With Great Power Comes Great Responsibility: A Tool for Energy-Aware Java Development

Elena Mihalache

With Great Power Comes Great Responsibility: A Tool for Energy-Aware Java Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Elena Mihalache
born in Bucharest, Romania

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

©2026 Elena Mihalache. All rights reserved.

With Great Power Comes Great Responsibility: A Tool for Energy-Aware Java Development

Author: Elena Mihalache

Abstract

As the energy consumption of the ICT sector continues to grow, there is an increasing need for developers to reason about the energy efficiency of their code. However, most energy measurement tools operate at the application level and require significant workflow disruption, leaving developers without accessible, in-situ feedback during development. In this thesis, we investigate the technical and practical feasibility of a lightweight, software-based energy measurement tool for Java code snippets, implemented as an IntelliJ IDEA plugin backed by a JShell execution environment and the Linux `powercap` framework. To evaluate the proposed tool, we conduct a two-phase study. In a verification phase spanning 30 algorithmic problem pairs and 1800 measurements, the tool detects statistically significant energy consumption differences in 83.3% of cases. In a mixed-methods validation study with 22 participants, accuracy in identifying the more energy-efficient implementation rises from 56.8% when relying on source code inspection alone, to 97.7% when using the tool, alongside an increase in participant confidence. Qualitative analysis further reveals that the tool assists in correcting flawed intuitions and provides educational value. These results suggest that fine-grained, in-IDE energy measurement is both technically achievable and empirically beneficial, and constitutes a concrete step toward making energy-aware development a routine part of software engineering practice.

Thesis Committee:

Chair & university supervisor: Prof. dr. A.E. Zaidman
Daily co-supervisor: Dr. X. Liu
Committee member: Dr.ir. J. Yang

Preface

It has been six years since I came to Delft for the first time. I consider myself lucky to have both started my Bachelor's and finished my Master's with Professor Zaidman, whom I would like to thank for all the support and advice provided. I cannot imagine receiving better guidance. I would also like to thank Xutong for giving me the opportunity to work with her and for all the time and enthusiasm she offered over the duration of this thesis. I am honoured to have had the chance to complete this work under their mentorship. I learnt so much and have truly enjoyed every moment of this process. Last, I am of course grateful to my family without whom I would not have been able to write this thesis in the first place, and to my friends for all their love and support over the past years.

Elena Mihalache
Delft, The Netherlands
June 22, 2026

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Terminology 3
- 1.2 Research Questions 3
- 1.3 Structure of the Thesis 4
- 1.4 Replication Package 4

2 Related Work 5
- 2.1 Taxonomy of Energy Measurement 5
- 2.2 Intel RAPL 6
- 2.3 Energy Measurement Tools 9
- 2.4 The Missing Pieces 13

3 Implementation 16
- 3.1 Backend 16
- 3.2 Frontend 21
- 3.3 Putting Everything Together 26

4 Study Setup 33
- 4.1 Verification 33
- 4.2 Validation 38

5 Verification 48
- 5.1 Results 48
- 5.2 Analysis of Exceptions 50
- 5.3 Summary 58

6 Validation **59**
6.1 Participant Overview 59
6.2 Quantitative Data Results 61
6.3 Qualitative Data Results 64
6.4 Summary 71

7 Discussion **72**
7.1 Revisiting the RQs 72
7.2 Implications 75
7.3 Limitations 76
7.4 Threats to Validity 77
7.5 Ethical Considerations 79
7.6 Use of Generative AI 80

8 Conclusion **82**

Bibliography **84**

A Glossary **90**

B Histograms with Gaussian Curve Overlay **92**

C Kernel Density Estimation (KDE) Plots **105**

D Microsoft Forms User Study Survey **112**

E Participant Consent Form **122**

List of Figures

2.1 Hierarchy of Intel RAPL power domains as presented in Khan et al. [26]. . . . 7

2.2 Visualisation of the RAPL `sysfs` directory structure on an AMD Ryzen 9 5900HX CPU using Kernel 6.14.0-36-generic. Generic kernel attributes (e.g., `power/`, `uevent`) are collapsed for clarity. 8

3.1 The backend structured as a Maven project and its respective package organisation. 18

3.2 Example use of the backend interactive mode of operation. 19

3.3 Example measurement output in the IDE tool window, as shown in light theme. 24

3.4 Primary interaction entry points for triggering the energy measurement action within the IntelliJ IDEA editor, as shown in light IDE theme. 25

3.5 The plugin user interface components, showing the floating toolbar for action triggering, the non-modal progress dialog for background execution, and the persistent console view displaying execution results and energy metrics in light IDE theme. 27

