: *Perspectives on psychological science* 4.4. Publisher: SAGE Publications Sage CA: Los Angeles, CA, pp. 379–383. ISSN: 1745-6916.

Mills, John A. (1985). "A pragmatic view of the system architect". In: *Communications of the ACM* 28.7, pp. 708–717. ISSN: 0001-0782, 1557-7317.

Milosevic, Mladen et al. (2013). "An environment for automated power measurements on mobile computing platforms". In: *Proceedings of the 51st ACM Southeast Conference*, pp. 1–6.

Mingay, Simon (2007). "Green IT: the new industry shock wave". In: *Gartner RAS Research Note G* 153703.7.

Mittapalli, Jishnu Saurav and Menaka Pushpa Arthur (2021). "Survey on Template Engines in Java". en. In: *ITM Web of Conferences* 37. Ed. by J. Kannan R. et al., p. 01007. ISSN: 2271-2097. DOI: 10.1051/itmconf/20213701007. URL: https://www.itm-conferences.org/10.1051/itmconf/20213701007 (visited on 02/28/2026).

Mockus, Audris (2007). "Large-scale code reuse in open source software". In: *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, pp. 7–7. ISBN: 0-7695-2961-5.

Moore, Julia E. et al. (2017). "Developing a comprehensive definition of sustainability". In: *Implementation Science* 12.1, p. 110. ISSN: 1748-5908.

Moreira, Jaziel S, Everton LG Alves, and Wilkerson L Andrade (2020). "A Systematic Mapping on Energy Efficiency Testing in Android Applications". In: *Proceedings of the XIX Brazilian Symposium on Software Quality*, pp. 1–10.

Morley, Janine, Kelly Widdicks, and Mike Hazas (2018). "Digitalisation, energy and data demand: The impact of Internet traffic on overall and peak electricity consumption". In: *Energy Research & Social Science* 38, pp. 128–137. ISSN: 22146296.

Murray, Brian H and Alvin Moore (2000). *Sizing the Internet*. Tech. rep. Cyveillance.

Mythic Beasts (2023). *Raspberry Pi - Mythic Beasts*. URL: `https://www.mythic-beasts.com/order/rpi` (visited on 09/25/2023).

Naderiparizi, Saman et al. (2016). "uMonitor: In-situ energy monitoring with microwatt power consumption". In: *2016 IEEE International Conference on RFID (RFID)*. IEEE, pp. 1–8. ISBN: 1-4673-8807-6.

NASA (2016). *'Computer' Conducts Data Analysis - NASA*. URL: `https://www.nasa.gov/image-article/computer-conducts-data-analysis/` (visited on 10/13/2023).

Naumann, Stefan (2008). "Sustainability Informatics-A new Subfield of Applied Informatics?" In: *EnviroInfo*, pp. 384–389.

Naumann, Stefan et al. (2011). "The GREENSOFT Model: A reference model for green and sustainable software and its engineering". In: *Sustainable Computing: Informatics and Systems* 1.4, pp. 294–304. ISSN: 2210-5379.

Naumann, Stefan et al. (2015). "Sustainable software engineering: Process and quality models, life cycle, and social aspects". In: *ICT Innovations for Sustainability*. Springer, pp. 191–205. ISBN: 3-319-09227-8.

Nazir, Shah et al. (2014). "Software component selection based on quality criteria using the analytic network process". In: *Abstract and Applied Analysis* 2014, pp. 1–12. ISSN: 16870409 10853375.

Nelson, Ted (1974). *Deeam Machines / Computer Lib*. Southe Bend, Ind.: The Distributors.

Netcraft (June 2023). *June 2023 Web Server Survey*. URL: `https://www.netcraft.com/blog/june-2023-web-server-survey/` (visited on 07/24/2023).

Netflix (2012). *Netflix Shares Cloud Load Balancing And Failover Tool*. URL: https://netflixtechblog.com/netflix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5?gi=f25b99d27a0a (visited on 09/22/2023).

