E-Compare: Automated energy regression testing for software applications

Master's Thesis

E-Compare

Koen Hagen

E-Compare: Automated energy regression testing for software applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Koen Hagen
born in Leiden, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Cover picture: The logo of E-Compare, which is presented in this paper.

E-Compare: Automated energy regression testing for software applications

Author: Koen Hagen
Student id: 5653681

Abstract

As data centres worldwide consume more power than ever, lowering the energy consumption of software is increasingly important. Software energy testing is often unclear due to a lack of comparable baselines. In this paper, we look at the use of regression testing to alleviate some of the struggles with energy testing. We introduce E-Compare, a tool designed to identify energy regressions in software updates by comparing the energy consumption of different versions of the same project. E-Compare is cloud-based, fully automated, and can be implemented in any project with just three lines of code. To validate its effectiveness, we applied E-Compare to thirteen real-world projects, ranging from long-established projects to newer, active ones. Over 700 code changes have been tested. Our findings indicate that energy regression testing can identify energy regressions missed by developers. Some of the indicated energy regressions could be traced back to specific code changes, confirming the tool's accuracy and relevance. However, the tool's usability varies significantly depending on the project, and unexpected energy regressions are relatively rare.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Daily supervisor:	Dr. L. Miranda da Cruz, Faculty EEMCS, TU Delft
Daily co-supervisor:	J. Sallou, Faculty EEMCS, TU Delft
External member:	Prof. Dr. M.M. Specht, Faculty EEMCS, TU Delft

Preface

I am immensely grateful to Luis Miranda da Cruz and June Sallou for guiding me throughout the entire process, providing valuable insights and allowing me to follow my interests. Their feedback and support helped me immensely. I am especially thankful to them for their willingness to have weekly progress meetings during the entire period.

Sustainable software engineering had always been my first choice as a research direction and I am very glad that the research group accepted me and allowed me to pick a topic after my interests.

Thank you to Arie van Deursen and Marcus Specht for finding the time as professors to join my thesis committee and to be willing to judge my thesis.

This thesis marks the end of my academic career. It has been quite a journey across multiple universities and countries. Nonetheless, throughout my career, I always had a strong sense of direction culminating in this very thesis. I am ready to dive into the next chapter of my life.

Koen Hagen
Delft, the Netherlands
June 5, 2024

Contents

Chapter 1

Introduction

We propose and evaluate a new energy regression testing tool, with the goal of finding changes is energy consumption across software updates. This paper begins by explaining the importance of this research in the background. Then we introduce essential terminology, followed by the research question. After that, we review related research and other works. Next, we discuss E-Compare's development, including its concept, technical aspects, challenges, and limitations. After explaining the inner workings of the tool, we use it in the conducted experiments, which includes selecting repositories, tool integration, and comparing commits. The results are presented per selected project and general findings across all projects. Finally, we conclude the study, discuss the limitations and suggest potential future research directions.

1.1 Background

Information and Communications Technology (ICT) consumes a significant portion of global electricity usage, estimated at up to 3.9% in 2022[1]. This usage is expected to increase further in the future[2][3]. Data centres, in particular, have doubled their energy consumption over the past decade[4] and are forecasted to increase fifteen-fold between 2016 and 2030[5]. Currently, data centres consume more energy than the entire aviation industry[4].

Given this rise, it becomes increasingly important to explore ways to reduce data centres' energy consumption. Lower energy consumption would reduce operational costs and increase the environmental impact. Reducing costs could facilitate innovation by making more theoretical operations economically viable. High energy consumption also contributes to a larger carbon footprint if the electricity comes from non-renewable sources.

Several factors contribute to this steep increase. Firstly, the necessity of data centres has grown due to the increased global internet usage. More people are connected to the internet than ever. An estimated 5.44 billion people are now connected to the internet[6], with the average person spending 6.5 hours online daily[7]. Secondly, there is a rise in high-performance computing, including artificial intelligence (AI) and blockchain. Training complex machine learning models and running simulations for scientific research require substantial computing power, leading to increased energy demands. The continuous evolution of hardware capabilities, influenced by Moore's Law[8], provides the necessary computational power for these tasks. Globally, the high-performance computing market is expected to grow by over 15% annually[9]. Thirdly, the adoption of high-level programming languages, such as Python and JavaScript, contributes to increased energy consumption during code execution compared to low-level alternatives. In 2017, 32% of programmers reported using Python within the last twelve months, rising to 54% in 2023, according to JetBrains[10]. Lastly, cloud computing has become essential for organisations seeking scalability and on-demand access to computing resources, storage, and services, significantly impacting the rise of data centres.

These data centres can be stripped down to individual programmes that get run. Constant data streams from these programmes create heat and cost energy. Sometimes these programmes do not behave as expected, due to updates or other things. We call these changes in behaviour "regressions". A regression could be, for instance, more software bugs, higher execution speed or lower performance. Regressions can be positive or negative. When a change relates to the energy consumption of a programme, we refer to it as an "energy regression".

Negative energy regressions pose several issues. They indicate inefficiencies in a program's use of CPU, memory, and other resources, resulting in unnecessary energy consumption. Excessive energy consumption can lead to performance issues, such as elevated device temperatures, reduced computing speeds, or even complete shutdowns. For mobile devices like smartphones and battery-powered gadgets, energy regressions can significantly impact battery life, tethering these devices to power outlets more frequently. For cloud computing, energy regressions can have financial implications, as cloud service costs are tied to application demand, making cloud-based services more costly to maintain.

Addressing energy regressions involves identifying and resolving inefficiencies in the software code, such as unnecessary computations, inefficient algorithms, or resource-intensive operations. The tool presented in this paper aims to detect and mitigate energy regressions during the development and testing phases to ensure that software applications are energy-efficient and provide a positive user experience.

In software development, the importance of energy testing has become increasingly evident. Energy testing ensures that software applications are optimised for minimal energy consumption and that negative energy regressions are identified and removed. This practice is not only about reducing operational costs but also about improving other things, like overall system performance and user satisfaction.

A CI pipeline commonly includes operations related to building, testing, validating, and verifying infrastructure. Only a limited number of organisations have incorporated energy testing into their Continuous Integration (CI) pipelines to automatically assess the energy impact of their code. This is typically applied in cloud computing environments to optimise the energy efficiency of data centres and server farms, ensuring that energy considerations are part of the regular development workflow, reducing the likelihood of introducing energy bugs.

The field of energy testing is still relatively small but continues to evolve with ongoing advancements in tools and methodologies. Various energy profiling tools can be employed to measure and analyse the energy consumption of software applications. PowerLog[1], perf (built into the Linux kernel)[2], powerstat[3], Nvidia-smi[4], provide basic information on energy consumption, typically from the command line. Other tools provide a visual dashboard, like Intel Performance Counter Monitor[5], Scaphandre[6]. These tools provide insights into the energy usage patterns of different system components.

However, many of these energy testing tools face challenges in terms of user-friendliness and clarity, which are critical factors for developers considering tool implementation. Both the readability of the reported information and the ease of integration can influence a developer's decision to implement such a tool in their software. Some tools require users to clone a project twice with different latest commits and compare them locally, which can be cumbersome. Regarding readability, if a tool indicates that a piece of code emits a specific amount of Joules per second, developers may struggle to interpret whether the figure is good or bad without a baseline for comparison. Establishing a baseline is crucial for providing context to the reported information and enabling developers to assess the significance of the energy consumption metrics in their specific software context.

To address the lack of a standard baseline in energy regression testing we propose using the previous version of the software as a baseline. By comparing the current version with the previous one, it is easier to detect changes in energy consumption. This enables developers to pinpoint which parts of the code or functionality changes have led to increased or decreased power usage. Furthermore, it helps developers evaluate whether the benefits of the changes made outweigh potential drawbacks in energy efficiency. Previous versions also serve as historical data points for energy consumption. Over time, this historical data can help establish trends and identify areas where the software has become more or less energy-efficient.

To prove this concept, we implement E-Compare (the tool described in this paper) that uses this approach of energy regression testing by providing users with comparison data alongside standard energy testing data. This approach is not common in energy testing tools or frameworks.

1.2 Terminology

In this section, we introduce key terminology related to software development and testing. This is essential for a full understanding of the work discussed in this paper.

[1]`https://github.com/Thev2Andy/PowerLog`. Retrieved April 26th 2024.
[2]`https://perf.wiki.kernel.org/index.php/Main_Page`. Retrieved April 26th 2024.
[3]`https://github.com/ColinIanKing/powerstat`. Retrieved April 26th 2024.
[4]`https://developer.nvidia.com/system-management-interface`. Retrieved April 26th 2024.
[5]`https://github.com/intel/pcm`. Retrieved April 26th 2024.
[6]`https://github.com/hubblo-org/scaphandre`. Retrieved April 26th 2024.

This section serves as a foundation and can be referred back to for clarity on common concepts.

Energy consumption Energy consumption refers to the amount of energy used by a device or system over a period of time. It is measured in joules (J).

Power usage Power usage refers to the rate of energy consumption at a particular moment. It is measured in watts (W).

Software testing Software testing is an essential part of building software, aimed at finding problems or unintended events in the source code. It involves running (parts of) the code in a controlled environment. Software testing is a broad term that includes various quality attributes such as functionality, performance, and reliability. Medium to large companies often have Quality Assurance (QA) testers who focus solely on testing software.

Energy testing Energy testing is a specific type of software testing that focuses on the energy consumption of the project. It involves measuring the energy consumption of software during execution to identify inefficiencies and ensure that software is energy-efficient. The result is that the software minimises its power usage and impact on a device's battery life. Energy testing can be split into hardware-based measuring and software-based measuring. This paper focuses only on software-based measuring.

- **Unit Testing** Testing individual units or components of the software in isolation.
- **Integration Testing** Testing the interactions between different units or components.
- **System Testing** Testing the entire system as a whole.

We primarily focus on unit testing.

Testing suite The testing suite refers to all tests combined. This could include unit tests as well as any other tests written for the project.

Testing techniques There is a variety of testing techniques that can be utilised depending on the developer's requirements. Important ones for us are black box testing and regression testing.

Black Box Testing Testing the software without knowledge of its internal structure or implementation. There is no access to the underlying source code, hence the name "black box." The tester focuses on testing the functionality of the software based on its inputs and outputs, typically validating a set of output values based on their expectations.

Energy regression Energy regression refers to the unintended increase or decrease in a software's energy consumption after changes to the source code. Throughout the text, we might refer to energy regressions as regressions.

Energy regression testing Energy regression testing refers to finding unintended changes in the energy consumption of software as a result of changes to the source code. The term "energy regression testing" was inspired by Danglot et al.[11].

This manuscript is deeply intertwined with Git terminology. Here are some basic Git terms used:

Repository A repository is a storage space where your project's files and revision history are kept. It's like a folder for your project that tracks changes over time. GitHub allows you to run workflows accessing the repository. The term "repository" is used similarly to "project" but in the context of Git.

Commit A commit is a snapshot of the source code at a specific point in time. It contains changes made to files in a repository. Each commit has a unique identifier and includes a message describing the changes made.

Branch A branch is a parallel version of a repository's code. It allows developers to work on separate features or fixes without affecting the main development code. Branches are commonly used for feature development, bug fixes, or experiments. A typical workflow is to build new code additions in a separate branch and merge them into the main branch when completed.

Head branch The head branch is the current branch where development or specific changes are being actively made. It is typically a feature branch or a topic branch created from the main branch. Developers use the head branch to implement new features, fix bugs, or experiment with changes. Once the changes are tested and reviewed, the head branch is usually merged back into the main branch.

Base branch The base branch is the branch against which a new branch is compared and into which it will be merged. It is typically the main or master branch, but it can be any branch that serves as the foundation for development. When creating a pull request, the base branch is the target branch where changes from the head branch will be integrated.

Pull Request (PR) A pull request is a request to merge changes from one branch into another. It allows developers to review, discuss, and collaborate on code changes before merging them into the main branch. E-Compare uses these to show its results.

1.3 Research question

Based on the problems identified in section 1.1, we phrase our research question as follows:

Can automated energy regression testing contribute towards finding energy regressions in software applications?

This research question is true if the following two criteria are met:

1. Energy regression testing can find energy regressions in software applications
2. After initial setup, energy regression can be tested automatically, without any manual interaction from the developer.

To answer this research question, we first create an energy regression testing tool that reports its findings to the developer making changes to their software application. Then, we apply this tool to several open-source projects. For each of these projects, we retroactively apply the tool to previous code changes, simulating as if the code change was new. The findings of all these code changes are then compared and checked for any energy regressions that could potentially be mitigated had the developer been aware of their existence.

"Automated" means testing without manual interaction from the developer, typically done using Continuous Integration / Continuous Development (CI/CD) pipelines.

Chapter 2

Related Work

Energy regression testing is a relatively unexplored domain in computer science. This chapter discusses previous research related to this study, providing a foundation for our own work. We can subdivide it into energy testing and regression testing, which we will discuss individually first.

2.1 Energy Profiling

Power usage measurement can be conducted using software-based or hardware-based approaches. In this paper, we focus exclusively on software-based approaches, considering hardware-based approaches out of scope.

Research on energy consumption of software is limited compared to research on general computational speed. While higher computational time often correlates with higher energy consumption, this is not always the case and definitely not a one-to-one relationship. There are instances where energy consumption is reduced without a significant impact on execution speed[12].

There are many different tools that can be used for measuring energy consumption of software. However, due to differences in hardware and operating system, specific tools apply to specific use cases. Cruz gives an overview of six commonly used tools and in what kind of situation someone would apply them[13]. On Linux, when you have an AMD processor, the *PowerTOP* can be used as an energy profiler. Otherwise, if you use an Intel CPU, *PowerStat, Perf* or *Likwid* can be used. On Windows or MacOS, PowerGadget or Intel PowerLog can be used. Though, these profilers only work with Intel CPUs. GPU-intensive programmes can be best measured with Nvidia-smi if the system uses a Nvidia GPU. More recently, *EnergiBridge*, strives to solve this platform-dependent landscape by creating a cross-platform energy profiling tool[14].

Several frameworks have been developed to profile and enhance the energy efficiency of computing systems. These frameworks focus on identifying design patterns within applications to optimise performance effectively. For instance, the *GEOPM* model is an energy management framework designed for extendability, allowing for future energy management strategies[15]. The *POSE* model enables developers to evaluate the benefits of decreased energy consumption in their applications, helping identify opportunities for power optimisation[16]. Maranthos et al.[17] provide a framework that uses static analysis to estimate the performance and energy consumption of applications.

Multiple models for energy consumption estimation tools have been proposed. These tools utilise AI to estimate energy consumption. An older model[18] lays the groundwork for measuring large, complex systems, identifying cooling as the most energy-intensive operation within enterprise data warehouses. This depends massively on the database size; I/O subsystems consume the most power for smaller databases, while CPUs use the most power for larger databases[18]. The *CHAOS* model is a linear regression model using various features, validated on six different server hardware configurations[19]. Kim et al. propose a model[20] that utilises time-dependent variables, such as the number of occurring events. *GreenOracle*[21] provides a pre-trained model for estimating energy consumption through training on data from a wide range of applications. By the same people, *GreenScaler* introduces a software energy model that estimates CPU utilisation based on randomly generated or developer-specific tests[22].

Energy models often focus on battery-powered devices, such as mobile phones[22]. This focus is logical, as the consequences of high energy consumption are easily observed in reduced battery life and slower processing speeds.

In cloud environments, measuring energy consumption accurately is more challenging since conventional measurement methods, such as RAPL, are often unavailable. *Interact*[23] is a tool that uses machine learning to predict energy consumption, achieving an 8% error rate for average power estimation. The Green-coding team also developed a machine learning model for predicting energy consumption based on *Interact*[24]. *GreenFlow*[25] measures the energy consumption of data stream processing systems in the cloud. The model is tested on multiple common data streaming platforms, such as Apache Flink and Apache Kafka. GreenFlow uses Scaphandre under the hood.

Research has been done towards measuring energy consumption in a CI environment. [26] broadly outlines the requirements for incorporating energy consumption measurements in a CI pipeline. Eco-CI[27] allows a developer to estimate energy consumption in GitHub and GitLab workflows by positioning itself in the CI pipeline.

To estimate the energy consumption of active programs, the energy consumed when the server is idle must be subtracted from the total energy consumption of the server. Quesnel et al. propose a model to calculate the idle state energy consumption[28]. *TEEC* is a model that estimates changes in power at runtime based on software execution, considering CPU, memory, and disk metrics[29].

Despite the availability of several tools and methods, many developers are unaware of how to measure the energy consumption of their code[30][31]. However, some developers express a willingness to sacrifice certain features to reduce energy consumption[31].

Addressing code does influence power usage. Research shows that minor changes in data handling can lead to significant changes in energy consumption[32]. Other code-based solutions have also been shown to decrease energy consumption, such as compressing infrequently accessed data and reusing software services[33].

2.2 Regression Testing

Regression testing ensures that recent code changes do not introduce new defects or negatively impact existing functionality. This is crucial in maintaining software reliability and performance. Compared to energy testing, performance testing is much more widely adopted. Regression testing frameworks and models function similarly to their energy testing counterparts, thus we can look at them to get inspired.

We find that recent research into performance regression testing moves away from measuring every change. Instead, opting for predicting the emergence of performance regressions to limit the testing time. *Perphery*[34] is an performance prediction model that runs much faster that regular performance regression tests. Perphecy can identify 85% of code modifications that impact performance while saving up to 83% of benchmarking time. It uses eight different performance indicators that range from "The number of deleted functions" to "the length of the changed functions". An alternative model to Perphery is *PRICE*[35], which aims to predict performance regressions by creating a prediction model that uses some other static and dynamic metrics such as "the percent overhead of the top most called function that was changed". This model was comparable with the Perphecy model, with a slightly lower hit rate but a slightly higher dismiss rate compared to Perphecy.

PefImpact[36] is a programme that automatically traces back performance regressions to specific code changes. It identifies code changes likely causing performance regressions by using a genetic algorithm to find inputs that slow down execution in the new release. These kinds of tools could potentially be adapted for use with energy regressions.

In general, looking at the scientific landscape of regression testing could give us an insight in where energy regression testing in currently lacking, as well as where the field might be heading in the future.

2.3 Energy regression testing

Energy regression testing is a relatively unknown term. Research on the subject is limited or did not use the same terminology.

Though, some research towards energy regression testing has been conducted. We find the first occurrence of the term in a paper by Danglot et al.[11]. They demonstrate that energy regressions can be measured stably and accurately based on studies involving seven Java projects. They trace back multiple energy regression to pieces of code that were added in commits analysed. Their tool, called "diff-JJoules", is able to measure the energy consumption of commits by measuring locally on the developer's computer. This requires some manual labour on part of the developer. Other paper that do not use the term are also relevant. A paper by Hindle et al. proposes a methodology that relates software source code changes to energy consumption[37]. Instead of running the testing suite, they measure the energy consumption at actual use of the software. One interesting finding of theirs is that a single version update to Firefox contributed to a decrease in power usage of the monthly power use of ninety American households per hour, assuming everyone upgraded to the new version. This further emphasises the importance of energy efficient code.

In conclusion, while energy regression testing is still a developing field, the existing research provides a solid foundation for further exploration. Our study aims to build upon these findings to create a robust tool for detecting energy regressions in software applications.

Chapter 3

Approach

In this chapter, we discuss our approach to the problems with the final goal to answer the research question. This approach is built upon previous research stated in Chapter 2 and aims to address as many issues that arise with current approaches to energy testing and regression testing.

First, we look at our approach from a conceptual level, only discussing the fundamental workings without discussing any technical aspects. Then, we go more in-depth into our proposed solution, where we break down the created tool on a technical level. After that, we discuss some of the challenges that came up throughout development. Finally, we talk about the limitations of this approach and some future directions that the tool can be taken.

3.1 Conceptual Approach

A potential solution for improving software tests will involve measuring the energy consumption during test runs using a wrapper around the testing suite. This approach ensures that the tool does not interfere with the actual testing logic while still providing information. This wrapper will track energy usage for every code change and generate data when the developer is about to implement changes to production and report that. These reports will aid developers in making informed decisions about whether to approve or deny code changes. The reports can aid the developer in three different ways: detect code changes that significantly increase energy consumption and may require optimisation, confirm that changes intended to improve efficiency are having the desired effect, and ensure that new code maintains or improves upon the energy efficiency of the existing production code.