3.6 Communication flow of the energy measurement process. The plugin extracts the user's selection (a), constructs a context-aware string (b), and passes it as an argument to the backend via a command-line single-shot invocation (c). The backend executes the snippet and returns a structured output string (d). 29

3.7 High-level data flow diagram showing the interaction between the IntelliJ IDEA frontend and the backend components. (a) The flow of data in the IntelliJ IDEA frontend components, starting from the code editor selection. (b) The energy monitoring backend, showing the data flow from the start of the process to the collection of energy metrics. 31

3.8 The execution workflow of the energy analysis process, illustrating the sequence of steps from user input to the display of results. 32

4.1 Selected code snippet used for the energy and time measurement of the *fast* solution for the *PerfectRectangle* problem alongside a respective measurement output given by the plugin. 35

4.2 The data collection pipeline from problem selection to metric capture. 35

4.3 Decision flow for selecting the appropriate statistical tests based on data distribution constraints. 39
4.4 Experimental Java implementation pairs used for task-based evaluation, representing common performance and energy-efficiency scenarios. The highlights (the yellow-shaded regions) pinpoint the lines of code that represent the scenario being tested. 43

5.1 Spearman rank correlation between median execution time and median energy consumption across algorithm implementations. Each point represents one implementation (*fast* or *slow* variants) in rank space. The fitted rank-space trend line and reported ρ (with p-value) indicate a strong positive monotonic association. 54
5.2 Density plots for the *Candy* problem comparing execution time and energy consumption between the *fast* and *slow* solutions. 55
5.3 Density plots for the *ShortestPalindrome* problem comparing execution time and energy consumption between the *fast* and *slow* solutions. 56
5.4 Density plots for the *SlidingWindowMaximum* problem comparing execution time and energy consumption between the *fast* and *slow* solutions. 57
5.5 Density plots for the *SelfCrossing* problem comparing execution time and energy consumption between the *fast* and *slow* solutions. 57
5.6 Density plots for the *MaxPointsOnALine* problem comparing execution time and energy consumption between the *fast* and *slow* solutions. 58

6.1 Overview of participant demographics and technical background. 60
6.2 Performance accuracy scores broken down by specific tasks and the overall average, comparing participants working *with the tool* to those working *without the tool*. Data labels show the percentage accuracy and absolute number of correct responses. 62
6.3 Visualisation of the inductive initial (or open) coding results. 66
6.4 Final thematic map of participant experiences with the energy-analysis tool, illustrating core themes, sub-patterns, and inter-thematic relationships. 67

B.1 Energy consumption distributions with normal curve overlay (Part 1). 93
B.2 Energy consumption distributions with normal curve overlay (Part 2). 94
B.3 Energy consumption distributions with normal curve overlays (Part 3). 95
B.4 Energy consumption distributions with normal curve overlay (Part 4). 96
B.5 Energy consumption distributions with normal curve overlay (Part 5). 97
B.6 Energy consumption distributions with normal curve overlay (Part 6). 98
B.7 Execution time distributions with normal curve overlay (Part 1). 99
B.8 Execution time distributions with normal curve overlay (Part 2). 100
B.9 Execution time distributions with normal curve overlay (Part 3). 101
B.10 Execution time distributions with normal curve overlay (Part 4). 102
B.11 Execution time distributions with normal curve overlay (Part 5). 103
B.12 Execution time distributions with normal curve overlay (Part 6). 104

C.1 Energy consumption KDE plots comparing `fast` and `slow` implementations (Part 1). 106
C.2 Energy consumption KDE plots comparing `fast` and `slow` implementations (Part 2). 107
C.3 Energy consumption KDE plots comparing `fast` and `slow` implementations (Part 3). 108
C.4 Execution time KDE plots comparing `fast` and `slow` implementations (Part 1). 109
C.5 Execution time KDE plots comparing `fast` and `slow` implementations (Part 2). 110
C.6 Execution time KDE plots comparing `fast` and `slow` implementations (Part 3). 111

Chapter 1

Introduction

The Information and Communications Technology (ICT) sector is a significant and growing contributor to global energy consumption and carbon emissions [27, 42]. According to Verdecchia et al., by 2030 software and IT "*will be one-third of the global demand*" [70, p. 7] in terms of energy consumption. There are multiple reasons to care about the increasing energy use caused by software, be they personal, economic, or environmental. From a user perspective, energy efficient software prolongs battery life on mobile devices, as no one wants phones or laptops to die quickly [23]. Economically, companies have shown growing interest in software power consumption in the last decade [23], and this interest has since intensified, as ecological transformation is thought to be the next major wave of innovation after the digital transformation, reflected in the fact that "*investors are currently changing behaviours toward financing based on ecology and sustainability*" [70, p. 13]. Environmentally, excessive energy use accelerates carbon emissions. Thus, as our world becomes ever more dependent on software, developers and engineers bear a social responsibility to mitigate this environmental impact [60].