Newport, Cal (2016). *Deep Work: Rules for Focused Success in a Distracted World*. Piatkus. ISBN: 978-0-349-41190-3.

Nguyen, Phuong T et al. (2020). "CrossRec: Supporting software developers by recommending third-party libraries". In: *Journal of Systems and Software* 161, p. 110460. ISSN: 0164-1212.

Nielsen, Henrik et al. (1999). *RFC 2616 Hypertext transfer protocol – HTTP/1.1*.

Noman, Hira et al. (July 2022). "An Exploratory Study of Software Sustainability at Early Stages of Software Development". In: *Sustainability* 14.14, p. 8596. ISSN: 2071-1050.

Noureddine, Adel and Ajitha Rajan (2015). "Optimising energy consumption of design patterns". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE, pp. 623–626. ISBN: 1-4799-1934-9.

Noureddine, Adel et al. (2012). "Runtime monitoring of software energy hotspots". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 160–169.

O'Callaghan, John (2017). "Hardware Acceleration of an Open Source Web Server". PhD thesis. Leuven, Belgium: KU Leuven.

Odlyzko, Andrew (2016). "The Growth Rate and the Nature of Internet Traffic". In: *IPSI BGD TRANSACTIONS ON ADVANCED RESEARCH* 12.1.

OECD (2003). *Glossary of Statistical terms: Transaction Costs*. URL: https://stats.oecd.org/glossary/detail.asp?ID=3324 (visited on 11/16/2022).

Open Source Initiative (2024a). *The Open Source AI Definition*. URL: https://opensource.org/ai (visited on 02/08/2025).

— (2024b). *The Open Source Definition*. URL: https://opensource.org/osd (visited on 02/08/2025).

Openhub (2023a). *The Apache HTTP Server Open Source Project on Open Hub*. URL: https://openhub.net/p/apache (visited on 07/26/2023).

— (2023b). *The nginx Open Source Project on Open Hub*. URL: https://openhub.net/p/nginx (visited on 07/26/2023).

Oracle (2018a). *Go Java. About Java*. URL: https://go.java/index.html.

— (2018b). *Oracle Java Tutorials: JavaBeans*. URL: https://docs.oracle.com/javase/tutorial/javabeans/.

— (2021). *Conditional Operator ? :* The Java Language Specification. URL: https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html (visited on 11/27/2023).

Owens, Susan (Apr. 2003). “Is there a meaningful definition of sustainability?” In: *Plant Genetic Resources* 1.1, pp. 5–9. ISSN: 1479-2621, 1479-263X.

Palomba, Fabio et al. (2019). “On the impact of code smells on the energy consumption of mobile applications”. In: *Information and Software Technology* 105, pp. 43–55. ISSN: 0950-5849.

Pang, Candy et al. (2016). “What Do Programmers Know about Software Energy Consumption?” In: *IEEE Software* 33.3, pp. 83–89. ISSN: 07407459.

Panigrahi, Debashis et al. (2001). “Battery life estimation of mobile embedded systems”. In: *VLSI Design 2001. Fourteenth International Conference on VLSI Design*. Bangalore, India: IEEE, pp. 57–63. ISBN: 978-0-7695-0831-3.

Paradis, Carlos, Rick Kazman, and Damian Andrew Tamburri (2021). “Architectural tactics for energy efficiency: Review of the literature and research roadmap”. In: *Proceedings of the 54th Hawaii International Conference on System Sciences | 2021*.

Parr, Terence John (2004). “Enforcing strict model-view separation in template engines”. In: *Proceedings of the 13th conference on World Wide Web - WWW '04*. ISBN: 158113844X, p. 224.

Paschali, Maria-Eleni et al. (2017). “Reusability of open source software across domains: A case study”. In: *The Journal of Systems and Software* 134, pp. 211–227. ISSN: 0164-1212.