Ideally, implementing the tool will be straightforward and accessible for anyone integrating it into their projects. Comprehensive documentation will provide a clear guide on integration, customisation, and other useful information, helping users understand the tool's inner workings. There will be no user registration required, and settings changes will be minimal. Changes to the testing suite will also be kept as low as possible. However, implementing this solution may necessitate some inevitable changes to the original test workflow. The tool will need specific trigger moments to function correctly, which could result in tests being run under different conditions.

The developer is free to use any type of test to measure the energy consumption of, such as unit tests, integration tests, functional tests, acceptance tests, etc. They have to use their own judgement to decide whether their tests fit, as some testing types might not be impacted by changes in the source code. Some testing types will also introduce more overhead compared to others. Typically higher-end tests, like end-to-end tests, are less stable than lower-end tests, like smoke tests or unit tests.

The quality and coverage of the tests significantly affect the usefulness of the results. Low test coverage increases the risk of missing energy regressions in untested code. Any energy regressions in code that is not being tested will not be found. Conversely, with low coverage, any detected regressions may appear more significant than they are. Developers must evaluate whether this tool and similar tools are appropriate for their projects and whether the results align with their tests and source code.

To allow for the widest applicability, we will use a black-box approach, testing any command that can be run in a GitHub Runner environment. This approach's main advantage is its versatility, allowing the testing of various commands and scenarios without needing specific knowledge of the tested commands' internals. This ensures that the tool remains adaptable and capable of testing new technologies without requiring constant updates.

Energy consumption is inherently non-deterministic and influenced by various factors, both internal and external. Hardware variability, background processes, and environmental conditions can all impact energy consumption, meaning running tests twice could yield different results. For some tests, this fluctuation may be more pronounced. This may cause insurmountable issues for some projects. Further discussion on this topic can be found in section 3.4.

3.1.1 Example scenario

Consider a scenario where a developer introduces a new feature that involves complex data processing. The wrapper records an increase in energy consumption by 10% compared to the production code. The report highlights this increase in red, prompting the developer to review the changes. Upon investigation, the developer identifies an inefficient algorithm and optimises it, reducing the energy consumption back to acceptable levels. The final report shows a minor 2% increase, which is within the acceptable range, allowing the developer to proceed with the deployment.

3.2 Implementation of the solution: E-compare

To address the problems with energy testing, we propose E-Compare. E-Compare is an energy estimation tool that creates comparisons between different versions of software. In this chapter, we explain our approach from a conceptual level, and then, discuss the inner workings of E-Compare from a technical point of view.

The source code can be found on GitHub[1]. This can be used to replicate the experiments. Documentation and implementation guides can be found on its website: `https://koenhagen.github.io/E-Compare/`.

The name E-Compare is short for "energy compare", reflecting the tool's primary function of comparing energy consumption. The first three letters, "eco", are green and bolded to highlight the tool's ecological focus.

3.2.1 Tool breakdown

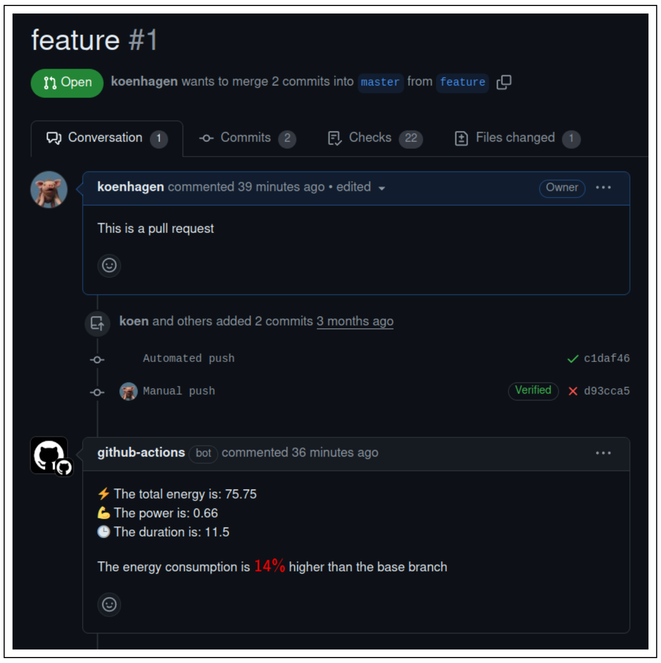
E-Compare operates as a reusable GitHub Actions workflow. GitHub was chosen for its status as the leading platform for software storage and sharing and its extensive free CI platform. The tool is called as a reusable workflow[2] from a standard YAML file. YAML files are used on GitHub to define your workflow configuration. These reusable workflows function as if you refer to another workflow from another location. This other workflow is our tool E-Compare. E-Compare requires some extra parameters where the developer can pass through extra information to the tool. Most importantly, the `run` parameter is used where the developer can pass through the command that they want to measure the energy consumption of. This will typically be the command that runs the testing suite. Instead of one command, the developer can also pass a pipeline of commands or multiple commands divided by a semi-colon, in the `run` parameter. Instead of running the command directly, it is now run by the tool while its energy consumption is measured.

The developer has to trigger the tool through push and pull requests, each performing different actions. Other triggers will result in undocumented behaviour. Every push generates an energy report for the specific version of the software pushed to GitHub. The report will include details such as test duration (in seconds), energy consumption (in joules), and power (in watts). It will also provide comparative data between the new and existing production codes, highlighting the percentage change in energy consumption. Significant changes will be highlighted in the report as green or red to indicate positive or negative regressions, respectively. A change is deemed significant if it is exceeding 5% or below falling -5%. When a pull request is made, E-Compare compares the latest energy report of the head branch with that of the base branch (typically main/master). This comparison is visualised and reported to the developer through a GitHub comment on the pull request. When a new push is made to a branch that already has an open pull request a new comment is made with the updated information. Figure 3.1 shows an example of such a comment on a pull request.

Other than the `run` parameter, there is one more parameter that E-Compare requires. This is the `GITHUB_TOKEN` parameter. This token is required by E-Compare

[1] `https://github.com/koenhagen/E-Compare`

[2] `https://docs.github.com/en/actions/using-workflows/reusing-workflows`. Retrieved May 30th 2024.



feature #1

Open koenhagen wants to merge 2 commits into `master` from `feature`

Conversation 1 | Commits 2 | Checks 22 | Files changed 1

koenhagen commented 39 minutes ago • edited · Owner

This is a pull request

koen and others added 2 commits 3 months ago

```
Automated push                      ✓ c1daf46
Manual push              Verified   ✗ d93cca5
```

github-actions bot commented 36 minutes ago

⚡ The total energy is: 75.75
💪 The power is: 0.66
🕒 The duration is: 11.5

The energy consumption is 14% higher than the base branch

Figure 3.1: Example of a GitHub comment generated with E-Compare.

to obtain the necessary permissions regarding access to the repository. There are also other parameters available, but those are all optional.

Figure 3.2 shows a breakdown of the different external applications used by E-Compare. Zooming into the striped box from Figure 3.2, the inner workings of E-Compare are detailed in Figure 3.3 and Figure 3.4. The tool breakdown is split into two figures because the push and pull_request triggers function differently within the tool. Generally, the push trigger creates energy reports, while the pull_request trigger creates the visualisation shown to the end user. However, the push trigger may also create the visualisation if a pull request already exists. We now describe both applications of the tool:

Push A push triggers the most complicated part of the tool. When a commit is pushed to the repository, E-Compare starts running the test command that gets passed through. Before that, it starts its CPU usage measurement service. Once the tests are complete, the tool stops the measurement service to ensure accurate results by ensuring no other functions are run between the start and end of the measurement period. The CPU usage data is then processed by the energy estimation AI, which, combined with static hardware data, provides an energy consumption estimate. Then, a report is generated containing the AI results, test duration, and power consumption. This report is then stored in the `energy` branch of the project's repository. If the energy branch does not yet exist (if this is the first created report), the tool will create it. After generating the report, the tool checks if the push is part of a pull request. If

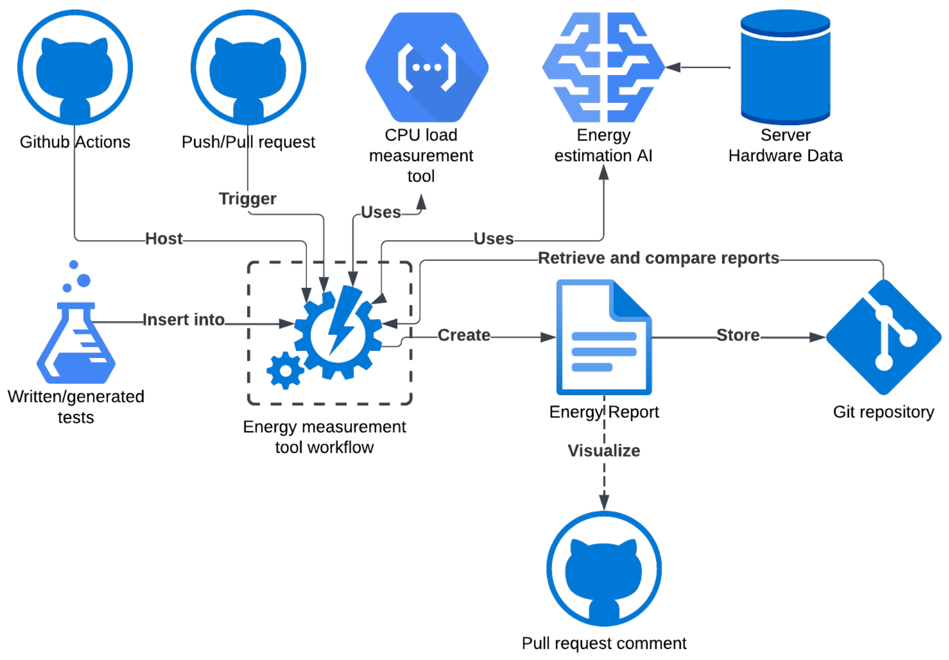


Figure 3.2: Breakdown of how E-Compare interacts with external applications.

so, the tool informs the developer about the commit's energy consumption, following the same process as for a pull request, described below:

Pull request The process triggered by a pull request is simpler than that triggered by a push. The pull request trigger aims to provide the developer with information collected during pushes. It does this by identifying the commit where the branch was branched off (the fork point). It then retrieves the energy report from the energy branch for both the fork point commit and the current commit. If all data is correctly retrieved, the tool compares the two reports. This comparison data, along with data on the latest commit, is then reported to the developer through a comment on the pull request.

3.2.2 Detailed Implementation and Rationale

GitHub runs all its workflows on the Azure cloud. One limitation of this cloud-based approach is that E-Compare does not have access to detailed hardware information. Namely, power usage numbers are not available. To remedy this, E-Compare estimates energy consumption based on accessible information, such as CPU load. For this part we import a third-party artificial intelligence model created by Green Coding Berlin[3]. Specifically, we use their XGBoost model. The accuracy of this model

[3]`https://github.com/green-coding-berlin/spec-power-model`. Last retrieved 1 June 2024.

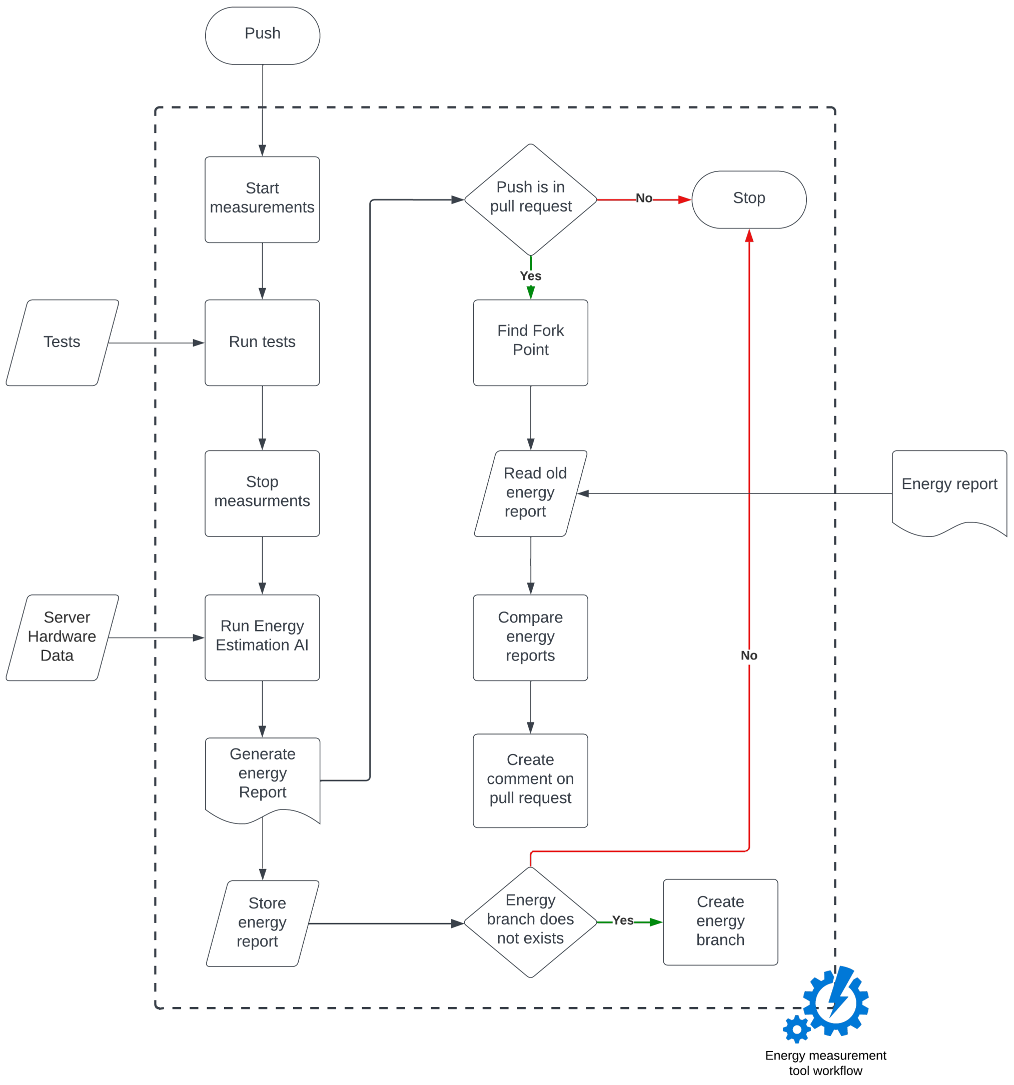


Figure 3.3: Flowchart of the inner workings of E-Compare in the case of a push. Every push will trigger the tool to start making measurements of the specified tests. After these are done and processed with the AI tool, an energy report is generated and stored on the GitHub repository. If a branch for the reports does not exist yet it creates it. If the push is already tied into an existing pull request, it runs similar code to the pull_request trigger.

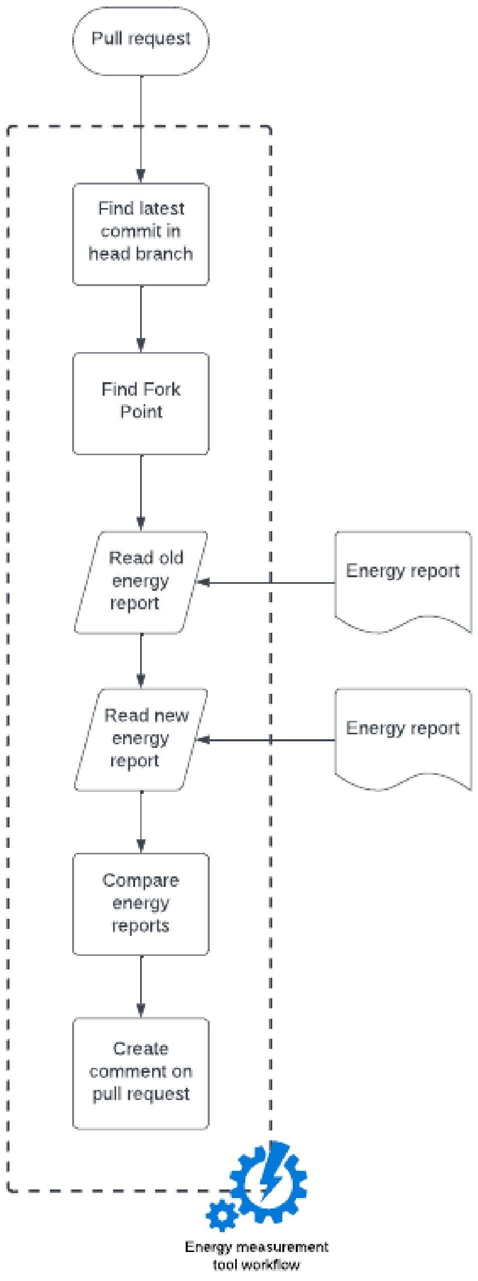


Figure 3.4: Flowchart of the inner workings of E-Compare in the case of a pull request. First, the tool has to figure out which commits to compare. It does this by finding the latest commit in the head branch and then finding its fork point with the base branch. After retrieving both energy reports, it compares the two after which the findings can be placed in a comment on the pull request.

ranges between 0% and 9% depending on the hardware[38].

The main part of E-Compare is written in JavaScript, though as said in Chapter 1, JavaScript is a high-level programming language that creates a lot of energy overhead. To minimise this overhead, E-Compare uses a C-language program for measuring CPU load. C is a low-level programming language that compiles directly to machine code, avoiding runtime interpretation. This contributes to faster and more efficient execution. C provides a high degree of control over hardware and system resources. This level of control allows E-Compare to optimise code for performance and minimise unnecessary overhead.

Other than CPU load, the AI model utilises other internal variables to make energy estimations. There are the `MODEL_NAME`, `TDP`, `CPU_THREADS`, `CPU_CORES`, `CPU_MAKE`, `RELEASE_YEAR`, `RAM`, `CPU_FREQ`, `CPU_CHIPS`, `VHOST _RATIO`. All of these variables can be retrieved from the Azure cloud environment. The next part explains some of these variables. The model name is used, which specifies which machine is used for the workflow. This could be any of the six models that Microsoft Azure uses (on which GitHub is hosted)[24]. The thermal design power (TDP) specifies the maximum amount of heat generated by the server that is supported by the cooling system. CPU cores are physical processing units, while CPU threads are virtual virtual sequences of instructions given to a CPU. Threads and cores do not have to be equal, due to hyperthreading. `CPU MAKE` specifies whether the CPU is from AMD or Intel. The release years are the years that the specific model is released. The RAM is the amount of RAM in GBs. The CPU frequency is the CPU clock speed in GHz. 1 GHz would execute 1 billion cycles per second. The CPU chips are the number of chips that are installed on the motherboard. In our case, all machines have only one chip installed, as GitHub only uses machines with one chip. vHost ratio is put in place for shared cloud servers. It specifies the ratio of threads available to the user. If only half of the threads are available this variable would be 0.5. All hardware data was collected from the SPEC[39] benchmark database[4].

Storing the energy reports generated by the tool within the GitHub repository ensures both accessibility and security. Reports are placed in the .energy folder within the designated `energy` branch, making them easily accessible alongside the source code. This allows developers and team members to easily review them. The integration with Git facilitates tracking changes, reviewing historical data, and correlating energy metrics with specific code revisions. An example of an energy report is shown in Listing 3.1

[4]`https://www.spec.org/power_ssj2008/results/power_ssj2008.html`. Retrieved January 23rd 2024.

Listing 3.1: Energy report example

```json
{
    "total_energy": 645.6717832225576,
    "power_avg": 0.45889963270970685,
    "duration":140.7
}
```

From a security perspective, local storage within the GitHub repository enhances data security by keeping potentially sensitive information within the version-controlled environment. This approach mitigates risks associated with storing sensitive energy-related details on external or unfamiliar servers.