Patel, Savan K, V.R. Rathod, and Satyen Parikh (2011). “Joomla, Drupal and WordPress - a statistical comparison of open source CMS”. In: *3rd International Conference on Trendz in Information Sciences & Computing (TISC2011)*. Chennai, India: IEEE, pp. 182–187.

Peitek, Norman et al. (2021). “Program Comprehension and Code Complexity Metrics: An fMRI Study”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, pp. 524–536. ISBN: 978-1-6654-0296-5.

Penzenstadler, Birgit (2012). *Sustainability in Software Engineering: A Systematic Literature Review for Building up a Knowledge Base*. Tech. rep. Technische Universität München.

— (2013). “Towards a Definition of Sustainability in and for Software Engineering”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal: ACM, pp. 1183–1185. ISBN: 978-1-4503-1656-9.

Penzenstadler, Birgit, Henning Femmer, and Debra Richardson (2013). “Who is the advocate? Stakeholders for sustainability”. In: *2013 2nd International Workshop on Green and Sustainable Software, GREENS 2013 - Proceedings*. IEEE, pp. 70–77. ISBN: 978-1-4673-6267-2.

Penzenstadler, Birgit et al. (2014a). “Safety, security, now sustainability: The nonfunctional requirement for the 21st century”. In: *IEEE software* 31.3, pp. 40–47. ISSN: 0740-7459.

Penzenstadler, Birgit et al. (2014b). “Systematic mapping study on software engineering for sustainability (SE4S)”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. London, England, pp. 1–14.

Pereira, Rui et al. (2016). “The influence of the Java collection framework on overall energy consumption”. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp. 15–21. ISBN: 978-1-4503-4161-5.

Pereira, Rui et al. (2017). “Energy Efficiency across Programming Languages”. In: *International Conference on Software Language Engineering*. ISBN: 9781450355254, pp. 256–257.

Pereira, Rui et al. (2020). “SPELLing out energy leaks: Aiding developers locate energy inefficient code”. In: *Journal of Systems and Software* 161, p. 110463. ISSN: 01641212.

Pereira, Rui et al. (2021). “Ranking programming languages by energy efficiency”. In: *Science of Computer Programming* 205. ISSN: 01676423.

Petro, G (2017). *The Need For Speed: Time-To-Market Acceleration Is The Ultimate Retail Differentiator*. URL: `https://www.forbes.com/sites/gregpetro/2017/10/06/the-need-for-speed-time-to-market-acceleration-is-the-ultimate-retail-differentiator/1` (visited on 07/31/2018).

Pierrehumbert, Raymond (2019). “There is no Plan B for dealing with the climate crisis”. In: *Bulletin of the Atomic Scientists* 75.5, pp. 215–221. ISSN: 0096-3402, 1938-3282.

Pigott, Diarmuid (2020). *Online Historical Encyclopaedia of Programming Languages*. URL: `https://hopl.info/` (visited on 08/07/2023).

Pinto, Gustavo and Fernando Castor (2017). “Energy efficiency: A new concern for application software developers”. In: *Communications of the ACM* 60.12, pp. 189–190.

Pinto, Gustavo, Fernando Castor, and Yu David Liu (2014). “Mining Questions about Software Energy Consumption”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 22–31. ISBN: 978-1-4503-2863-0.

Pinto, Gustavo et al. (2016). “A comprehensive study on the energy efficiency of java’s thread-safe collections”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 20–31. ISBN: 1-5090-3806-X.

Pisu, Lorenzo, Davide Maiorca, and Giorgio Giacinto (May 2024). *A Survey of the Overlooked Dangers of Template Engines*. en. arXiv:2405.01118 [cs]. DOI: 10.48550/arXiv.2405.01118. URL: http://arxiv.org/abs/2405.01118 (visited on 02/28/2026).

Povoa, Lucas Venezian et al. (2013). "A model for estimating energy consumption based on resources utilization". In: *2013 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, pp. 1–6. ISBN: 1-4799-3755-X.

Rahman, Akond, Rezvan Mahdavi-Hezaveh, and Laurie Williams (2019). "A systematic mapping study of infrastructure as code research". In: *Information and Software Technology* 108, pp. 65–77. ISSN: 09505849.

Raja, Uzma and Marietta J. Tretter (2012). "Defining and evaluating a measure of Open Source Project survivability". In: *IEEE Transactions on Software Engineering* 38.1. ISBN: 9781467372848, pp. 163–174. ISSN: 00985589.

Rani, Lekshmi Murali et al. (Jan. 2025). *An Empirical Study on Decision-Making Aspects in Responsible Software Engineering for AI*. en. arXiv:2501.15691 [cs]. DOI: 10.48550/arXiv.2501.15691. URL: http://arxiv.org/abs/2501.15691 (visited on 02/08/2025).

Raspberry Hosting (2023). *Raspberry Pi hosting | Raspberry Pi Colocation*. URL: https://raspberry-hosting.com/en (visited on 09/25/2023).

Raspberry Pi Foundation (2023a). *Raspberry Pi Documentation - Raspberry Pi hardware*. URL: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html (visited on 09/27/2023).

— (2023b). *Raspberry Pi*. URL: https://www.raspberrypi.com/ (visited on 09/22/2023).

— (2023c). *The 3 types of computer software*. URL: https://www.futurelearn.com/info/courses/computer-systems/0/steps/53500 (visited on 08/10/2023).

Raymond, Eric (1999). "The cathedral and the bazaar". In: *Knowledge, Technology & Policy* 12, pp. 23–49.

Raymond, Eric Steven (2010). *The cathedral and the bazaar*. Snowball Publishing.

RedHat (2022). *How to write greener Java applications*. URL: https://www.redhat.com/en/resources/greener-java-applications-detail (visited on 07/30/2023).

Ren, Xiaoxia et al. (2004). "Chianti: a tool for change impact analysis of java programs". In: *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 432–448.

Rice, Andrew and Simon Hay (2010). "Measuring mobile phone energy consumption for 802.11 wireless networking". In: *Pervasive and Mobile Computing* 6.6, pp. 593–606. ISSN: 1574-1192.

Rodgers, Paul A., Avon P. Huxor, and Nicholas H M Caldwell (1999). "Design support using distributed web-based AI tools". In: *Research in Engineering Design - Theory, Applications, and Concurrent Engineering* 11.1, pp. 31–44. ISSN: 09349839.

Rudra, Sourav (2024). *OSI Calls Out Meta for its Misleading 'Open Source' AI Models*. URL: `https://news.itsfoss.com/osi-meta-ai/` (visited on 02/08/2025).

Sabovic, Adnan et al. (2020). "Accurate online energy consumption estimation of IoT devices using energest". In: *Advances on Broad-Band Wireless Computing, Communication and Applications: Proceedings of the 14th International Conference on Broad-Band Wireless Computing, Communication and Applications (BWCCA-2019) 14*. Springer, pp. 363–373. ISBN: 3-030-33505-4.

Sahin, Cagri, Lori Pollock, and James Clause (2014). "How do code refactorings affect energy usage?" In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10.

Sahin, Cagri et al. (2012). "Initial explorations on design pattern energy usage". In: *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*. IEEE, pp. 55–61. ISBN: 978-1-4673-1832-7.

Salam, Muhammad and Siffat Ullah Khan (2018). "Challenges in the development of green and sustainable software for software multisourcing vendors: Findings from a systematic literature review and industrial survey". In: *Journal of Software: Evolution and Process* 30.8, e1939. ISSN: 2047-7473.

Sametinger, Johannes (1997). *Software Engineering with Reusable Components*. Berlin, Heidelberg: Springer. ISBN: 978-3-642-08299-3.

Saputri, Theresia Ratih Dewi and Seok-Won Lee (2016). "Incorporating sustainability design in requirements engineering process: A preliminary study". In: *Requirements Engineering Toward Sustainable World: Third Asia-Pacific Symposium, APRES 2016, Nagoya, Japan, November 10-12, 2016, Proceedings 3*. Springer, pp. 53–67. ISBN: 981-10-3255-6.