E-Compare requires specific GitHub permissions to function correctly. The workflow's existing permissions may suffice. `contents: write` is necessary for storing data on the `energy` branch and creating new branches. `pull-requests: write` is required for commenting on pull requests. For private repositories, `actions: read` is necessary for the workflow to access the GitHub token that is passed to the tool. This token is used to make the necessary API calls.

Setting `permissions: write-all` will also work, though it is not recommended because it grants overly broad permissions that can pose security risks. This setting grants the workflow access to write to all repositories within an organisation, potentially allowing other called reusable workflows to make unintended or harmful changes across this and other projects.

By default, E-Compare runs expect Unix shell commands or shell scripts (suffix .sh). If users instead would like to run bash, they can do that by setting the `isBash` parameter to `true` when passing variables to E-Compare in the workflow YAML. This parameter is set to false by default and only accepts `true` or `false`.

As said in 3.1, the results could be different despite testing the same code twice due to the non-deterministic nature of the tool. Developers can mitigate this through the `count` parameter in E-Compare. The `count` parameter specifies the amount of times E-Compare needs to run the unit test. Setting the parameter to ten will make E-Compare run the tests ten times and return the average of these. Developers have to decide for themselves what would be a good number for their projects. More stable projects do not require a high count compared to projects with a lot of fluctuations. In most cases, it is okay to allow some tolerance for inaccurate results. This parameter only accepts a number.

3.2.3 Finding a common ancestor

Sometimes, the base branch may have updated since the head branch was created, resulting in merge conflicts that need to be resolved. In such cases, a comparison between the latest version of the base branch with the head branch can be inaccurate, as changes in the base branch can affect energy consumption. The goal of the tool is to report the change in energy consumption of the code in the head branch. To circumvent this, E-Compare compares the latest version of the head branch with the fork point of the two branches. The fork point is the most recent commit where the two branches were the same (the “common ancestor”) or the commit from which the head branch was created.

Let’s show some examples. The listings represent a timeline where older commits are further to the left. Every point (o, A, B, etc.) represents an update to the code (commit). Every parallel horizontal line is a unique branch. The slashes show spots where a new branch was created, while backslashes show merges of two branches.

1. Take a look at Figure 3.5, where we have one main branch ending in point A and two other branches ending in points B and C. If we want to merge point B or point C with point A, the fork points will be 1 and 2 respectively because that is where the branches split off. These points are the common ancestors.

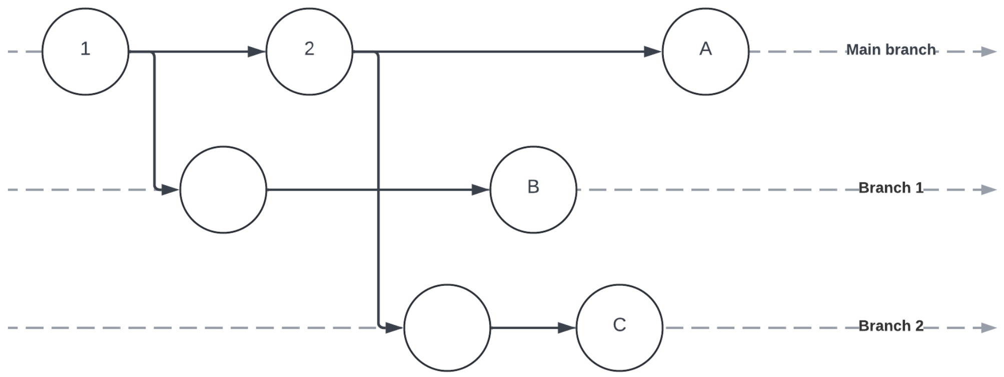


Figure 3.5: Fork point example 1

2. Now let’s say we have merged point B into point C resulting in point M, the common ancestor of this would be point 1. If we were to merge point M into the main branch, the common ancestor would be point 2, despite point 1 also being a common ancestor. This is because point 2 is a newer common ancestor than point 1.

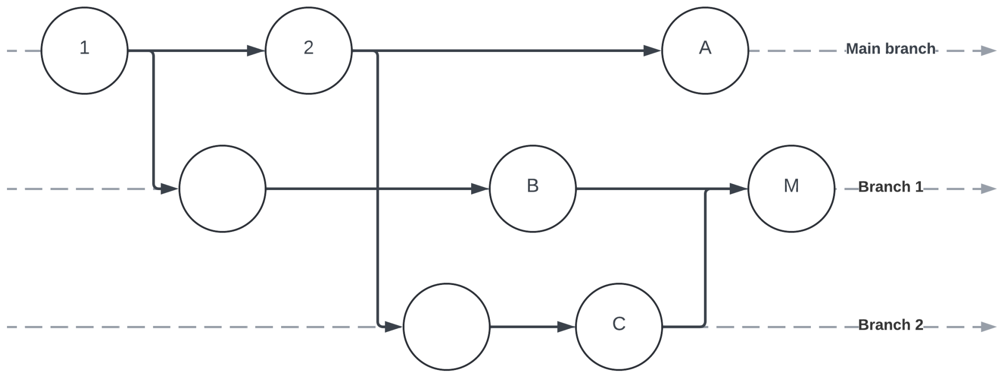


Figure 3.6: Fork point example 2

3. When a branch is updated with a new version of the main branch, as shown in Figure 3.6, the fork point updates. In this case, the new latest common ancestor will be point 2.

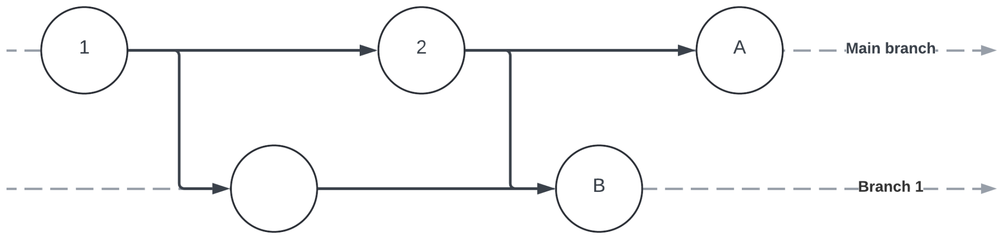


Figure 3.7: Fork point example 3

4. It is not always possible to find a fork point. This can happen, for instance, when cherry-picking or when a criss-cross merge occurs, as illustrated in Figure 3.7. In such cases, E-Compare cannot find a common ancestor and will not create a pull request comment. Comparing the head branch with either point 1 or 2 would lead to a faulty comparison and could misinform the developer.

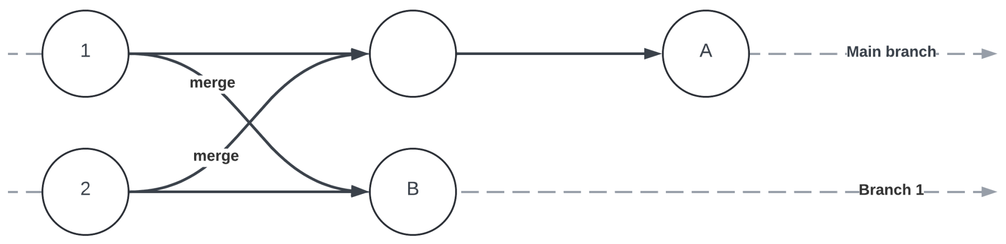


Figure 3.8: Fork point example 4

3.3 Challenges

This section discusses some challenges encountered while developing E-Compare. These challenges are documented to showcase our development journey and to allow other developers to understand the reasoning behind some functions in the source code of E-Compare, to learn from it, and avoid similar pitfalls.

Integration issues Developing E-Compare posed several engineering challenges, notably the inexplicable failures encountered when applying the tool to certain projects. Given its versatile nature, E-Compare needs to seamlessly integrate with diverse project stacks, making these failures particularly frustrating but understandable. While some issues were resolved, others persisted.

Buffer overload One issue was a buffer overload when tasks took too long to run. Using a high `count` often meant that the CI environment crashed. Due to how the tool works, any output normally printed in the console is now saved until the entire process is done and then printed to the console. This limits the energy required for the print function but requires more memory if the printing buffer becomes lengthy and sometimes even causes crashes. The solution involved increasing the buffer length to prevent tool crashes caused by a lengthy stdout stack. As the tests ran, the stdout stack increased. If the tests continued for too long, the entire tool could crash, failing to measure energy consumption.

Package versioning Another common issue had to do with package versioning. This occurred when the project CI used the same Python packages but different versions. This sometimes caused clashes in dependencies resulting in crashes. Sometimes imported functions work differently causing unexpected behaviour. The solution was to create a separate virtual environment tailored specifically for E-Compare. This means that a new virtual environment is created within the virtual environment that GitHub sets up. In this new environment, all languages and packages required by the tool have to be reinstalled to ensure smooth functionality without disrupting the project processes.

Arbitrary functioning The seemingly arbitrary functioning of the tool depending on the project hinders the usability of the tool. When testing the tool on different types of projects, the tool often fails without explanation. Some projects broke across different versions of the runtime environment, with no other changes to the settings or source code.

Automating backtracking A significant amount of time was spent trying to automate a backtracking process. This process involved gathering data for older commits from before the tool was imported into a project, allowing the developer to see data from earlier commits and get comparison data from newer branches split off from these commits. The idea was to have a `historic` parameter that backtracks a certain number of commits automatically depending on the developer's specifications. The tool would then create separate branches for all previous commits and create pull requests with the main branch. This system could have alleviated much of the work from the experiments in Chapter 4. The problem was that pull requests done through an automated action did not trigger the `pull_request` workflow trigger. This means that there is no way to automatically trigger E-Compare without manual interaction with GitHub.

3.4 Limitations and future

In this section, we discuss the limitations of the current implementation and design of E-Compare. These limitations range from practical constraints, like hardware limitations, to conceptual issues. These limitations are inherent issues that arise because of the core concept and implementation of the tool and they are different from the challenges described in section 3.3 that discuss problems faced during development and the solutions that were found. We also explore potential improvements to the tool that could be made in the future.

3.4.1 Technical limitations

It is important to recognise that E-Compare, like any tool, has its limitations. In this section, we'll highlight these limitations to provide a balanced view of its utility. Additionally, we'll explore potential future directions for refining energy testing methodologies, aiming to address the shortcomings of E-Compare.

Limited measurability The main limitation of E-Compare is its inability to measure the energy consumption directly from the machine hardware. This limitation can largely be overcome by using an AI model that estimates energy consumption instead. In the worst case, this can lead to inaccuracies in identifying energy-intensive software components or inefficiencies, potentially resulting in misguided optimisation efforts.

Overhead from instrumentation While E-Compare tries to minimise disruptions and maintain a consistent testing environment, it's important to acknowledge that any instrumentation or monitoring tool, including E-Compare, may introduce some level of overhead. This overhead has the potential to impact the accuracy of energy consumption measurements and, consequently, influence the results to a certain extent. E-Compare consumes additional system resources (CPU, memory, etc.) while collecting energy data. This utilisation could impact the availability of resources for the software under test and affect overall execution time, potentially impacting its energy efficiency.

Server overhead Besides the overhead from E-Compare, the server may also introduce overhead. This is especially important to keep in mind as there are multiple machine types that GitHub utilises to run workflows. Each of these machines shows different characteristics. Research by Green Coding shows that energy variability can range from 0% to 9% depending on which machine is used[38]. CPU types 8272CL and 8370C perform with nearly no variability, while CPU type E5-2673v4 performs the most inconsistently with 9 percent variability. Unfortunately, GitHub does not allow the developer to specify a specific CPU type when starting the workflow.

Operating system limitations A notable limitation of the tool is that its testing and validation have focused on the Ubuntu-latest servers provided by GitHub Actions, which are running on Ubuntu 22.04. While it has demonstrated usefulness in this environment, this does not mean that the tool performs the same on MacOS or Windows servers. E-Compare's performance on MacOS or Windows servers has not been systematically tested or validated. As different operating systems exhibit unique characteristics, the tool's compatibility and reliability in diverse server environments remain an area for exploration and potential improvement. Since the underlying energy estimation AI is exclusively trained on Ubuntu servers, we have to assume that the results will be inaccurate. Future iterations and developments could include testing across multiple operating systems to ensure broader applicability. Though Linux being the most common operating system for servers, the tool already offers wide usability.

Time reporting limitation Another limitation of E-Compare is that the tool only reports full seconds. This is due to the way the underlying CPU load measurement tool functions. It reports the CPU load every second until it gets deactivated. This limitation may be a problem when the unit tests are very short and fractions of seconds might have a large percentage difference. However, as tests get more expansive and cohesive, this limitation becomes less of an issue.

Reusable workflow incompatibility E-Compare cannot be integrated into every testing set-up. As E-Compare uses a reusable workflow system, it is incompatible with tests that already use reusable workflows. GitHub does not allow reusable workflows to call each other, so this limitation cannot be resolved as long as E-Compare uses reusable workflows. A reusable workflow is a workflow that uses the keyword `uses:` instead of `run:` and then refers to a location instead of a shell command.

No trigger filter A typical workflow has a couple of trigger filters implemented so that it does not trigger when it is not needed. A common filter is updated to .md files (readme files), for which tests are typically not needed. These filters interfere with E-Compare when the tool tries to compare a commit with a skipped commit. The main downside of this limitation is that the developer is forced to run tests for unnecessary commits, resulting in unnecessary time and energy costs.

3.4.2 Conceptual limitations

Conceptual limitations refer to the fundamental limitations of E-Compare. Unlike technical limitations, which stem from hardware, software, or implementation constraints, conceptual limitations are caused by the underlying principles and assumptions of the tool.

Testing frequency mismatch One important consideration is that energy testing does not have a one-to-one equivalence with real-world use. For instance, with unit tests, the aim is to test every possible input for every function once. However, some functions or inputs may be used more frequently than others. A clear example is a log-in and register function. Logically, the register function is triggered only once per account, while the log-in function may be used much more often. Energy testing will not differentiate based on frequency of use.

Developer procedures Another conceptual limitation is that E-Compare requires a specific procedure from the developer. Developers have to utilise branches and merges, tests have to be run from the CI/CD environment, and commits have to be done manually. Otherwise, the tool does not work. While these are best practices, not everyone deems them necessary, and not every project benefits from this specific procedure. Getting such projects to use E-Compare can only be done through behavioural change, which is a tall order and too much to ask for many. Projects sometimes even have to be changed beyond tool implementation. Workflow triggers might need to be changed, which could interfere with the current process.

3.4.3 Future

If development on E-Compare were to continue, there are several directions that could benefit the tool.

Enhanced reporting Starting with the reporting part, currently, the results are fairly limited, showing only one percentage number to indicate the difference between the two software versions. The tool could show more useful information to the end-user. For instance, when using the count variable, E-Compare could show all the results instead of providing just the average. Outlier detection could help obtain a more natural average. Providing a standard deviation next to the average could tell the developer a lot about the consistency of the results. More reporting could involve developing interactive dashboards, charts, and graphs that visually represent energy consumption data over time, across different code changes, and in relation to other performance metrics.

Detect changes in tests Relating to this, another useful feature would be to identify if the tests have been modified in the commit. A modification to the tests will almost always be reflected in E-Compare's results. If more tests are added, the total energy consumption of all tests will be higher, but this will not translate to the actual energy consumption of the project in deployment.

Cross-platform versatility In the future, E-Compare should work on its versatility across different platforms and projects. Theoretically, due to the black-box principle, the tool should work with any setup. In practice, however, there are many unexplainable hitches that prevent the tool from working. Ironing these out will increase the usefulness of E-Compare as it becomes more widely applicable. Related to this, adapting the tool to work on MacOS and Windows is another great way to make the tool more versatile.

Other CI platforms Furthermore, in the future, E-Compare could be adapted to other CI platforms besides GitHub. Functionality for GitLab would be a logical next step for the tool, especially as the underlying AI is already trained on the machines that GitLab runs its servers on.

Chapter 4

Experiments

The primary goal of these experiments is to shed light on whether automated energy regression testing can effectively identify energy regressions in software applications. The results will provide valuable insights, like the benefits and limitations, for researchers and practitioners in the field of energy-aware software development. We do this by conducting a retrospective analysis of historical commits to the code of selected projects on GitHub. This part can be done instantly for a project as the data is already available.

The tool is wrapped around the existing unit tests of the tested software. Unit tests were chosen because they are a commonly used method of testing code. Due to the nature of unit tests, being tests that test tiny pieces of code or functions, they would logically be fairly consistent across multiple iterations. Other tests such as integration tests, system tests or acceptance tests can also be used, but they are less common and potentially less consistent.

The procedure consists of Repository selection, Tool integration and then Commit comparison.

4.1 Repository selection

To select repositories, we need to create an objective selection method. We settle on two different methods: First, we look at a list of the most starred repositories on GitHub. For this, we leveraged EvanLi's GitHub-Ranking website[1], which dynamically compiles and updates a list of the most-starred repositories for each programming language. Second, we look at the top trending repositories, provided by GitHub[2], which publishes a weekly list of currently trending repositories based on some undisclosed variables.

For both lists, we specifically look at the list from January 2024. Any potential changes in ranking that the list went through after that are disregarded. With the most-starred list, we look at the first hundred repositories, which is all that is being

[1]`https://evanli.github.io/Github-Ranking`. Last retrieved 1st of January 2024
[2]`https://github.com/trending`. Last retrieved 1st of January 2024

tracked. With the trending list, we look at the list for each week of January 2024, which comes down to twenty-five repositories per week. Specifically, we look at the lists released on the 1st, 8th, 15th, and 22nd. Historically data on the trending list can be found on horiguchi.net[3]. Removing duplicates (repositories that are trending multiple weeks) we end up with 90 (25 + 22 + 21 + 21) repositories.

4.1.1 Most stars

Looking at the most starred repositories brings several advantages.

Firstly, the star count is a good indicator of community interest and approval. A repository garners stars when developers find it noteworthy or valuable, signifying a level of trust and credibility within the programming community.

Secondly, these repositories often encapsulate best practices, innovative solutions, and robust coding standards. By selecting highly-starred repositories, we position ourselves to scrutinise codebases that exemplify excellence, offering a rich ground for assessing the efficacy of our energy measurement tool in real-world, high-quality software projects.

Moreover, the most-starred repositories are likely to attract diverse contributors and undergo continuous improvement. This dynamic nature ensures a varied landscape of commits and updates, providing a thorough testing ground for our energy measurement tool across different development scenarios and practices.

This selection of repositories may introduce bias as it favours more widely recognised projects. The average projects that might use the tool in the future will most likely be less popular. This bias might impact the representativeness of the sample. Thus, we utilise a second methodology of repository selection: the trending repositories tab.

4.1.2 Trending

While the most-starred repositories on GitHub serve as an indicator of popularity and community acclaim, it's not a good selection to represent the average GitHub project.

Most-starred repositories often belong to established, widely recognised projects, such as programming languages, frameworks, or tools. This concentration might skew the representation towards specific domains. In contrast, the GitHub trending list captures a more dynamic snapshot, including a broader range of projects, reflecting a balance between current trends and older technologies.

Trending repositories often signify active engagement from the developer community. Projects that make it to the trending list are likely to receive contributions, updates, and feedback, providing a more dynamic and current representation of ongoing development efforts. In turn, analysing these projects and finding results would signify that energy regression testing could benefit these kinds of active projects.

[3]`https://horaguchi.net/github-trending-history/`. Last retrieved 27th of March

The GitHub trending list provides a complementary perspective to the most-starred list. Combining insights from both sources could offer a more comprehensive and representative set of projects for our experiments.

4.1.3 Inclusion Criteria

Not all projects are fit for our experiments. To streamline our selection further, we implement a couple of inclusion criteria.

1. A repository must have a suite of unit tests. Otherwise, there would not be anything that the tool can test the energy usage of.

2. Repositories have to permit free access to the repository and/or creating forks. This is because, for this part of the experiments, we implement the tool in a fork to avoid interference with the main repository. Repositories that are archived are excluded as well

3. The repositories also have to have a long enough revision history to perform the analysis. If a repository is early development stage then there is not enough data from which the energy regressions can be determined. We set the bar to a minimum of 100 commits.

4. Repositories have to use GitHub-hosted runners. Some repositories use self-hosted runners to run their unit tests. As we do not have access to the servers that these runners are hosted on from a fork, we can not adequately test these repositories, nor do we have the necessary hardware information of these servers to make an accurate estimation of the energy consumption.

5. Tests need to be able to run offline. The variable nature of download speeds can lead to fluctuations in test durations, potentially impacting the reliability and reproducibility of energy consumption measurements.

6. Due to limitations with E-Compare's workflow implementation, the tool does not work with reusable workflows, as explained in section 3.4.

These inclusion criteria help us refine our dataset, ensuring that the selected projects are fit for our experiments, as well as align with the capabilities and requirements of E-Compare.

Other than these criteria, there may also be some projects that do not function with E-Compare for any unknown reason. While there must be a reason for failure, we could not find it based on the logs or code. These projects have to be removed as well as we can not obtain the necessary data.

Figure 4.1 shows the impact of each inclusion criterion on the experiments.

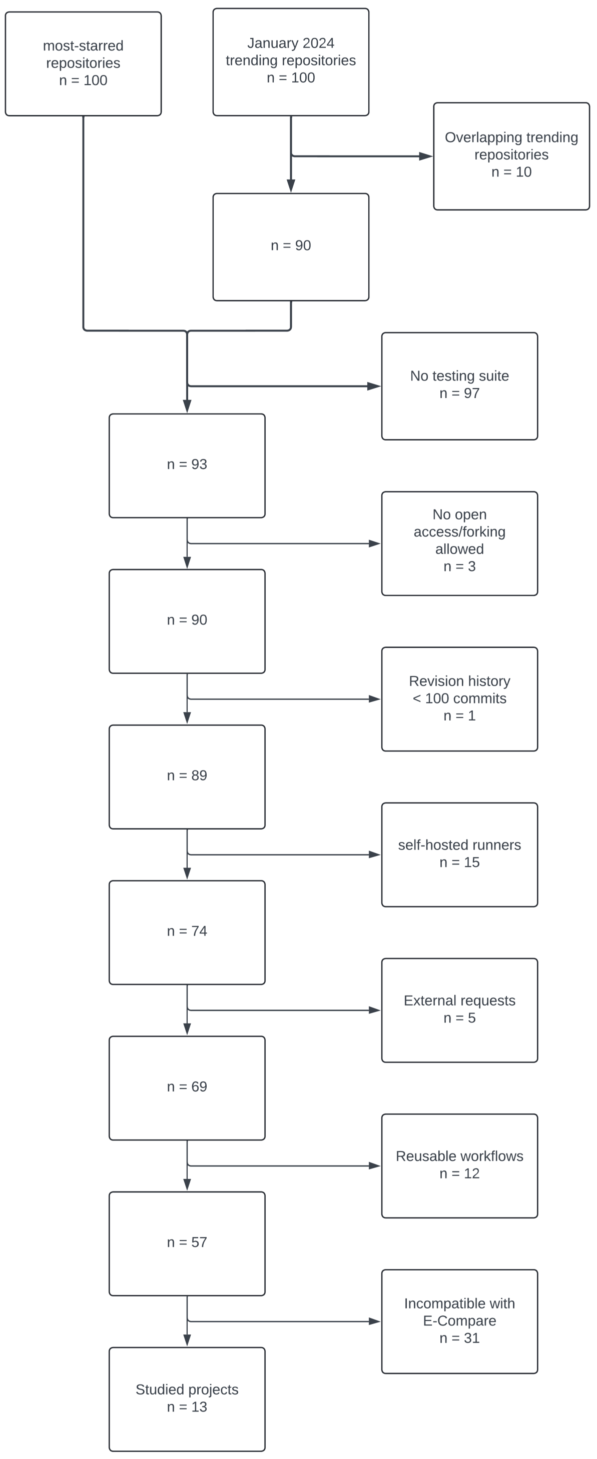

most-starred repositories n = 100
January 2024 trending repositories n = 100
Overlapping trending repositories n = 10
n = 90
No testing suite n = 97
n = 93
No open access/forking allowed n = 3
n = 90
Revision history < 100 commits n = 1
n = 89
self-hosted runners n = 15
n = 74
External requests n = 5
n = 69
Reusable workflows n = 12
n = 57
Incompatible with E-Compare n = 31
Studied projects n = 13

Figure 4.1: Flowchart showcasing the impact of the selection criteria by stating the number of projects filtered for each step.

Selection method	Repository name	Repository owner	Programming language	testing framework	Active developers
Most-starred	vue	vuejs	TypeScript	Vitest	364
	bootstrap	twbs	JS, HTML, SCSS, CSS	Karma	1387
	flutter	flutter	Dart	Native	1332
	stable-diffusion-webui	AUTOMATIC1111	Python	Pytest	551
	d3	d3	JavaScript	Mocha	132
	puppeteer	puppeteer	TypeScript	Mocha	487
	tailwindcss	tailwindlabs	HTML, Rust	Jest	283
	neovim	veovim	VIM script, C, Lua	CTest	283
Trending	ui	shadcn-ui	TypeScript, MDX	Vitest	138
	crewAI	joaomdmoura	Python	Pytest	28
	twenty	twentyhq	TypeScript	Storybook	170
	plate	udecode	TS, MDX, HTML	Jest	163
	hiddify-next	hiddify	Dart, Kotlin	Flutter	42

Table 4.1: List of selected repositories and their programming language. The selection method states the method through which the repository has been selected, those being either through the most-starred list or through the trending list. Commits analysed show the amount of successful and failed commits analysed.

4.1.4 Selection

The selected repositories are shown in Table 4.1. In total, 14 different repositories met the established criteria and managed to work with the tool. The selection contains thirteen different programming languages. Only languages that make up over 10% of the source code, according to GitHub, were included. Any language that GitHub lists will be counted as such, even if some of these languages are debatable, such as MDX. The most common language is TypeScript, which is used by seven of the fourteen repositories.

In the end, there are slightly more repositories from the most-starred list compared to the trending list. As expected, we find that the most starred repositories have more active developers, as stars and developers are tightly related to popularity. You could say that these repositories are generally more popular than the trending repositories.

The source code for each of these projects can be found by going to `https://github.com/{repository_owner}/{repository_name}`. The `repository_owner` and `repository_name` are the same as "Repository name" and "Repository owner" from Table 4.1

4.2 Implementation

In this chapter, we delve into the practical aspects of integrating E-Compare into software projects and workflows. We explore the historical integration process, detailing how the tool was applied to past commits in selected repositories.

Our aim is to provide insights into the implementation process and offer transparency regarding the tools and methodologies used in our experiments.

4.2.1 Historical integration

We integrated E-Compare into the selected repositories. To simulate the historical integration of the tool, we applied it to past commits in the selected repositories. This process involved applying the tool to past commits on the main branch, and analysing each commit's energy consumption based on the available historical context.

The tool is running on fifty commits for each project. Starting with the most recent commit on the main branch and going backwards. Some commits we can be sure of will not change the energy consumption, so those are skipped. This includes: README updates, CI/CD updates, docs updates, or new version releases. This filter is put in place because running these tests on old commits takes a long time, which can at least be slightly curbed by skipping redundant commits. We approach this filter conservatively though as minor changes can have a big impact. Despite not updating the code of the project directly, commits that upgrade the versions of imported packages are included. This is because the efficiency of packages can have a big impact on projects that use them. Some projects add flags to commits to signify whether to skip the CI process. This can be done as a label or as a specified phrase in the title of the commit, such as `[CI skip]`. When the project intends to skip the tests, we skip them as well.

Sometimes it happens that the tests fail for one reason or another. This is a rather common occurrence when introducing disruptive changes. We always try to run the tool on at least fifty commits, so when a commit does not pass the filter or fails the test we select the next commit down the line. In the case of *CrewAI*, we could not gather fifty commits as we went back to the very commit where the first tests were created.

The backtrack unfolds methodically: we spawn a new branch from each chosen to commit and adapt the CI workflow to include E-Compare in the unit testing process. This adaptation empowers E-Compare to run its diagnostics, generating results. Analysing fifty commits of a project this way takes about 2-3 hours. The entire process is visualised in Figure 4.2.

To ensure the reliability of tests with short durations, we employ a multiple-run approach, followed by averaging the results. This practice aims to enhance consistency and mitigate the impact of outliers. The number of test runs varies across projects, with the specific count documented in the source code within the

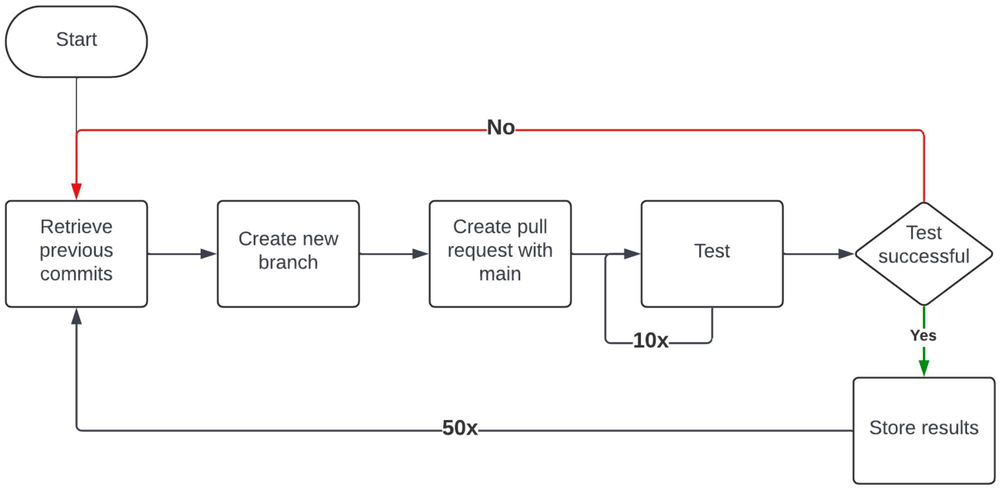


Figure 4.2: Flowchart of the backtracking process utilised for the experiments.

`projects` folder[4] on the E-Compare GitHub repository. The decision on the number of test runs is project-dependent. Our criterion is to run tests for a cumulative duration of approximately thirty minutes. This systematic approach helps maintain a balance between test accuracy and efficiency. For instance, if a test takes around three minutes, we conduct ten runs. This is not exact as test duration can vary wildly across a repository's history. Typically, older tests from the same repository take less time than more recent ones. In practice, we find that for many projects the tests are run ten times. For precise information on the number of test runs applied to each project, interested parties can refer to the source code available on the E-Compare GitHub repository. The `projects` folder contains project-specific details, providing transparency and reproducibility of our testing methodology.

Integrating the tool into a standard workflow only takes a few lines of code. Listing 4.1 shows one example of a workflow that was adapted for this experiment, specifically Vue.js. The trigger conditions are changed a bit to make sure the tool also collects energy consumption data on non-main branches. This is so that the tool can later draw comparisons when a pull request is made. At a minimum, three lines are added to the end of the workflow to integrate the tool. These are required for any project looking to implement E-Compare. Note that one line contains an extra indent for readability, though this is not required. Sometimes changes have to be made to other parts of the workflow. In this case, a trigger filter was removed so that the tool would run for any push and not just pushes made to the main branch. Other than that, the push and pull request triggers were already put in place.

[4]`https://github.com/koenhagen/E-Compare/tree/main/projects`. Retrieved April 23 2024

Listing 4.1: Example of an adapted YAML file to integrate E-Compare into the workflow. This specific YAML file is from Vue.js.

```yaml
name: 'ci'
on:
  push:
    branches:
      main
  pull_request:
    branches:
      main
jobs:
  unit-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Install pnpm
        uses: pnpm/action-setup@v2

      - name: Set node version to 18
        uses: actions/setup-node@v2
        with:
          node-version: 18
          cache: 'pnpm'

      - run: pnpm install

      - name: Run unit tests
        uses: koenhagen /measure-energy-action@v0.20
        with:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: pnpm run test:unit
          run: pnpm run test:unit
```

4.2.2 Tool integration

Sometimes the workflow files require additional changes to become compatible with E-Compare. As discussed in 3.2.1, sometimes changes to the workflow permissions are required. As E-Compare is run in a fork of the real repository we set the permissions to write-all to make it easy for ourselves.

In CI workflows, triggers determine when specific tasks are executed. Common triggers include actions like updating code or creating a new release. These triggers ensure that the workflow runs automatically after events in the development process. E-Compare requires specific triggers to function correctly. Specifically, the `pull _request` trigger and the `push` trigger are needed. These start the workflow automatically when a pull request or push is made. Some workflows from the experiments already use one or the other or neither. They may instead opt for triggering tests at a recurring interval using a cron job. These workflows have to be changed to utilise the `push` and `pull request` trigger instead. It is important to note that the `pull_request_target` trigger is insufficient as it runs E-Compare on the base branch instead of the head branch, meaning it will compare the base branch with itself, obviously resulting in inaccurate results.

E-Compare currently does not work with parallel processing. These processes have to be adapted to a sequential process, meaning all tests need to be run back to back instead of in parallel. Concurrent processes like sharding have been removed. Sharding typically involves splitting tasks or tests into smaller units and running them concurrently to decrease overall execution time. Removing this means that instead of executing tasks concurrently, each task is now linearly executed one after the other.

Some workflows utilise matrices to test multiple versions of different software. GitHub will run parallel versions of the testing suite in different environments. Matrices are used for instance to test different operating systems or different versions of the same operating system. They are also used to test different versions of languages or frameworks. Commonly, a matrix is used to run the tests on different versions of Python or Node.js. The parallel nature of matrices makes them incompatible with E-Compare. For our experiments, all historical commits are only tested on the latest version of any software or language. Also, only Ubuntu is tested.

The implementation is only done on code pushed to the main branch. Any other branches are considered out of scope.

The exact implementation for each project can be found in the `projects` folder of the replication package, available as a Github repository[5]. This includes the exact version of operating systems, languages and frameworks used for each project.

[5]`https://github.com/koenhagen/E-Compare/tree/main/projects`. Retrieved April 23rd 2024

4.3 Commit comparison

Now that the energy consumption has been measured for multiple versions of the software, they are visualised in some charts. This allows us to find energy consumption patterns associated with each commit or across multiple commits. Making the data visual also helps to find interesting data points not directly found when looking at the raw data. Also, you can look at the trend for some extra insights. Finding differences in energy consumption between two consecutive commits could signify an energy regression caused by a change to the source code.

4.3.1 Qualitative assessment

Then we proceed with a qualitative assessment of the commit tags and the underlying code changes to gain insights into the root cause of the energy regression. The goal is to find potential energy efficiency improvements that could have been realised if E-Compare had been in use. Ideally, the analysis might show that there are direct and unnecessary energy regressions included in the code that could easily be solved if developers were aware. Or the analysis might show that energy consumption slowly increases over commits and clear causes are not direction identifiable, or solvable. This qualitative analysis encompasses diverse commit categories, such as bug fixes or feature additions. Notably, we consider scenarios where significant energy spikes are identified. For essential commits, such as security patches or high-priority features, if the energy report reveals substantial spikes, it may indicate potential limitations in its utility for critical updates. On the flip side, if E-Compare captures such spikes for new low-priority features or nice-to-haves, it would suggest that E-Compare is effective in flagging energy implications for non-essential enhancements.

We consider analysing external packages and other imported code to be out of scope for this experiment.

4.3.2 Defining energy regressions

Before heading into the results, the question is asked as to what would be considered a spike in energy consumption. Defining energy regressions involves setting a threshold to identify significant fluctuations in energy consumption. In our case, we chose a threshold of 9% based on the maximum variability observed when running tests with E-Compare on a server, as stated in research by green coding[38]. You may notice that this number is different from the 5% at which E-Compare highlights a number as green or red. It is important to note that this value is somewhat arbitrary, as it can be assumed energy variability tends to decrease as tests are repeated multiple times, which is done in our experiments, and because other reasons for overhead exist. However, we believe this to be the best method for defining energy regressions. The 5% threshold used by E-Compare was chosen for its simplicity and intuitiveness for users, while the 9% threshold is more robust for experimental purposes.

We recognise that some regressions might not reach the threshold of 9%, while still having a significant impact on energy consumption for that project. So, other than the main threshold of nine percent we define a lesser threshold at half that (4,5%).

Chapter 5

Results and Discussion

All results can be found in Figure 5.1 and Figure 5.2. This section first gives generalised results and then goes more in-depth into the individual projects, looking at the found regressions and their underlying reason, if there are any. Higher quality versions of the plots can be found in Appendix A.

5.1 Overall result

A total of 13 projects were analysed. This is less than 7% of the initial 190 selected projects before applying our selection criteria. Among these projects, 759 commits were analysed. After removing broken commits (commits that did not successfully run the testing suite), 594 led to functional results. Among these, 84 were classified as energy regressions according to our definition, with 18 being major and 66 minor energy regressions.

The source code changes for specific commits shown in the charts can be looked up on GitHub by going to the following link: `https://github.com/[repository_owner]/[repository_name]/commit/[commit_hash]`. The repository name and repository owner can be found in Table 4.1. The commit hash can be found in the individual charts of Figure 5.1 and Figure 5.2.

5.1.1 Success criteria

E-Compare can operate fully automatically by wrapping the testing suite of any software application. A developer only has to adapt their CI workflow and the tool reports the energy consumption and regressions without any further manual work required from the developer. This solution satisfies the second criterion of the experiments specifying that the energy regression testing can be done in an automated manner.

The other criterion specified that the tool could find energy regressions in software projects. E-Compare managed to show energy regressions for twelve of the thirteen projects that were analysed. By looking at the source code of some of the changes in the analysis, we found that a few energy regressions could be logically traced back to changes in the code. These findings validate that the tool can find energy regressions, satisfying the first criterion. Importantly, these energy regressions were not addressed by the developer in later commits. If this is because the developer was not aware of any inefficiencies in the code, it would mean that E-Compare could potentially influence decision-making when developers are implementing new updates to their projects.

While we successfully implemented E-Compare into thirteen different projects, its success can be questioned. When you compare this number to the total number of projects that fit the inclusion criteria, we can say that this is a low amount. In 31 other cases, the tool was intended to be applied but failed for inexplicable reasons. If some projects are not able to utilise energy regression testing, the application of such tools remains limited. However, even though the reason for these failures is unknown, we do not believe that the reason is related to the fundamental concept of energy regression testing. Instead, more likely is that the reason is due to the implementation of the tool or because of issues with the underlying project.

Energy regressions that have a clear cause and effect are a rare find. One project from the results does not show any energy regressions, while others show many where the underlying cause is not clear. It is clear that some projects benefit from using an energy regression testing tool, but not all of them.

5.1.2 Causes of energy regressions

Many regressions were due to changes in the tests, which makes sense, as increasing the number of tests would likely lead to more computation and thus higher energy consumption. This then is reflected in the results from the tool. However, these findings are not particularly valuable, as more tests do not indicate that the code contains inefficiencies that should be fixed. The energy regressions that seem to be caused by actual changes to the source code can be considered valuable outcomes and are the primary reason a developer would want to use an energy regression testing tool. Among all analysed commits, only a few of them are we believe that the results would come as a surprise to the developer.

Not all energy regressions flagged by the tool have a clear root cause. This might be because the commit contains a lot of code changes, making pinpointing the exact cause very difficult. The reason for the energy regression could be any of the hundreds of lines changed in these commits. On the flip side, some smaller commits were also flagged without a logical explanation. Based on our limited analysis of the code change, we could not find a clear root cause in these small commits. However, It is hard to predict how a change might impact energy consumption as some of the glanced-over lines might have large consequences.

Energy testing is not deterministic, with various aspects influencing energy consumption, such as server overhead, testing framework overhead, or general inconsistencies in tests. These other aspects may significantly interfere with the measurements that energy regressions are flagged that may not be there, or vice versa. We must be aware of these factors and critically evaluate the results.

Some projects showed clearer results than others. Regular fluctuations significantly impacted some projects, with some even showing energy regressions for the majority of commits. For these projects, the usability of an energy regression testing tool might be questionable. Developers need to determine if such tools suit their projects. We observed more frequent and larger fluctuations in projects using more complex testing formats.