Sawyer, Steve (2004). "Software development teams". In: *Communications of the ACM* 47.12, pp. 95–99. ISSN: 0001-0782.

Schafer, Ronald W (2011). "What is a Savitzky-Golay filter?[lecture notes]". In: *IEEE Signal processing magazine* 28.4, pp. 111–117. ISSN: 1053-5888.

Schuler, Andreas and Gabriele Kotsis (2023). "A systematic review on techniques and approaches to estimate mobile software energy consumption". In: *Sustainable Computing: Informatics and Systems*, p. 100919. ISSN: 2210-5379.

Schwaber, Ken (1997). "Scrum development process". In: *Business Object Design and Implementation: OOPSLA'95 Workshop Proceedings 16 October 1995, Austin, Texas*. Springer, pp. 117–134. ISBN: 3-540-76096-2.

SellerApp (2023). *Amazon Statistics (Seller, FBA, and Product) That'll Surprise You*. URL: `https://www.sellerapp.com/blog/amazon-seller-statistics/` (visited on 07/24/2023).

Seo, Chiyoung, Sam Malek, and Nenad Medvidovic (2008). "Estimating the energy consumption in pervasive java-based systems". In: *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, pp. 243–247. ISBN: 0-7695-3113-X.

Shahin, Mojtaba, Muhammad Ali Babar, and Liming Zhu (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5, pp. 3909–3943. ISSN: 2169-3536.

Shehabi, Arman et al. (2016). *United States Data Center Energy Usage Report*. Tech. rep. LBNL–1005775, 1372902. Lawrence Berkeley National Laboratory, LBNL–1005775, 1372902.

Silva, WG da et al. (2010). "Evaluation of the impact of code refactoring on embedded software efficiency". In: *Proceedings of the 1st Workshop de Sistemas Embarcados*, pp. 145–150.

Similarweb (2023). *Top Websites Ranking - Most Visited Websites In The World*. URL: `https://www.similarweb.com/top-websites/` (visited on 08/01/2023).

Singh, Sanjay Kumar and Amarjeet Singh (2012). *Software testing*. Vandana Publications. ISBN: 81-941110-6-4.

Sinha, Amit and Anantha P Chandrakasan (2001). "Jouletrack: A web based tool for software energy profiling". In: *Proceedings of the 38th annual Design Automation Conference*, pp. 220–225.

Snowdon, David C, Stefan M Petters, and Gernot Heiser (2005). "Power measurement as the basis for power management". In: *Proceedings of the 2005 Workshop on Operating System Platforms for Embedded Real-Time Applications*. Palma, Mallorca, Spain.

Sojer, Manuel and Joachim Henkel (2010). "Code reuse in open source software development: Quantitative evidence, drivers, and impediments". In: *Journal of the Association for Information Systems* 11.12, pp. 868–901.

Sonatype (2023). *9th Annual State of the Software Supply Chain Report*. Tech. rep. Sonatype.

Souza, Ana Carolina Moises de (Jan. 2023). "Social Sustainability Approaches for a Sustainable Software Product". In: *ACM SIGSOFT Software Engineering Notes* 48.1, pp. 38–43. ISSN: 0163-5948.

Souza, Mario Rosado de, Robert Haines, and Caroline Jay (2014). "Defining sustainability through developers' eyes: Recommendations from an interview study". In: *Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE 2014)*.

Sowe, Sulayman K, Ioannis Stamelos, and Lefteris Angelis (2008). "Understanding knowledge sharing activities in free/open source software projects: An empirical study". In: *The Journal of Systerms and Software* 81, pp. 431–446.

SPEC (2008). *SPECpower_ssj® 2008*. URL: `https://www.spec.org/power_ssj2008/` (visited on 01/30/2024).

Spinellis, D. (2006). "Choosing a programming language". In: *IEEE Software* 23.4, pp. 62–63. ISSN: 0740-7459.