5.1.3 Outliers

The data often showed spikes, which we define as commits where the energy regression significantly changed and then changed right back to the original value in the following commit. These spikes are always analysed thoroughly, as one outlier could very well be caused by some unforeseen influence. The conditions needed for a spike to be real are very niche. The changes to the source code have to logically cause an energy regression, and the commit after where the energy consumption levels drop back down to their initial value has to have a logical explanation for such a drop. You would expect a revert of the previous commit or a code change that undoes or fixes the potential regression. In most cases, these criteria are not met.

When an energy regression is followed by multiple commits around the same value, it gives much more credibility to that commit. It suggests that the consecutive commits do not or only minorly impact the energy consumption and the energy consumption stays around the new value compared to the commits before the energy regression.

5.1.4 Failing tests

For some commits, the testing suite results in failure. This may be due to the underlying tests or source code, or this could be due to some conflict with the tool. In ten cases, these broken commits are followed by an energy regression. In cases like these, the cause of the regression can be in any of the commits including the commit showing the regression. While this makes it more difficult to find the cause of the regression, it can also be seen as one big commit that has all commits combined. This also reflects the typical use of E-Compare where you would check the energy consumption of an incoming merge, which is typically a combination of multiple commits.

5.1.5 Project archetypes

Some archetypes of projects can benefit from energy regression testing more than others. When we compare the results from different analysed projects we find a couple of differences. Projects that showed more signs of immaturity resulted in less relevant results for the experiments. These signs are larger commits without clear labels or descriptions, more broken commits due to code changes, and incomplete test suites. Larger commits and more broken commits mean larger code changes, resulting in more difficulty in tracing an energy regression back to individual line changes. Incomplete testing suites can result in some relevant energy regressions being overlooked.

Energy regression testing works better for relatively stable projects. This means that there are no major differences in energy consumption between runs of the tests and source code. We see larger fluctuations in projects with larger testing suites. This could be related to tests causing more overhead, or because more tests result in more possibilities for differences. We also find that the one AI project analysed also causes large fluctuations. AI is typically very inconsistent and non-deterministic, thus it makes sense that its tests also adopt the same properties.

Ultimately, it is hard to draw any conclusions on project archetypes, because the initial results are only comprised of a limited number of projects.

5.2 Per project

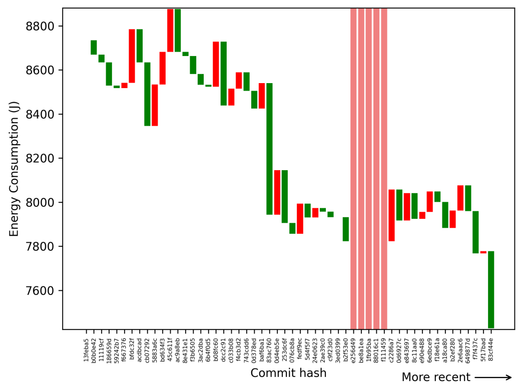
All results per project are shown in Figure 5.1. This section goes over each project one by one providing basic background information and showcasing insights that the tool provided. The aim is to figure out if the energy regressions found are caused by changes to the source code and not because of other reasons. In other words, we try to find out if the energy regressions are actually regressions and not just fluctuations in the measurement. The hope is for these regressions to potentially be fixable if the developer would be aware of their existence. Proving that the regressions are caused by source code changes will be difficult, but tracing down likely culprits in the source code helps to build a stronger case. The order here is the same as in Figure 5.1.

The X-axes in Figure 5.1 and Figure 5.2 show the hashes of each individual commits. The order is the same as the historical timeline of the main branch. Older commits are shown on the left. The Y-axes show the energy consumption of the tests for each commit. The energy consumption is measured in joules. Important to note is that the Y-axis does not start at zero. This is done to better show the regressions, but a negative side effect is that it might cause wrong takeaways when the graph is not read correctly. Broken commits are shown with coral red stripes going vertically.

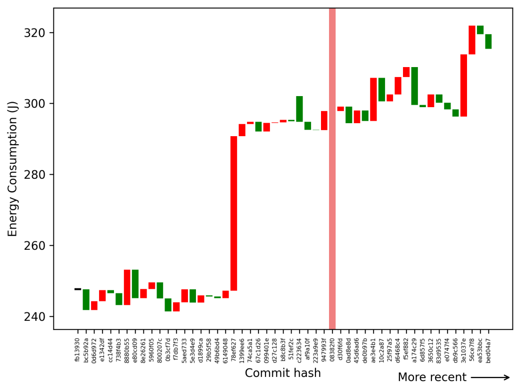
A table containing a summary of all findings in numbers can be found in Table 5.1. It lists the number of regressions per project, the number of analysed commits, and the analysis period.

(a) Flutter Flutter is a modern SDK developed primarily for mobile development by Google. Flutter uses the Dart language, which is also developed by Google. Flutter developed its testing framework natively.

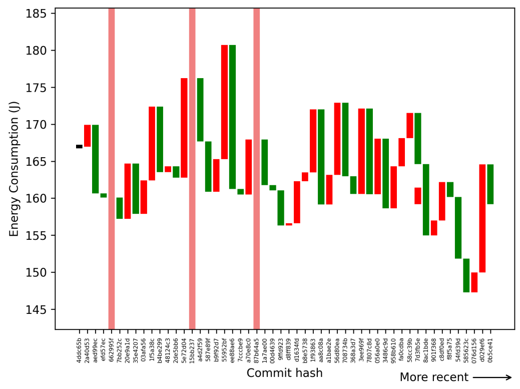
There are no direct major energy regressions found for Flutter. Though looking at the chart, some changes in energy consumption across commits can be seen. Important to note here is that the X-axis is relatively large. Based on the average time taken for the unit tests to run and the energy consumption for



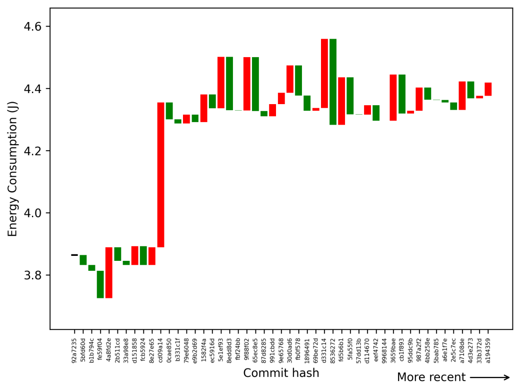
(a) Flutter



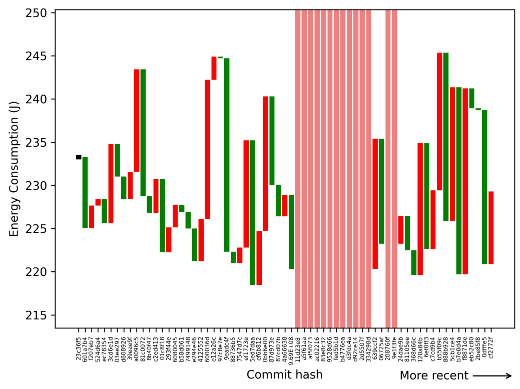
(b) Vue



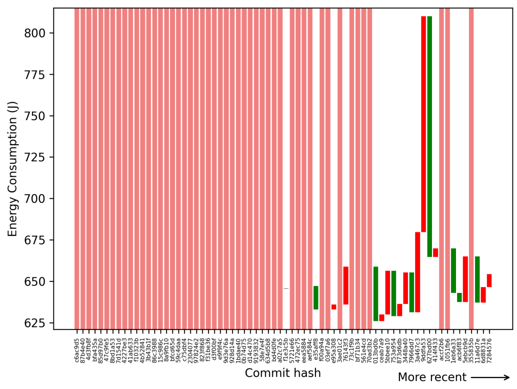
(c) Bootstrap



(d) d3

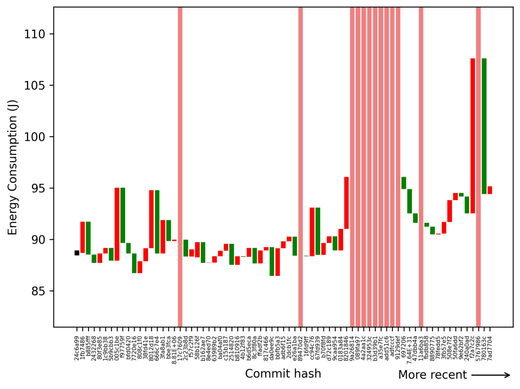


(e) Stable Diffusion

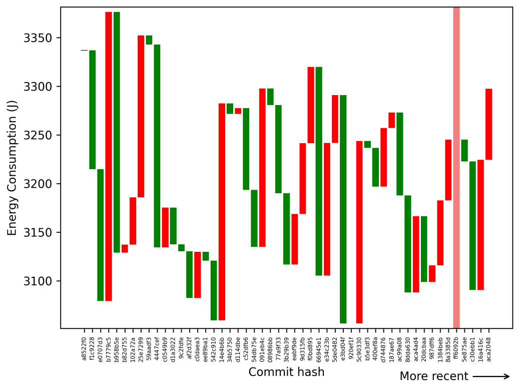


(f) Puppeteer

Figure 5.1: Results from the most starred repositories. The X-axis shows the commit hashes going from new to old. The Y-axis shows the energy consumption in joules. Bars show the difference in energy consumption between the current and previously measured commit. Red means the energy consumption increased and green means it decreased.



(g) Tailwind CSS



(h) Neovim

Figure 5.1: Results from the most starred repositories. (cont.)

these commits, the relative changes are small. On average, running all unit tests took around two hours and thirteen minutes. We run the entire testing suite three times and take the averages.

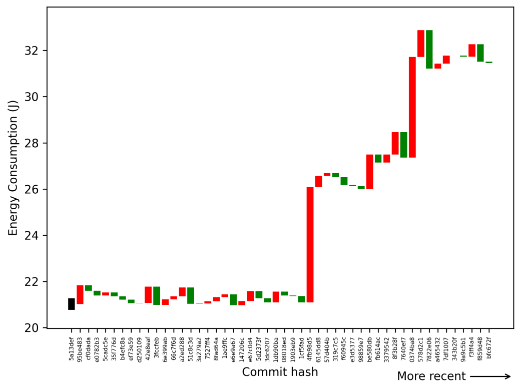
While there are no major energy regressions, there is one minor energy regression for commit *83ac760*, which dropped 596 Joules compared to the previous commit. Based on the chart, we see that the drop in energy consumption remains rather consistent for the following commits. What is most curious is that this commit exclusively adds a new function to the source code. The commit inserts 417 lines while removing zero. This is very atypical for a negative energy regression. We can not deduce what would be the reason for this drop, looking at the source code, though developers more versed with the project might.

Four back-to-back commits broke the unit testing suite. Presumably, the first commit breaks the suite. This commit updates many of the underlying packages. This commit gets reverted in the first commit afterwards that has a functioning testing suite.

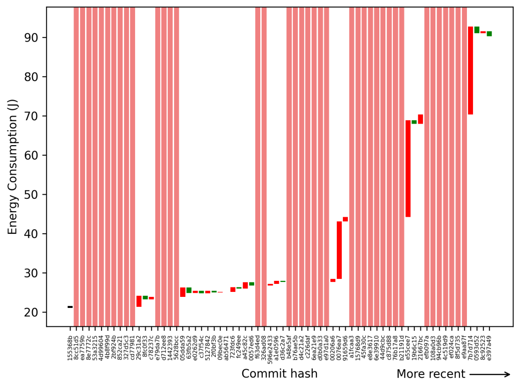
(b) Vue.js Vue is a JavaScript framework intended for building web user interfaces. The framework prides itself on being accessible and performant.

The results show that the energy consumption of the unit tests is very consistent. Changes in energy consumption typically hold for the commits following the change.

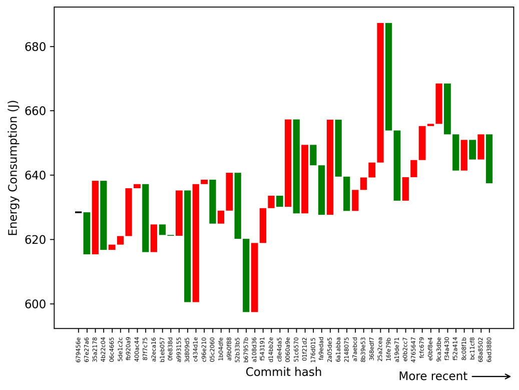
Vue contains one major energy regression in commit *78ef627* where the energy consumption rises from 245 joules to 290 joules. This commit does two things: it updates the testing framework *vitest* to a higher version, and it changes some server-side rendering functions to be able to run with Node 18 and up. Both of these could influence the energy consumption of the testing suite, though they



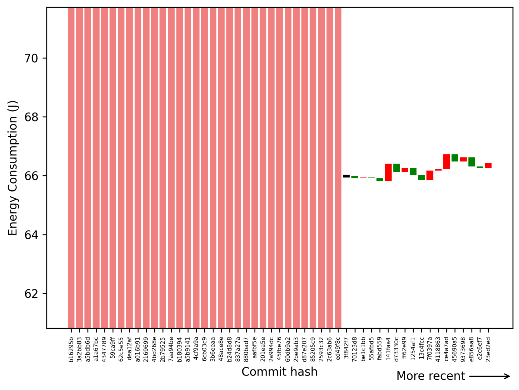
(i) ShadowCN UI



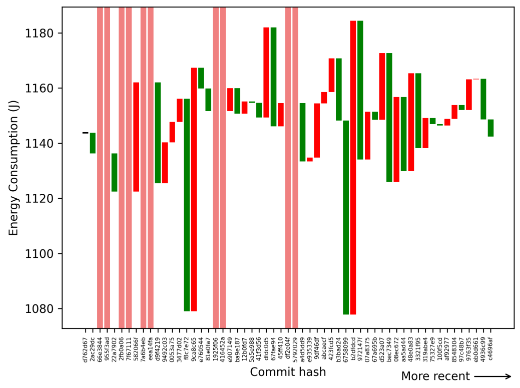
(j) CrewAI



(k) Twenty



(l) Hiddify



(m) Plate

Figure 5.2: Results from GitHub's trending repositories list. The X-axis shows the commit hashes going from new to old. The Y-axis shows the energy consumption in joules. Bars show the difference in energy consumption between the current and previously measured commit. Red means the energy consumption increased and green means it decreased.

would probably not show changes in an actual deployed environment. The new testing framework could introduce more overhead when running the tests. This would not be relevant for the developer as this increase in energy consumption would then only show during testing and not in production. The performance of the testing framework has no bearing on the actual performance. The fixes to some of the functions will increase the overall accuracy of E-Compare's results, which would be useful for the result of all commits going forward. However, it would not impact the real-life energy consumption of Vue, as the testability of a function does not matter for its use. The functions are still deployed and functioning in production.

Other than the major energy regression, there is another minor regression. Commit *3e1037e* results in a smaller increase in energy consumption. Similar to *78ef627*, it is also caused by an update of the testing framework. This time, we can be certain that this is the cause as this is the only change in the commit, consisting of a singular line edit. Supposedly, a change in *vitest* makes it so it takes more computational power to run.

There is one broken commit, which is followed by a fix right away.

(c) Bootstrap Bootstrap is another JavaScript framework that focuses on dynamic front-end usability across different screen sizes. It uses *Karma* as its testing environment.

Compared to Flutter and Vue, Bootstrap fluctuates a lot more between commits. This makes sense when we look at the testing suite of Bootstrap. In the source code, we find that Bootstrap runs integration tests together with their unit tests. Thus, the results from E-Compare show a combination of the two. Integration tests are typically less steady compared to unit tests due to their non-deterministic behaviour and complexity, which could explain the fluctuations.

There are three major regressions within these fifty commits. The first two are back to back, going up and down right after, causing a spike in the chart. As these commits only add minor changes to the code, there is a large chance that this spike is not a result of changes to the code, but instead comes from something else. This assumption is backed by the fluctuating nature of the results and the fact that the commit after the major regression goes back down again to around the same values as previously recorded. The third major regression is found in *d029ef6*. This is another small commit that updates the sass-true package[1] to a higher version. This package is a unit testing tool for SASS code, which is used by Bootstrap. This update potentially caused more overhead from the tool, causing an increase in energy consumption. Though, this is far from certainty. Looking at the graph, it seems more like the commit went back up to a typical level after a series of commits with a lower energy

[1]`https://www.npmjs.com/package/sass-true`. Retrieved March 10th 2024.

consumption. These previous commits mostly consist of version upgrades of packages, so it is hard to say what causes the gradual drop in energy consumption unless we look into the source code of the used packages. Combining this with the fluctuations makes it hard to draw any conclusions as to what the reason for the regression would be.

There are also many more minor regressions found in the results. In total nineteen commits show sufficient change to be classified as a regression, which would account for 38% of total commits for Bootstrap. It is highly unlikely that all of these commits significantly influence the energy consumption of the source code. Especially as fourteen of them are version bumps of package dependencies. Underlying packages may have an impact on the energy consumption of codebases that use them, but typically they are only used partly and in a small part of the source code. These version bumps are often only minor updates (e.g. from x.x.0 to x.x.1).

(d) D3 D3 stands for Data-Driven Documents. It is a JavaScript library used for data visualisations. D3 uses the Mocha testing framework[2] for its unit testing.

The first thing to note is that the test duration is short. Running the tests in their entirety takes about a second with an average of 4.3 joules. It would seem that the unit tests written on the main repository are not very comprehensive. D3 splits up their widgets and functions across multiple repositories. Each of these repositories hosts its own set of tests. Running the tests from the main repository does not run these split-up tests, resulting in low code coverage. Because of this, how useful the results are for D3 is questionable. The results look very stable averaging around 3.8, then later 4.4 joules.

The one major regression at commit *cd09a14* adds extra unit tests to the suite. This makes sense with the results because, as stated before, more tests run logically means more energy consumption. Two minor regressions appear back to back at commit *d331c14* and *8536272*, but they seem coincidental. When the values are this small, a 0.2-joule change would already indicate a minor regression. Because of these small values, D3 would probably not benefit from using D3.

(e) Stable Diffusion Stable Diffusion is a text-to-image Artificial Intelligence model. This specific repository is for the web user interface. Stable Diffusion uses *pytest* as its testing framework.

Stable Diffusion fluctuates a lot between commits, resulting in a lot of energy regressions. This would make sense for an AI-based project, as AI is highly inconsistent and non-deterministic. Looking at the written tests, they perform a variety of text-to-image tests, which vary in length between tests.

Despite the high fluctuation, there are only two major regressions. Both concern a drop in energy consumption. The first, commit *9eadc4f*, makes changes

[2]`https://mochajs.org/`. Retrieved May 1st 2024.

to the automated cropping functionality. These changes could lead to more efficiency, potentially dropping energy consumption levels. The three commits before and after the regression remained stable, which would support this hypothesis. However, it is ultimately hard to say due to the inconsistent nature of the Stable Diffusion's test results. The second major regression, commit *b7e0d4a*, is a smaller commit only changing a couple of lines and the results for the commit right after it jumps back to its previous level. Based on these things, combined with the high fluctuations, the cause of the drop appears to be random.

There are a lot more minor regressions, often appearing back to back as the commits go back and forth between energy values. In total, there are sixteen minor regressions. It is hard to find an explanation for the regression based on the changes to the source code. Even if some change would potentially impact energy consumption, the change right after would go back to the previous value without any logical explanation in the source code. I strongly suspect that the energy consumption is not influenced by the commits for these minor regressions.

(f) Puppeteer Puppeteer is a Node.js library that allows control of your browser through its API. It is mainly used as a web scraping tool. Puppeteer uses the *Mocha* testing framework.

Analysis of only 27 commits was possible. This is not the typical fifty that are conducted in these experiments. The reason for this is that all commits broke after the 41st. After the seventieth commit, we concluded that there was no longer any reason to make another commit because the likelihood of it working was so slim. As a result, the tests for this specific project were discontinued. The reason for these breaks is difficult to track down. Logs do not provide any extra information and look the same as logs for tests that succeeded.

Fluctuations of around thirty joules are a common occurrence for Puppeteer. Most of these are not classified as energy regressions, but they come close to it. This suggests that the fluctuations are naturally high and not caused by source code changes.

There is one major spike that causes two energy regressions. This spike goes way higher than any other point in the graph. The commit *5bbee10* contains an update to *Angular*, which is the primary JavaScript framework that Puppeteer uses. As Puppeteer is heavily dependent on Angular, an update could be the source of the energy regression. The problem here is that the energy consumption comes back down for the commit right after (753a954). This commit, as well as any commit after, are unrelated to this, making the instant drop rather odd.

(g) Tailwind Tailwind CSS is a CSS framework for front-end development. It uses class names to edit CSS properties directly from your HTML files. Tailwind uses *Jest* for unit testing with *Vite.*

Tailwind contains two major regressions that can be seen in the graph in Figure 5.1. Near the end, the energy consumption shoots up and goes back down two commits later. The first commit *f2a7c2c* consistently took around 68 seconds for each run of the test suite, while the commit before took 60 seconds on average. There are no changes to the tests and the same amount of tests succeed for both commits. The increase would not make sense as a fluctuation because of the consistency between runs and the extreme difference between it and other commits. The most logical explanation for the spike is a change in the source code. The commit is rather short, only editing 43 lines. It adds “Improve glob handling for folders with ‘(‘, ‘)‘, ‘[‘ or ‘]‘ in the file path” by adding a new function called “normalizePath” shown in Listing 5.1. The function does not contain any loops, which would typically increase the duration. There is no clear spot in the code that would indicate inefficiencies. Some of the used functions such as the .split function may be computationally intensive. Potentially, the function is called so often that a minor increase in energy consumption would already be significant. Ultimately, the cause of the regression is not easily spotted, which makes this a good case for E-Compare.

Listing 5.1: Additions made in commit `f2a7c2c` for `tailwindlabs/tailwindcss`

```
function normalizePath(path) {
  if (typeof path !== 'string') {
    throw new TypeError('expected␣path␣to␣be␣a␣string')
  }

  if (path === '\\' || path === '/') return '/'

  var len = path.length
  if (len <= 1) return path

  var prefix = ''
  if (len > 4 && path[3] === '\\') {
    var ch = path[2]
    if ((ch === '?' || ch === '.')
        && path.slice(0, 2) === '\\\\') {
      path = path.slice(2)
      prefix = '//'
    }
  }

  let segs = path.split(/[/\\]+(?![\(\)\[\]])/)
  return prefix + segs.join('/')
}
```

The other major regression takes place two commits after, with the commit in between being a broken commit. This second major regression at commit *780163c* is a bigger commit that encompasses multiple changes. The commit adds a new feature, updates the testing framework and fixes the regex in some tests. Any of these could influence energy consumption. The regression could also be because of the previous commit that was broken. As its values could not be measured, it is unknown if the regression already appeared in that commit. That commit adds a new feature and fixes some failing tests.

Other than two major regressions, there are also seven minor regressions for Tailwind. As seen in the graph, there are three upward spikes other than the one previously mentioned. For each of these commits (*005c1be, 8012d18, cc94c76*) there is no logical place in the source code that would warrant such a spike. None of the behaviours we look for to check if a spike is real are present, thus we assume that these minor regressions are not caused by source code changes. The last minor regression does not go back down significantly afterwards. This commit *8201846* adds more opacity options for Tailwind users. This is a change that could lead to an energy regression. Depending on how they are done, opacity calculation may be quite computationally intensive.

The is mainly one big gap of broken commits. This is caused by an update to the tests. Eight commits later another update to the tests fixes the workflow allowing E-Compare to work again.

(h) Neovim Neovim is a fork of *Vim* that focuses on bringing a new and improved version of the old text editor. Neovim adds dozens of new features and improved extensibility and usability. It uses *CMake*[3] and *CTest* to run its tests.

Neovim has a very large testing suite, reaching around 3200 joules just to test all of its functionality. Looking at the chart, it is not a stable suite with differences over 100 joules being a common occurrence.

However, because of the lengthy duration of the tests, there are no major regressions. A major regression requires a change in energy consumption of at least 290 joules, which did not occur.

There were however many minor regressions. The question again is if these were caused by changes to the source code or natural fluctuations. There is no significant trend to be found in the data. Plotting a trend line shows just about a straight line, suggesting that the commits have no impact on energy consumption. Anytime a regression occurs, it goes back in a few commits. The amount of minor regressions combined with the insignificance of some commits also brings up questions. Of course, you can not be sure a commit does not cause a regression, but some of these commits are very unlikely. For instance, one of the commits that caused a regression is an update to the localisation

[3]`https://cmake.org/`. Retrieved 2 May 2024.

of Japanese. Based on this, we assume that these regressions occur due to natural fluctuations of the tests.

(i) ShadowCN UI ShadowCN is a customizable component library that developers can use to copy and paste into their apps. ShadowCN uses the *Vitest* testing framework for typescript. ShadowCN is the first project from the trending repositories list. It is a younger project compared to the previous projects from the most starred list.

The unit tests for ShadowCN are short, totalling from 20 to 32 joules per commit. There is not much fluctuation between commits, where the first 28 commits all fall within a narrow range of energy consumption, without any energy regressions.

There are two major energy regressions for ShadowCN. Both commits are very large, with commit *0374ba8* reaching 12.098 line changes. Among the changed lines are also some additions to the test files. Some tests were changed and some were added. There were also some new packages imported, some changed functions and some new functions added. When a commit is this large, it is hard to find the root of the rise in energy consumption. It could be one specific change or a combination of multiple changes done in the commit. Commit *4fb98d5* shows the other major regression. It is a shorter commit compared to the other major regression, adding 477 lines. This commit adds support to the use of ShadowCN components with the *Tailwind* framework, which could cause a rise in energy consumption. It also adds more tests, which could be another reason.

There are also two minor regressions found in the results for ShadowCN. The first *be580db* is also a sizeable commit adding multiple functions and additional unit tests that test these new functions. A rise in energy consumption is completely logical for that. The other minor energy regression is a drop in energy consumption that is less easy to explain based on the source code. This commit *7822e06* is in between similarly valued commits. It is slightly below its neighbours, while the previous commit is slightly above its neighbours, resulting in a minor regression. So we presume that the regression is by chance and not because of changes to the source code.

(j) CrewAI CrewAI is a framework for creating AI agents based on specific identity specifications. It allows the agents to work together and can be used for role-playing. CrewAI uses *pytest* for testing.

CrewAI is a young repository, even though it still fits the revision history inclusion criteria. It is mostly created by one person. This is reflected in the chart with many broken commits and steep increases in energy consumption. Other than some clear energy regressions, the energy consumption remains stable across commits. In total, we obtained data for only 28 commits, which is 22 less than the goal of 50 commits. The reason for this is that we went back with the backtracking process to the very first commit where unit tests were created. Theoretically, we could have gone back further by applying the unit tests to older commits, but this seems illogical as such a scenario would never occur with regular use of the tool. Also, it could create inexplicable behaviour that would result in unreliable data. This is one of the two repositories where we did not manage to obtain enough data points to satisfy our goal.

There are five major regressions found among the 28 working commits. Often these regressions are preceded by several broken commits. Any of the broken commits and the commit with the energy regression could be the cause of the energy regression. Some of the regressions are preceded by ten broken commits, making it very hard to deduce what the origin of the regression is. There are two commits succeeded by seven broken commits and two succeeded by ten broken commits. For these four, we have to skip the analysis because of the difficulty of establishing the origin of the energy regression. The other major regression *0076ea7* has one broken commit before it. The former adds extra unit tests, which would coincide with the increase in energy consumption. The latter refactors the source code by creating a new class.

There are two minor regressions for CrewAI, both not preceded by broken commits. Commit *63fb5a2* mostly updates documentation and makes a couple of line changes, including a filter for AI agents. Such a filter could have an impact on the energy regression. Commit *a45c82c* mainly adds one extra unit test, which would explain the small increase in energy consumption.

The broken commits can be caused by the fact that this is an immature repository that gets frequent large updates. These updates could completely break the testing suite. When the tests are broken, it takes a while for a developer to fix them.

(k) Twenty Twenty is an open-source customer relationship management platform. It is designed to help businesses manage and analyse customer interactions, data, and relationships. Twenty used *storybook*'s testing environment[4].

The chart of this project shows a clear trend upward, with later commits structurally requiring more energy than earlier commits. However, it is difficult to pinpoint one specific commit as the culprit. It might be that multiple commits contributed a small amount to the increase, or one commit got masked by the natural fluctuations.

There are no major energy regressions found for Twenty. There are five commits with two back-to-back minor energy regressions. Keep in mind that these are only minor regressions. The first regression is a drop in energy consumption, despite adding many new lines and functions. Other than new functions it also has multiple fixes, including fixes to the tests. It is difficult to pinpoint exactly what would cause a drop in energy regression if there even is a reason

[4]`https://storybook.js.org/docs/writing-tests`. Retrieved May 16th 2024.

based on the code. The commit right after also shows an energy regression, but upwards. This commit makes a lot of changes to the testing suite, "making fixes and enhances". This would align with the increase in energy consumption. Commit *51c6570* shows another downward regression. The commit "removed the boxes around fields on shows and side panel". Having fewer objects to load logically takes less energy, so this change makes sense with our results. The fourth regression adds more dependencies and the fifth regression makes changes to the imports. It is hard to judge the impact of such changes on the energy regression as we consider analysing external code to be out of scope.

What is interesting is that looking at the chart there are two downward spikes can be seen. The first one shows a minor regression both ways, but the second one is not flagged as a minor regression. This is despite the second downward spike going lower to 597 joules, compared to 600 joules for the first spike. This shows how influential the regular fluctuations or gradual changes across multiple commits can be for the results.

(l) **Hiddify** Hiddify is a multi-purpose proxy tool. It provides a secure and private way of accessing free internet. Hiddify uses Flutter under the hood.

Hiddify contains no major or minor regressions. In this case, it is important to note that the Y-axis is only stretching a narrow range.

Only eighteen commits were able to be analysed. This is different from the usual fifty done for these experiments. After the eighteenth commit, every single commit broke. By the fiftieth commit, we figured that the chance another commit was going to work was so low it was not worth it to continue. Thus, the experiments for this particular project were stopped, much like what happened with Puppeteer. The earliest commit before the breaks was an update of the Flutter framework. Presumably, this Flutter update changed something to allow E-Compare to function with its tests.

(m) **Plate** Plate is a rich-text editor that can be implemented in your projects. Plate uses *Jest* as its testing framework.

The energy consumption of Plate seems to be stable, with most commits hovering between 1122 and 1184 joules in the majority of cases. There are no major energy regressions in the fifty commits analysed for Plate. What is notable is that the test results are cached after running them for the first time. This effectively means that the tests are only run once and consecutive runs are negligible. From the second run onward, the runs only take a few microseconds, not running any of the tests. This result is that the usual method of removing variability does not work as intended.

There are two relatively large dips in energy consumption, resulting in four minor energy regressions, but no major regressions. These might be the result of the increased chance of variability. Or they might be caused by source code changes. However, we believe the latter is not the case due to the nature of the changes: First of all, they only contain additions, which is weird for drops. Second of all, consecutive commit's code changes are not related. If the energy consumption jumps back, you would expect an edit to the previously chanced code.

Plot number	Project	Commits analysed	Successful commits	Major regressions	Minor regressions	Analysis period
(a)	Vue	51	50	1	1	Oct 11, 2022 to Dec 31, 2023
(b)	Bootstrap	53	50	3	16	Oct 31, 2023 to Jan 9, 2024
(c)	Flutter	54	50	0	1	Dec 27, 2023 to Jan 9, 2024
(d)	Stable Diffusion	65	50	2	15	Nov 27, 2023 to Dec 16, 2023
(e)	D3	50	50	1	2	Jun 11, 2023 to Jan 29, 2024
(f)	Puppeteer	71	27	2	2	Feb 6, 2024 to Mar 4, 2024
(g)	Tailwind	63	50	2	7	Apr 19, 2023 to Mar 7, 2024
(h)	Neovim	51	50	0	9	Mar 15, 2024 to Mar 19, 2024
(i)	ShadowCN	50	50	2	2	Oct 21, 2023 to Feb 4, 2024
(j)	CrewAI	70	29	5	2	Jan 10, 2024 to Feb 22, 2024
(k)	Twenty	50	50	0	5	Feb 22, 2024 to Feb 29, 2024
(l)	Plate	57	50	0	4	Feb 7, 2024 to Mar 1, 2024
(m)	Hiddify	50	18	0	0	Feb 21, 2024 to Mar 10, 2024
Total		735	574	18	66	

Table 5.1: Table containing a list of the selected projects. Commits analysed show the number of commits analysed, while successful commits only shows the number of commits that gave results. The analysis period shows the date of the first and last commit that were analysed.

5.3 Discussion

In this section, we discuss the implications of our key findings. We discuss the impact of various commit types, the robustness of our tool, and the difficulties with identifying energy regressions.

5.3.1 Main findings

In the experiments, a large amount of commits were analysed. This allows us to dive deeply into the underlying code and find correlations between types of commits and energy consumption. In the end, far and away most energy regressions came from commits that were focused on the tests. Other than that commits that updated packages also often impacted the energy consumption. This mostly happened when the updated package was related to the tests, like the testing framework. The potential impact of package updates is interesting as these commits are sometimes flagged as *chore*, for which tests are then subsequently skipped in CI. Other than tests and package updates, there were no other patterns found in other commit types. This does not necessarily mean that these do not impact energy consumption, but instead, it could be that they are not as common. Commits that target performance or implement refactors, likely have an impact on energy consumption, but did not occur frequently among the analysed commits. We also saw that tests were often written in a different commit than the features they are meant to accompany. In these cases, the initial commit that adds the new features would impact the energy consumption, but this is only reflected in the data once the new tests are written.

Although the number of commits analysed was high, the number of projects was low. Almost all analysed projects provided some new findings. While we spend a long time gathering all the data and analysing all the projects, analysing more projects will provide even more interesting findings. With the current study size, generalisations about different archetypes of projects were difficult to make. We can not reasonably discuss the impact of certain testing frameworks, for instance, when that framework is only used by one or two projects.

In general, we found only a few concrete examples of energy regression. Even with a sizeable sample size, energy regressions outside testing and package updates remained rare. Most relevant energy regressions came from the older, more established projects. This would suggest that more mature repositories benefit more from implementing an energy regression testing tool. On one hand, they more often have more comprehensive code coverage. On the other hand, they are often larger codebases, which would require a larger code inefficiency before it shows up in the results.

Bigger commits are much harder to dissect compared to small ones. When only a few lines are changed, it is easy for the developer to wrap their head around. When the commit has a lot of line changes, that can be tricky. Even if you have a hunch of where the energy regression comes from, it remains speculative. This happened often throughout our experiments. The result is that even though we can be reasonably sure that a change in the source code caused the regression, the underlying reason could not be found.

5.3.2 Robustness

Some of the commits are not correctly measured, making comparing with those impossible. These broken commits make it harder for the developer to trace back energy regressions, and more problematically, it might cause the developer to incorrectly identify a certain commit as the culprit, while the actual energy regression was caused by an earlier broken commit. This fragility of the tool is mostly not caused by the tool itself. Instead, it happens quite often that something in the tests itself breaks[40], resulting in errors. When the tests fail, the energy testing also fails. In a few cases, the energy measurement was broken, while the tests ran fine. We can trace these cases back to significant updates to the environment, such as Node. These cases are very peculiar as the tool operates as a black box, thus the environment should not be relevant. What's more likely is that the error is caused by a mistake in the implementation of the tool.

5.3.3 non-deterministic data

One thing we always keep in the back of the head when looking at results is to evaluate them critically. We can rarely be sure that a certain change is the cause of an energy regression. There are dozens of different factors that have to be taken into account before you can be confident. Different testing environments can significantly affect the results of energy regression analysis. Variations in hardware, operating systems, and other environmental factors could influence energy consumption measurements. It is important to consider these variables when interpreting the results. Throughout the experiments, we always made sure to remain speculative and open to other explanations. This is why defining logical thresholds is important.

There has been much debate about where to set the cut-off of an energy regression. A couple of different methods were discussed, before settling for the current percentage threshold. You could use the standard deviation to identify energy regressions. The benefit is that this method does not use hard values and works for any amount of joules. The problem with this is that highly fluctuating projects could potentially never get flagged for energy regressions despite constantly having big changes in energy consumption. In a hypothetical situation where an energy regression occurs for every single commit, using this method those regressions would not all show up. On the other hand, a very stable project could have a minor irrelevant energy regression get flagged, just because it is more significant than others from that project. Another method is to set a hard division of energy consumption. For instance, flag any change in energy consumption over fifty joules as significant. The problem with this method is that the size of the testing suite matters a lot. Some testing suites in our experiments do not reach over fifty joules in total and thus would never have a commit flagged. Other projects, like *Flutter*, might have too many commits flagged as energy regression. In the end, we ended up choosing a percentage-based definition for energy regression. The downside of this method is that it is very hard to reach the significance threshold for large codebases. You could introduce an unoptimised change, causing an energy regression, and still not be flagged with this method if the codebase is large enough. Ultimately, the percentage-based threshold proved to be the most logical, finding the most energy regressions with a low amount of false positives and negatives.

Chapter 6

Conclusions

This chapter gives an overview of the project's contributions. After this overview, we will conclude by answering the research question. We will also provide our reflections on the entire thesis. Finally, some ideas for future work will be discussed.

6.1 Can automated energy regression testing contribute towards finding energy regressions in software applications?

This is the research question that we defined at the start of this research paper. To answer the question, we built an energy regression testing tool that works automatically through a CI pipeline. This tool was then implemented in a diverse range of projects. Through the process of backtracking, the tool was applied to historical versions of these projects. The resulting data was put in an energy consumption timeline that shows the energy consumption of the testing suite over time. With this, energy regression could easily be found among the data.

So, based on the results from chapter 5, we can confidently say that energy regressions can be found by utilising automated energy regression testing. Energy regressions were found that could logically be the result of the underlying code changes, such as updates to the testing framework or additional unit tests. Also, multiple energy regressions could be traced back to a few line changes, making us confident that these were the source of the energy regression.

6.2 Contributions

In this research, we have made contributions to the scientific understanding and practical application of energy regression testing in software development.

First, we validated the effectiveness of energy regression testing through our experiments involving thirteen real-world software projects. These projects range from all-time most popular projects to newer, but highly active projects, providing a diverse range of projects for evaluation. Through these experiments, we demonstrated that E-Compare could reliably identify energy regressions. We traced multiple regressions back to specific code changes, confirming the tool's accuracy and relevance.

Second, we developed a state-of-the-art energy regression testing tool called E-Compare. This tool automates the process of detecting energy regressions by comparing the energy consumption of different versions of a project. E-Compare integrates into CI pipelines with minimal manual labour, enabling continuous monitoring and reporting of energy consumption changes. This is all done fully automatically. By alerting developers to potential energy regressions caused by code changes, E-Compare helps them to make informed decisions about whether the benefits of those changes justify the increased energy consumption.

6.3 Reflection

This paper and its contents have been the culmination of nine months of work. Throughout this period, we have been working on many different things. Going from background research to developing a functional tool, to conducting experiments, to processing results, to writing a thesis paper.

The majority of my time has been spent developing the tool. As I have previously never worked extensively with CI tools, this was not an easy task. With much trial and error, the result is something I am happy to bring to the scientific world and anyone who may want to continue with the tool. In the end, 32 different versions of the tool were created before reaching the last and current version v1.0. In total, 356 commits have been made.

Working on the experiments has been much more demanding of me than I originally anticipated. More manual labour ended up being required. These experiments have been very educational to me personally. Diving into dozens of different codebases, investigating what they do, how they would do that, what frameworks they use, etc, has been a valuable experience. This is one of the big reasons why I enjoyed working on this thesis. Though, it is also one of the limitations, as we discuss in the next section.

6.4 Limitations

This section only goes into the limitations of the experiments performed. For technical and conceptual limitations of the E-Compare tool, go to section 3.4.

Superficial analysis Within the experiments, many different projects were analysed. As we had no experience with the source code of any of these projects, the analysis was always gonna be rather superficial. In an intended situation, it will be an actual developer analysing the results given by E-Compare. Because of this, there may be commits analysed in these experiments that could have been analysed further had someone else looked at the results.