Spinellis, Diomidis (2019). "How to Select Open Source Components". In: *Cpmputer* 52.12, pp. 103–106.

Stalnaker, Trevor Wayne (2023). "A Comprehensive Study of Bills of Materials for Software Systems". PhD thesis. The College of William and Mary.

Stathopoulos, Thanos, Dustin McIntire, and William J Kaiser (2008). "The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes". In: *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*. IEEE, pp. 383–394. ISBN: 0-7695-3157-1.

Statista (2023). *Internet browser market share 2012-2023 | Statista*. URL: `https://www.statista.com/statistics/268254/market-share-of-internet-browsers-worldwide-since-2009/` (visited on 10/03/2023).

Steffen, Will et al. (2015). "Planetary boundaries: Guiding human development on a changing planet". In: *Science* 347.6223, p. 1259855. ISSN: 0036-8075, 1095-9203.

Stier, Christian et al. (2015). "Model-based energy efficiency analysis of software architectures". In: *ECSA 2015 Proceedings 9*. Dubrovnik/Cavtat, Croatia, Springer, pp. 221–238. ISBN: 3-319-23726-8.

Stoico, Vincenzo et al. (2023). "An Approach Using Performance Models for Supporting Energy Analysis of Software Systems". In: *European Workshop on Performance Engineering*. Springer, pp. 249–263.

Strattic (2023). *Pricing | Strattic*. URL: `https://www.strattic.com/pricing/` (visited on 10/06/2023).

Strawn, George (2014). "Masterminds of the World Wide Web". In: *IT Professional* 16.4, pp. 58–59. ISSN: 1520-9202, 1941-045X.

Taplin, Jonathan (2018). *Move Fast and Break Things: How Facebook, Google and Amazon Have Cornered Culture and Undermined Democracy*. Macmillan. ISBN: 978-1-5098-4770-9.

Teixeira, Leonor et al. (2015). "Selecting an Open-Source Framework: A practical case based on software development for sensory analysis". In: *Procedia - Procedia Computer Science* 64.64, pp. 1057–1064. ISSN: 1877-0509.

Tekie.com (2023). *SSD vs HDD - Comparing Speed, Lifespan, Reliability*. URL: https://tekie.com/blog/hardware/ssd-vs-hdd-speed-lifespan-and-reliability/ (visited on 09/25/2023).

Texas Instruments (2015). *INA 219 Data sheet*.

— (2016). *INA 260 Datasheet*. URL: https://www.ti.com/lit/ds/symlink/ina260.pdf (visited on 09/22/2023).

Tilson, David, Kalle Lyytinen, and Carsten Sørensen (2010). "Research commentary—Digital infrastructures: The missing IS research agenda". In: *Information systems research* 21.4, pp. 748–759. ISSN: 1047-7047.

TIOBE (2018). *TIOBE Index for June 2018*. URL: https://www.tiobe.com/tiobe-index/ (visited on 07/15/2018).

— (2024). *TIOBE Index for December 2024*. URL: https://www.tiobe.com/tiobe-index/ (visited on 12/20/2024).

— (2025). *TIOBE Index - TIOBE*. URL: https://www.tiobe.com/tiobe-index/programminglanguages_definition/ (visited on 02/03/2025).

Toczé, Klervie et al. (2022). "The Dark Side of Cloud and Edge Computing: An Exploratory Study". In: *8th Workshop on Computing within Limits (LIMITS 2022)*.

Torchiano, Marco and Maurizio Morisio (2004). "Overlooked aspects of COTS-based development". In: *IEEE software* 21.2, pp. 88–93. ISSN: 0740-7459.

Travers, Matthew (2015). *CPU Power Consumption Experiments and Results Analysis of Intel i7-4820K*. Tech. rep. Newcastle University.

Trimble (2006). *AtRoad - Leading Mobile Resource Management Solutions*. URL: https://www.road.com/solutions/fsm_taskf_intro.html (visited on 11/15/2022).