Splitting up results in individual tests Currently, the energy consumption is measured for all tests combined. This way, a regression will typically only occur if it impacts multiple tests. However, a developer that only changes one function would want to be notified if that function now significantly differs in energy consumption. Ideally, you would like to compare individual unit tests across different versions of the software. If one unit test shows a regression, the developer can be made aware despite the total energy consumption not significantly changing.

Backtracking method limitations One flaw with the backtracking method in the experiments is that it is not how the tool would normally be used. The result is that more commits are checked in the experiments than E-Compare would in real-world cases. Now, every commit on the main branch is checked when it passes through the filters. With actual use of the tool, only commits that get merged in the main branch are checked. These merging commits are typically a combination of multiple commits and are larger than normal. This means that the experiments might not exactly reflect the actual use of the tool. If time is no constraint, future research could monitor actual natural use of the tool, instead of simulating natural use.

Only main branch analysed Furthermore, through normal use of the tool, the developer can see the energy consumption of commits made to other branches than the main branch. This allows developers to analyse smaller pieces of code, making it easier to pinpoint possible causes of energy regressions. In these experiments, only the merges are analysed, focusing on the combined code changes of an entire branch rather than individual commits made to that branch. Future research could include commits to other branches than main.

Limited testing types tested One big limitation is the lack of diversity among types of tests. For the projects analysed in these experiments, the unit tests were used for reasons stated in Chapter 4. Many other types of tests were not used, even though they are usable in combination with E-Compare. The results might be different for other testing types. The usefulness of E-Compare for these types is not yet proven. Future research could focus on other testing types.

Standard deviation One thing that was not done is to look at the standard deviation. For some repositories, the fluctuations were checked with the individual runs to find an explanation. If energy consumption between individual runs is very inconsistent it could lead to questionable data. On the other hand, if there is an energy regression and the individual runs are stable it provides much more credibility.

Outlier removal To improve the accuracy of the average and standard deviation, outlier removal could be implemented. Individual runs that differ wildly from the norm can be skipped when calculating the average or standard deviation.

Caching We repeat tests multiple times to diminish the impact of outliers. But sometimes the first test costs significantly more joules than any following tests. This can happen due to caching. The testing framework stores the results for the tests so that it does not need to run them again somewhere down the line. The result is that the repetition of tests is useless and even makes the results inaccurate. Every selected project was checked if any caching occurred. In only one case, *Hiddify*, did caching occur. Caching is not a big issue as long as researchers and developers are aware of it.

Graphical limitations The graphs introduce a few inevitable shortcomings because of the way they are shown. Firstly, the y-axis does not start at zero, nor is the same place for every graph. This makes the energy regressions seem larger than they are. Secondly, the differences between steps on the y-axis also change depending on the project. This is important to note before readers compare the energy regressions of two different graphs. The length of one bar on one graph does not equate to the same values as a similar length bar on another graph. Third and last, large energy regressions in one project might mask other potential regressions in the graph, as a large regression stretches the y-axis, making other bars smaller.

6.5 Future

This section describes potential future alleys for these experiments. Future researchers can use these suggestions to continue the research done in this paper. The suggestions made have been skipped due to time constraints, or because other methods had been prioritised. The future direction of the tool is discussed in subsection 3.4.3.

Limited data Despite the large amount of repositories and commits analysed, only a few relevant energy regressions could be found. While we can be certain that they exist and that the tool can spot them. In future work, these experiments can be expanded even further to a wider net of projects.

Backtracking method Due to time constraints, the backtracking method was used. As stated in section 6.4, this method might give slightly different results than when E-Compare is normally used. Future research could observe actual projects using E-Compare in real-time. This method stays more true to the actual use, allowing for more accurate experiment results. The main differences are that fewer commits are checked and the average checked commit is larger. It would be interesting to see if the results of the experiments would be the same as the current results.

Long-term developer interaction Also, further exploration could take a look at the behaviour of developers using E-Compare over an extended period. This allows developers time to interact with E-Compare and make adjustments to their code based on the insights provided by the tool. The primary goal is to investigate the dynamic relationship between developers and E-Compare, examining whether the tool influences decision-making regarding code optimisation for energy efficiency. This would capture changes in developer responses over time and explore whether the tool prompts adjustments in coding practices and resource usage. As this research primarily focused on whether the tool could identify energy bugs, it does not necessarily guarantee that developers will adapt their code to address these issues. As developers engage with E-Compare, it would be valuable to observe whether the tool inspires them to actively modify their code or adopt more energy-efficient coding practices. Gathering qualitative feedback from developers is important to understand their experiences & the challenges they faced with the tool, and the perceived impact of E-Compare on their development workflows.

Bibliography

[1] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, and A. Friday, "The real climate and transformative impact of ict: A critique of estimates, trends, and regulations," *Patterns*, vol. 2, no. 9, p. 100340, 2021.

[2] A. S. G. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *MDPI*, Apr 2015.

[3] L. Belkhir and A. Elmeligi, "Assessing ict global emissions footprint: Trends to 2040 & recommendations," *Journal of Cleaner Production*, vol. 177, pp. 448–463, 2018.

[4] B. Knowles, "Acm techbrief: Computing and climate change," 2021.

[5] A. Katal, S. Dahiya, and T. Choudhury, "Energy efficiency in cloud computing data centers: a survey on software technologies," *Cluster Computing*, vol. 26, pp. 1845–1875, June 2023.

[6] A. Petrosyan, "The state of developer ecosystem in 2023." `https://www.statista.com/statistics/617136/digital-population-worldwide/`, May 2024. Accessed: 13-05-2024.

[7] G. W. Index, "Digital vs. traditional media consumption." `https://www.gwi.com/hubfs/Digital_vs_Traditional_Media_Consumption.pdf`, 2017. Accessed: 13-05-2024.

[8] G. Moore, "Cramming more components onto integrated circuits (1965)," *Electronics Magazine*, 4 1965.

[9] B. Publishing, "Global high performance computing market." `https://www.bccresearch.com/market-research/information-technology/high-performance-computing-market.html`, 2022. Accessed: 13-05-2024.

[10] Jetbrains, "The state of developer ecosystem in 2023." `https://www.jetbrains.com/lp/devecosystem-2023/languages/`, 2023. Accessed: 13-05-2024.

[11] B. Danglot, J.-R. Falleri, and R. Rouvoy, "Can we spot energy regressions using developers tests?," *arXiv preprint arXiv:2108.05691*, 2021.

[12] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 835–848, 2007.

[13] L. Cruz, "Tools to measure software energy consumption from your computer," *Blog post*, July 2021.

[14] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," *ArXiv*, vol. abs/2312.13897, 2023.

[15] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, F. Ardanaz, A. Al-Rawi, K. Livingston, F. Keceli, M. Maiterth, and S. Jana, "Global extensible open power manager: A vehicle for hpc community collaboration on co-designed energy management solutions," in *High Performance Computing* (J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, eds.), (Cham), pp. 394–412, Springer International Publishing, 2017.

[16] S. I. Roberts, S. A. Wright, S. A. Fahmy, and S. A. Jarvis, "The power-optimised software envelope," *ACM Trans. Archit. Code Optim.*, vol. 16, jun 2019.

[17] C. Marantos, K. Salapas, L. Papadopoulos, and D. Soudris, "A flexible tool for estimating applications performance and energy consumption through static analysis," *SN Computer Science*, vol. 2, no. 1, p. 21, 2021.

[18] M. Poess and R. Othayoth Nambiar, "A power consumption analysis of decision support systems," in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, (New York, NY, USA), p. 147–152, Association for Computing Machinery, 2010.

[19] J. D. Davis, S. Rivoire, M. Goldszmidt, and E. K. Ardestani, "Chaos: Composable highly accurate os-based power models," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 153–163, 2012.

[20] N. Kim, J. Cho, and E. Seo, "Energy-based accounting and scheduling of virtual machines in a cloud system," in *2011 IEEE/ACM International Conference on Green Computing and Communications*, pp. 176–181, 2011.

[21] S. A. Chowdhury and A. Hindle, "Greenoracle: estimating software energy consumption with energy measurement corpora," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), p. 49–60, Association for Computing Machinery, 2016.

[22] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, “Greenscaler: training software energy models with automatic test generation,” *Empirical Software Engineering*, vol. 24, no. 4, pp. 1649–1692, 2019.

[23] N. Rteil, R. Bashroush, R. Kenny, and A. Wynne, “Interact: It infrastructure energy and cost analyzer tool for data centers,” *Sustainable Computing: Informatics and Systems*, vol. 33, p. 100618, 2022.

[24] G. C. Berlin, “Cloud energy | green coding.” `https://www.green-coding.io/projects/cloud-energy/`. Accessed: November 6, 2023.

[25] G. KP, G. Pierre, and R. Rouvoy, “Studying the Energy Consumption of Stream Processing Engines in the Cloud,” in *IC2E 2023 - 11th IEEE International Conference on Cloud Engineering*, (Boston (MA), United States), pp. 1–9, IEEE, IEEE, Sept. 2023.

[26] A. Kruglov, G. Succi, and X. Vasuez, “Incorporating energy efficiency measurement into ci cd pipeline,” in *Proceedings of the 2021 European Symposium on Software Engineering*, ESSE ’21, (New York, NY, USA), p. 14–20, Association for Computing Machinery, 2022.

[27] G. C. Berlin, “Eco ci | green coding.” `https://www.green-coding.io/projects/eco-ci/`. Accessed: November 6, 2023.

[28] F. Quesnel, H. K. Mehta, and J.-M. Menaud, “Estimating the power consumption of an idle virtual machine,” in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pp. 268–275, 2013.

[29] H. Acar, G. I. Alptekin, J.-P. Gelas, and P. Ghodous, “The Impact of Source Code in Software on Power Consumption,” *International Journal of Electronic Business Management*, vol. 14, pp. 42–52, 2016.

[30] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, “What do programmers know about the energy consumption of software?,” *PrePrints*, 3 2015.

[31] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, “An empirical study of practitioners’ perspectives on green software engineering,” in *Proceedings of the 38th international conference on software engineering*, pp. 237–248, 2016.

[32] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes, “The influence of the java collection framework on overall energy consumption,” in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, pp. 15–21, 2016.

[33] M. Funke, P. Lago, E. Adenekan, I. Malavolta, and I. Heitlager, "Experimental evaluation of energy efficiency tactics in industry: Results and lessons learned," in *21st IEEE International Conference on Software Architecture (ICSA)*, Feb. 2024.

[34] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Perphecy: Performance regression test selection made simple but effective," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 103–113, 2017.

[35] D. Alshoaibi, K. Hannigan, H. Gupta, and M. W. Mkaouer, "Price: Detection of performance regression introducing code changes using static and dynamic metrics," in *Search-Based Software Engineering* (S. Nejati and G. Gay, eds.), (Cham), pp. 75–88, Springer International Publishing, 2019.

[36] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 25–36, 2016.

[37] A. Hindle, "Green mining: a methodology of relating software change and configuration to power consumption," *Empirical Software Engineering*, vol. 20, pp. 374 – 409, 2013.

[38] D. Mateas, "Ci pipeline for energy variability management." `https://www.green-coding.io/case-studies/ci-pipeline-energy-variability/`, September 2023. Accessed: April 5, 2024.

[39] K. Huppler, K.-D. Lange, and J. Beckett, "Spec: enabling efficiency measurement," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, (New York, NY, USA), p. 257–258, Association for Computing Machinery, 2012.

[40] M. M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 356–367, 2017.

Appendix A

Plots

In this appendix we show higher quality versions of the charts shown in chapter 4. These versions can be used to look up commit hashes and to look at the energy regressions more easily.