Trisovic, Ana et al. (2022). "A large-scale study on research code quality and execution". In: *Scientific Data* 9.1, p. 60. ISSN: 2052-4463.

United Nations (2015). *THE 17 GOALS | Sustainable Development*. URL: https://sdgs.un.org/goals (visited on 11/12/2022).

Unknown (2023). *Wordwise*. URL: https://en.wikipedia.org/wiki/Wordwise (visited on 08/10/2023).

Usman, Muhammad et al. (2017). "Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method". In: *Information and Software Technology* 85, pp. 43–59.

Varghese, Benoy et al. (2015). "Greening web servers: A case for ultra low-power web servers". In: *2014 International Green Computing Conference, IGCC*

2014. Institute of Electrical and Electronics Engineers Inc. ISBN: 978-1-4799-6177-1.

Various (2023). *What are the different types of software, such as system software, application software, and utility software?* URL: `https://www.quora.com/What-are-the-different-types-of-software-such-as-system-software-application-software-and-utility-software` (visited on 10/05/2023).

Vasilescu, Bogdan, Alexander Serebrenik, and Mark G J Van Den Brand (2013). "The Babel of software development: Linguistic diversity in open source". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8238 LNCS, pp. 391–404.

Venters, Colin C et al. (2017a). "Characterising sustainability requirements: A new species red herring or just an odd fish?" In: IEEE, pp. 3–12. ISBN: 1-5386-2673-X.

Venters, Colin C. et al. (2014). "Software Sustainability: The Modern Tower of Babel". In: *RE4SuSy: Third International Workshop on Requirements Engineering for Sustainable Systems*.

Venters, Colin C. et al. (2017b). "Software sustainability: Research and practice from a software architecture viewpoint". In: *The Journal of Systems & Software* 138, pp. 174–188.

Venters, Colin C. et al. (2021). "Software Sustainability: Beyond the Tower of Babel". In: *2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS)*, pp. 11–12. ISBN: 978-1-6654-4460-6.

Venters, Colin C. et al. (2023). "Sustainable software engineering: Reflections on advances in research and practice". In: *Information and Software Technology*, p. 107316. ISSN: 09505849.

Vercauteren, Tom et al. (2007). "Hierarchical forecasting of web server workload using sequential Monte Carlo training". In: *2006 IEEE Conference on Information Sciences and Systems, CISS 2006 - Proceedings* 55.4. ISBN: 1424403502, pp. 899–904. ISSN: 1053587X.

Verdecchia, Roberto, Patricia Lago, and Carol de Vries (2022). "The future of sustainable digital infrastructures: A landscape of solutions, adoption factors, impediments, open problems, and scenarios". In: *Sustainable Computing: Informatics and Systems*, p. 100767. ISSN: 22105379.

Vetro, Antonio et al. (2013). "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system". In: *Proceedings of ENERGY 2013: The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pp. 34–39. ISBN: 1-61208-259-9.

Vollebregt, Tobi, Lennart C L Kats, and Eelco Visser (2012). "Declarative Specification of Template-Based Textual Editors". In: *LDTA 12*. (Visited on 04/02/2017).

Vom Brocke, Jan, Alan Hevner, and Alexander Maedche (2020). "Introduction to design science research". In: *Design science research. Cases*, pp. 1–13. ISSN: 3030467805.

Vosloo, Iwan and Derrick G. Kourie (2008). "Server-centric Web frameworks". In: *ACM Computing Surveys* 40.2, pp. 1–33. ISSN: 03600300.

W3Techs (2022). *Usage Statistics and Market Share of WordPress, June 2022*. URL: `https://w3techs.com/technologies/details/cm-wordpress` (visited on 06/24/2022).

Wahler, Michael (2024). "Exploring Assessment Criteria for Sustainable Software Engineering Processes". In: *ICSE-SEIS'24*. Lisbon, Portugal.