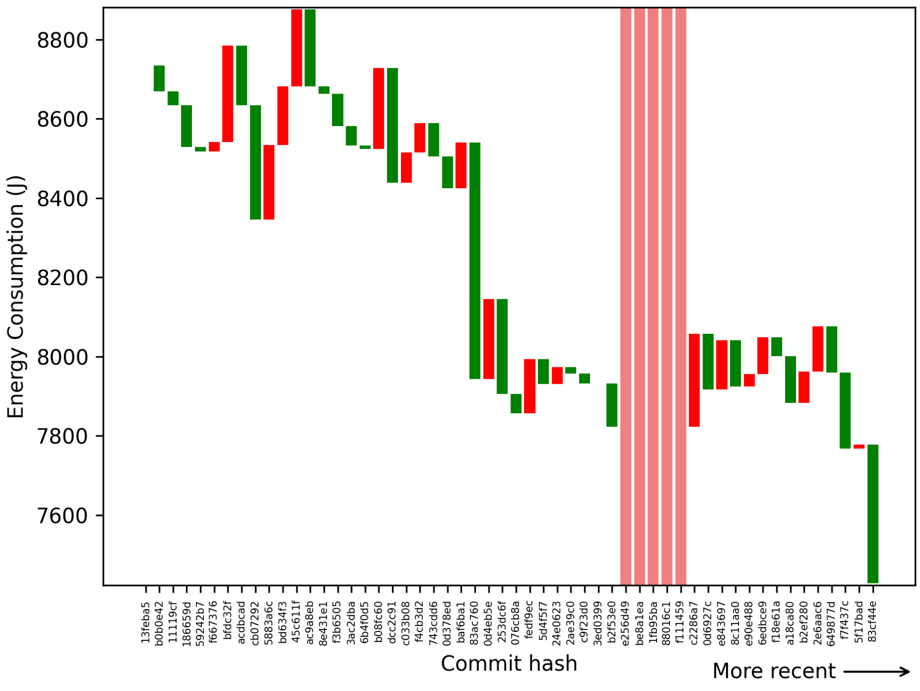


Figure A.1: Flutter

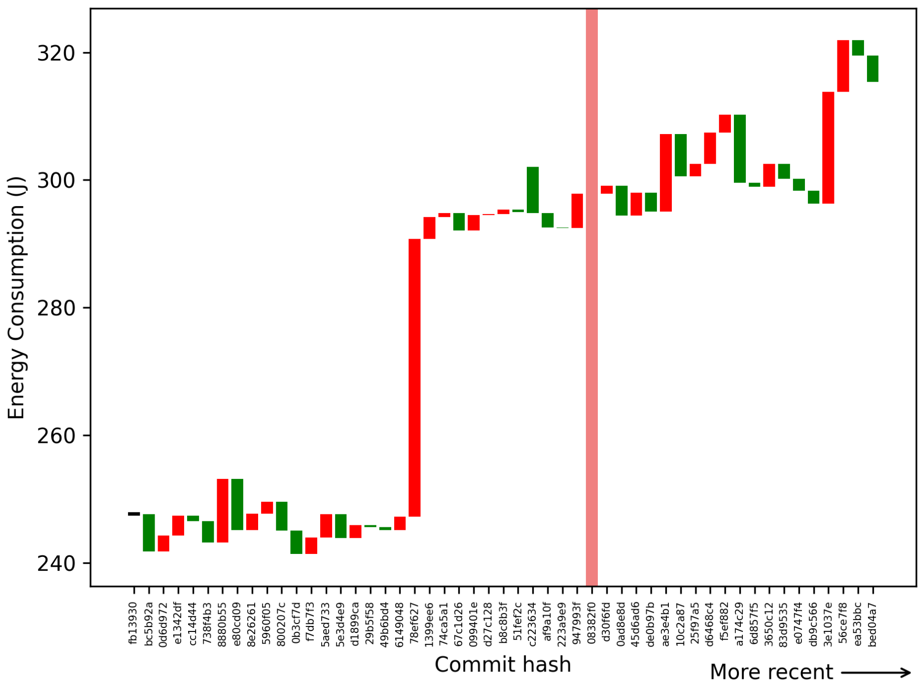


Figure A.2: Vue

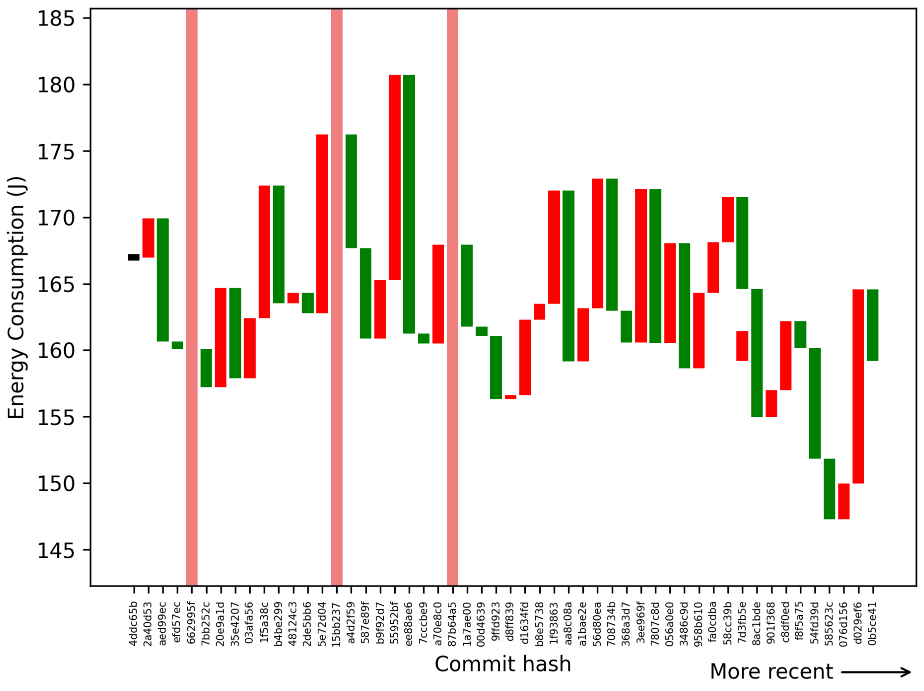


Figure A.3: Bootstrap

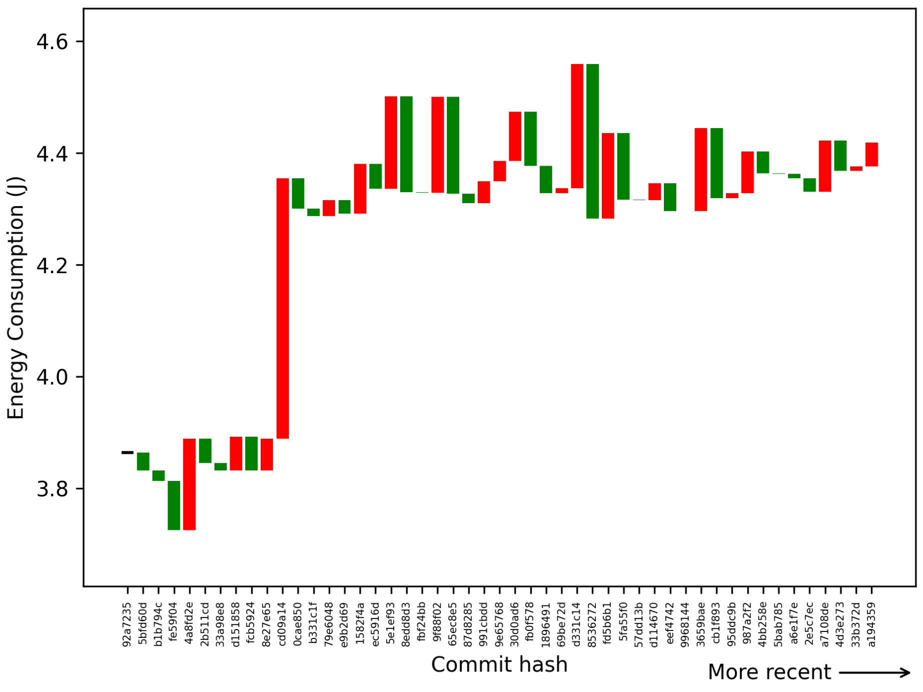


Figure A.4: D3

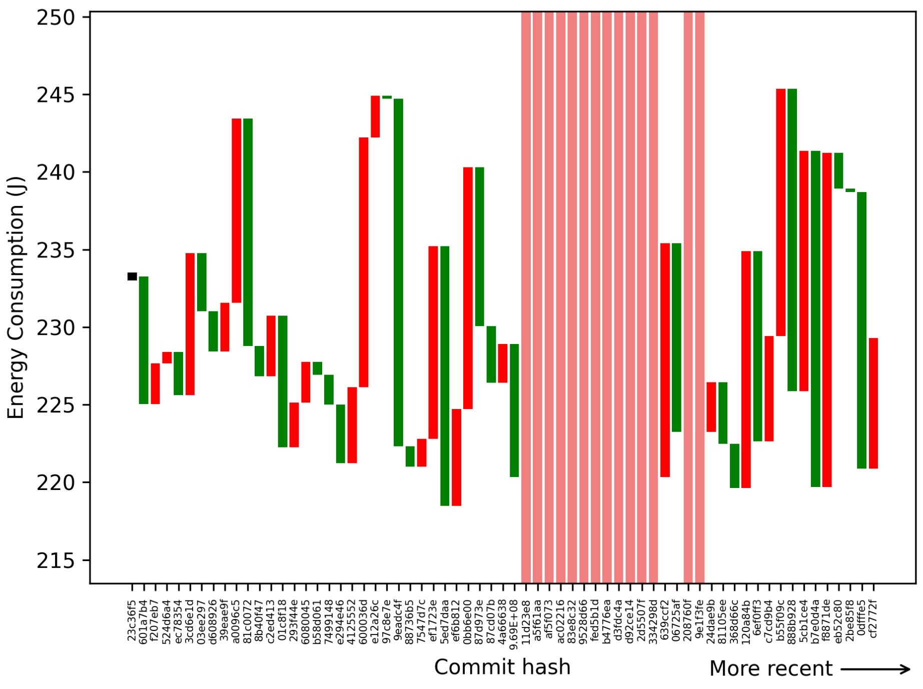


Figure A.5: Stable Diffusion

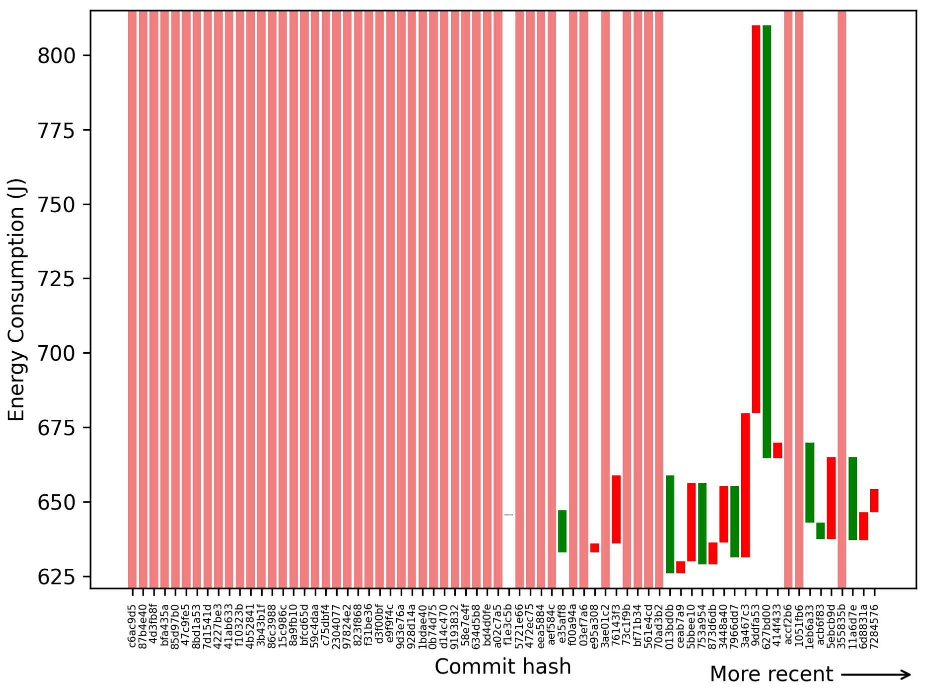


Figure A.6: Puppeteer

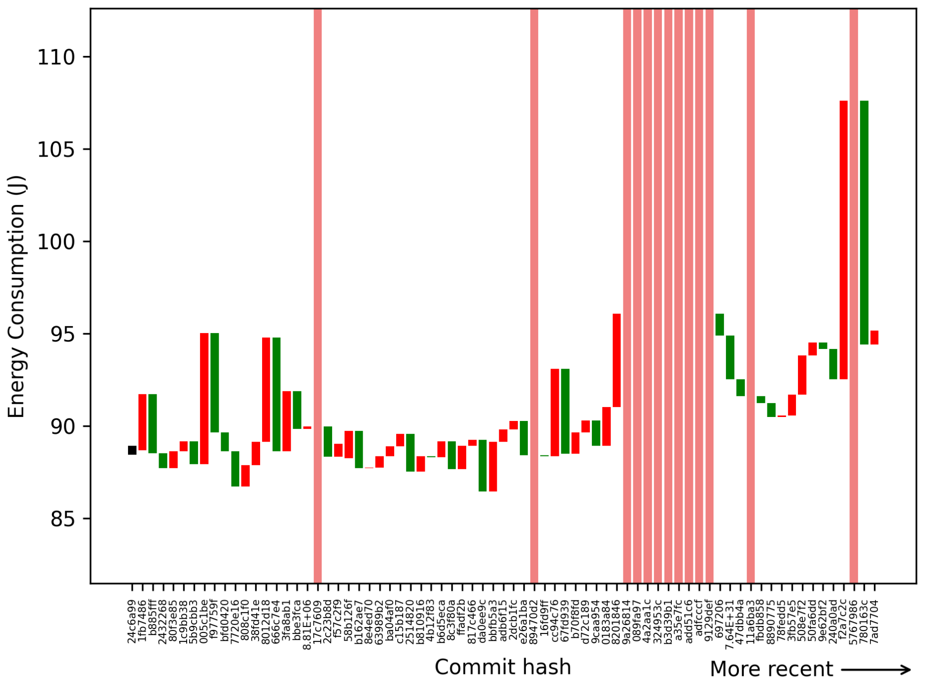


Figure A.7: Tailwind

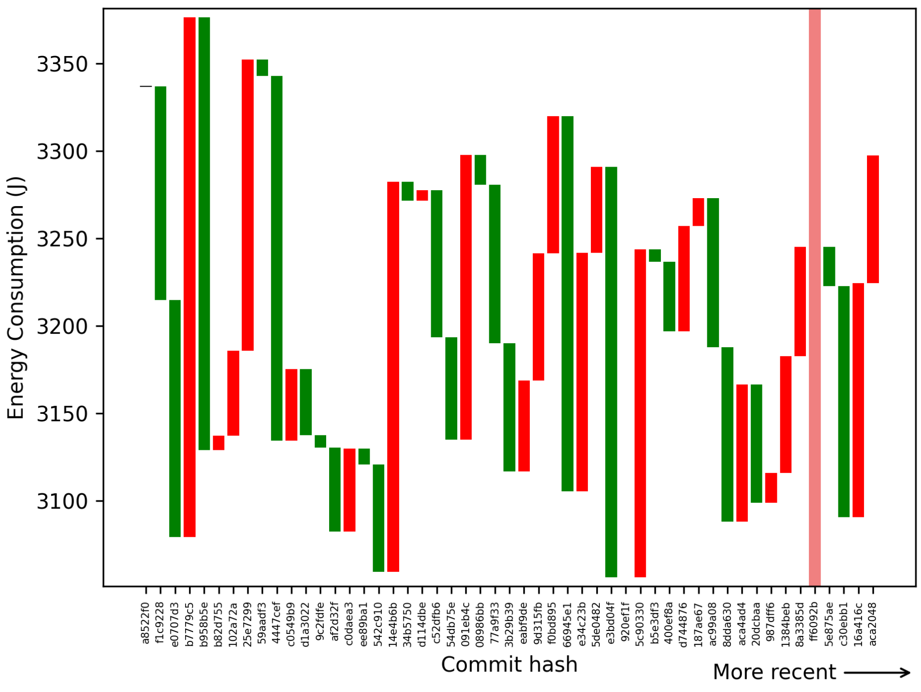


Figure A.8: Neovim

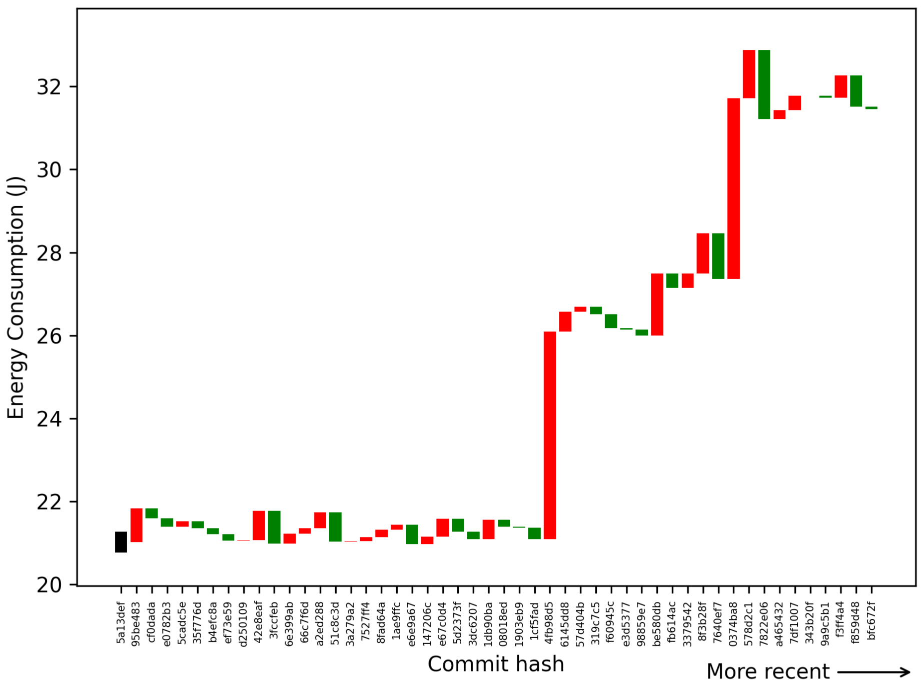


Figure A.9: ShadowCN UI

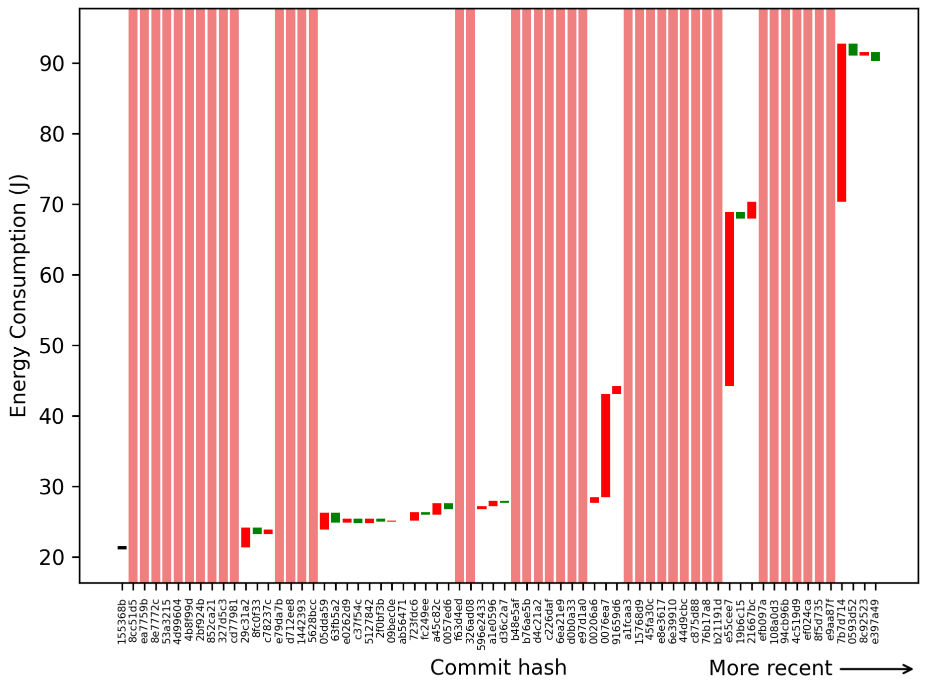


Figure A.10: CrewAI

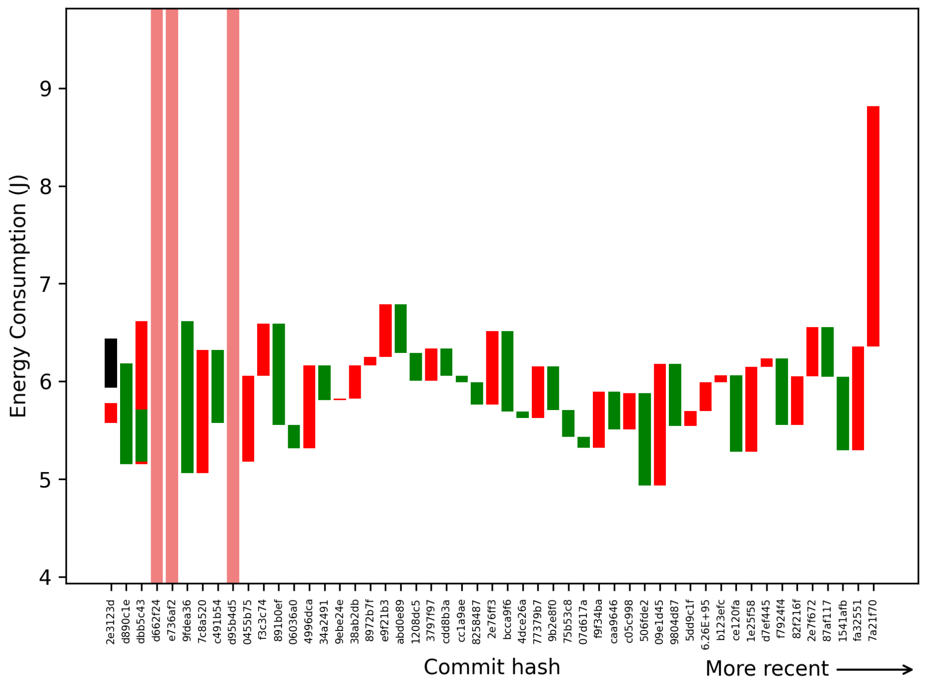


Figure A.11: Atomicals

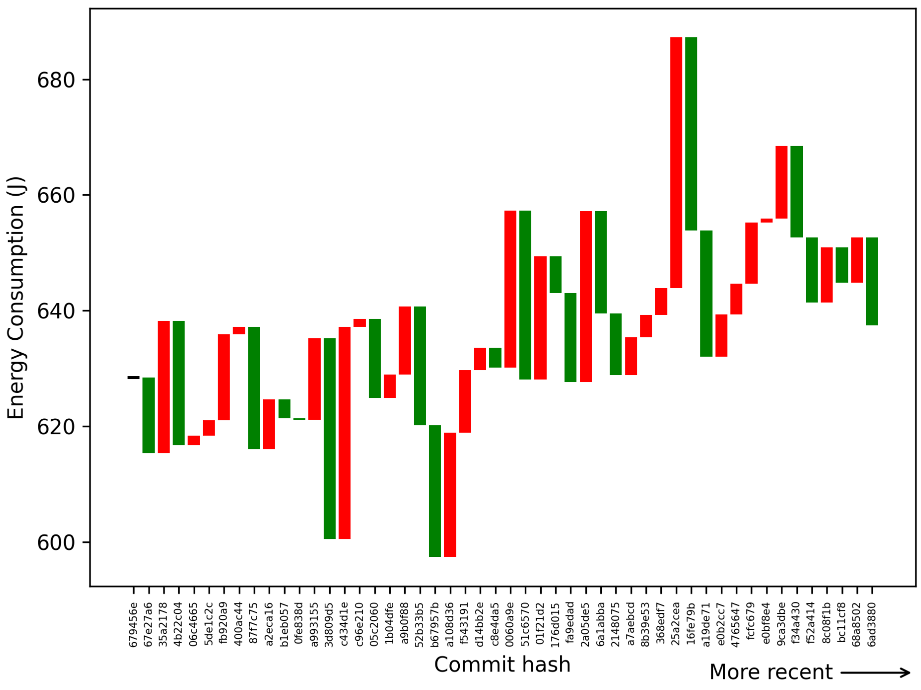


Figure A.12: Twenty

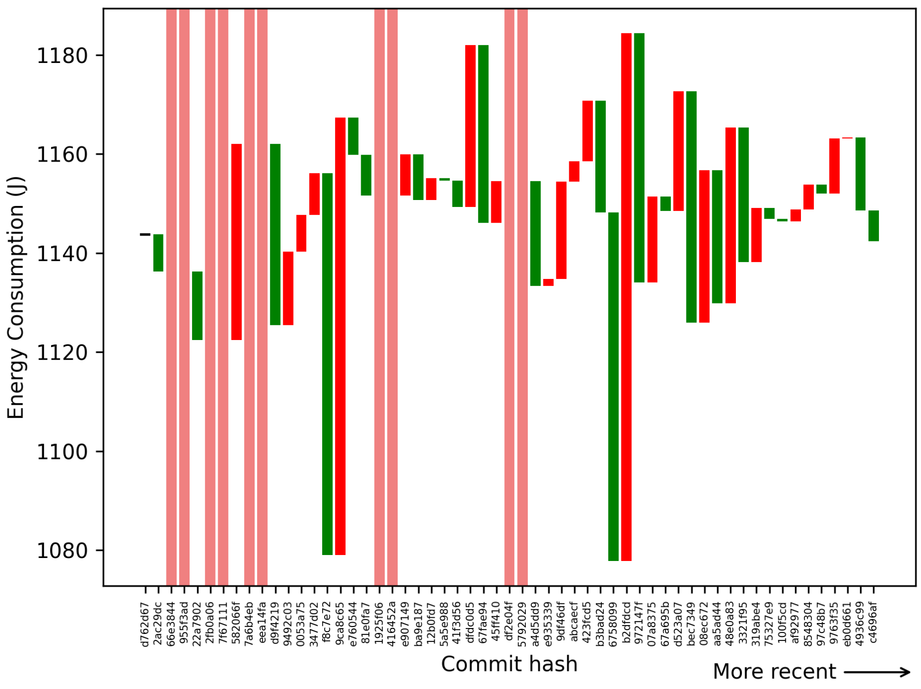


Figure A.13: Plate

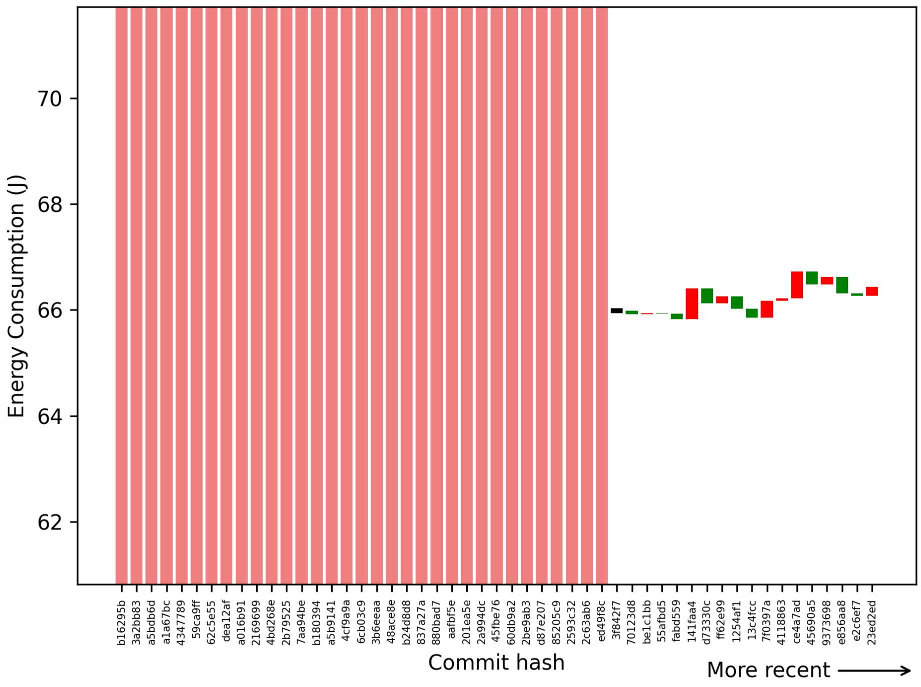


Figure A.14: Hiddify