Wang, Peng et al. (Nov. 2022). "Trends in energy consumption under the multi-stage development of ICT: Evidence in China from 2001 to 2030". In: *Energy Reports* 8, pp. 8981–8995. ISSN: 23524847.

Whittaker, J.A. (Feb. 2000). "What is software testing? And why is it so hard?" In: *IEEE Software* 17.1, pp. 70–79. ISSN: 07407459.

Widdicks, Kelly Victoria et al. (2018). *Undesigning the Internet: An exploratory study of reducing everyday Internet connectivity*.

Wiggers, Steef-Jan (2023). *Why Cloud Zombies Are Destroying the Planet and How You Can Stop Them*. URL: `https://www.infoq.com/news/2023/03/stop-cloud-zombies-qcon/` (visited on 04/01/2023).

Wikipedia (2018). *Comparison of web template engines*. URL: `https://en.wikipedia.org/wiki/Comparison_of_web_template_engines`.

Wilke, Claas, Sebastian Götz, and Sebastian Richly (2013). "Jouleunit: a generic framework for software energy profiling and testing". In: *Proceedings of the 2013 workshop on Green in/by software engineering*, pp. 9–14.

Wilke, Claas et al. (2012). "Energy labels for mobile applications". In: *INFORMATIK 2012*. ISSN: 3885796023.

Williams, Chris (2016). "How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript". In: *The Register* March 2016.

Williams, Daniel R and Yinshan Tang (2015). "A Methodology to Measure the Environmental Impact of ICT Operating Systems across Different Device Platforms". In: *JOURNAL OF ELECTRONIC SCIENCE AND TECHNOLOGY* 13.3, p. 15.

Wirth, Niklaus (1995). "A Plea for Lean Software". In: *Computer* 28. ISBN: 00189162/95, pp. 64–68.

Worldwidewebsize (2024). *The size of the World Wide Web (The Internet)*. URL: `https://www.worldwidewebsize.com/` (visited on 03/18/2024).

Xia, Boming et al. (2023). *An empirical study on software bill of materials: Where we stand and the road ahead*.

Xie, Guowu, Jianbo Chen, and Iulian Neamtiu (2009). "Towards a better understanding of software evolution: An empirical study on open source software". In: *2009 ieee international conference on software maintenance*. IEEE, pp. 51–60. ISBN: 1-4244-4897-2.

Xing, Song and Bernd Peter Paris (2003). "Measuring the size of the Internet via importance sampling". In: *IEEE Journal on Selected Areas in Communications* 21.6, pp. 922–933. ISSN: 07338716.

Yang, Shao-Hua (2008). "Automatic Data Extraction from Template-Generated Web Pages". In: *Journal of Software* 19.2, pp. 209–223. ISSN: 1000-9825.

Yuki, Tomofumi and Sanjay Rajopadhye (2013). "Folklore Confirmed: Compiling for Speed Compiling for Energy". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer, pp. 169–184.

Zaidman, Andy (2024). "An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects". In: *AST'24*, Lisbon, Portugal.

Zhang, Chenlei and Abram Hindle (2014). "A green miner's dataset: mining the impact of software change on energy consumption". In: *Proceedings of the 11th working conference on mining software repositories*, pp. 400–403.

Zhao, Li et al. (2016). *Fuel Cells for Data Centers: Power Generation Inches From the Server*. Tech. rep. (Visited on 10/24/2020).

Zhou, Ruogu and Guoliang Xing (2013). "Nemo: a high-fidelity noninvasive power meter system for wireless sensor networks". In: *Proceedings of the 12th international conference on Information processing in sensor networks*. Philadelphia Pennsylvania USA: ACM, pp. 141–152. ISBN: 978-1-4503-1959-1.

Zoio, P (2005). "JavaServer Faces vs Tapestry - A Head-to- Head Comparison". In: *The Server Side*. URL: `http://www.theserverside.com/news/1365201/JavaServer-Faces-vs-Tapestry-A-Head-to-Head-Comparison`.