However, to understand whether energy consumption is in itself problematic, it must be related to its environmental impact, which depends on carbon intensity [74]. Carbon intensity represents the number of grams of CO_2 released to produce a kilowatt hour (kWh) of electricity [55], and varies depending on whether energy is generated from renewable resources or fossil fuels [74]. This makes it difficult to estimate how polluting energy usage truly is. One might argue that if the energy used is green, meaning it comes from low-carbon sources, such as renewable energy, there is less need to reduce consumption. Nonetheless, Verdecchia et al. counter that "*renewable energy only cures the symptoms*" [70, p. 7] and does not actually reduce the need for energy. Thus, lowering consumption remains an important goal even when renewable and other low-carbon sources are available.

A major obstacle to achieving energy efficiency in software has been that most programmers lack awareness and appropriate tools [42, 56]. Pang et al. [44] found that as recently as 2015, developers did not prioritise energy efficiency because customers and clients were not demanding it. Similarly, Verdecchia et al. argue that "*designers determine the ecologic footprint of their software,*" [70, p. 11] yet practitioners and researchers are largely ignorant of energy concerns [18]. Software engineers are generally not trained to consider or manage energy consumption in the systems they create [18]. In this regard, Verdecchia et al. stress

that moving forward, software developers "*must become active in ecological behaviors*" [70, p. 7], while Fonseca et al. assert that the software community must now "*design for, monitor, and manage software usage with energy in mind*" [18, p. 79].

Concerns may arise that energy awareness could compromise other aspects of software quality [42]. However, "*software quality should not come at the expense of energy awareness*" [18, p. 81]. The call for energy aware software development does not mean energy efficiency should override other quality attributes, but that it should be considered alongside them during design. It is possible to have similar software systems with significantly different energy profiles, implying that conscious design decisions can improve energy efficiency without sacrificing functionality [18].

One path toward sustainable software is making systems more energy aware [74]. As Fonseca et al. argue, "*measuring energy consumption is a first step towards energy awareness*" [18, p. 81]. Measurement can target either the software development process or the deployed software [74]. Yet, even when developers attempt to measure energy usage, they often do so incorrectly, despite using specialised tools [44]. There is thus a strong call for user-friendly development tools that can identify inefficient code and suggest improvements [44]. This is further argued by Zaidman [74], who stresses that programmers need direct feedback on energy consumption, and by Schubert et al., who note that developers lack the tools and data necessary to "*pinpoint energy-hungry sections in their code*" [56, p. 515].

To measure the energy efficiency of the software development process or that of the deployed software, testing the power consumption of an entire application is expensive and time-consuming [23], which underlines the need for lightweight tools that integrate into the development process. Verdecchia et al. remark that every single line of code matters, reinforcing the value of analysing energy consumption at the code level rather than only in aggregate, advocating that "*software energy consumption must be measured, estimated, and predicted at all levels*" [70, p. 12]. Fonseca et al. likewise add that "*energy efficiency must be designed into a system early in its lifecycle*", as leaving such considerations until after deployment "*is a recipe for disaster*" [18, p. 80]. Their principle that "*energy awareness should be engineered throughout the lifecycle*" [18, p. 80] further supports developing tools that integrate early in the software creation process.

Given the aforementioned needs and current context, this work proposes and evaluates a practical solution centred on Java, one of the most widely used programming languages according to the 2025 TIOBE Index[1]. The Java Read-Eval-Print Loop (REPL), popularised by environments like JShell [67], provides a long-running and stateful execution model that serves as an experimental sandbox well-suited to in-situ energy measurement. Concretely, we engineer an IntelliJ IDEA plugin backed by a JShell execution environment and the Linux `powercap` framework, providing developers with immediate, in-situ energy feedback directly within their workflow. The REPL's capacity for immediate feedback differentiates this tool from others such as EnergiBridge[2] [54], which measure overall application consumption rather than the energy cost of individual code chunks. Such granularity can help developers, who often have little intuition about energy-efficient code, gain better aware-

[1] https://www.tiobe.com/tiobe-index/

[2] https://github.com/tdurieux/EnergiBridge

ness. The tool is also designed to be as lightweight as possible to encourage adoption. This approach aligns with the "*technology landscape for energy-efficient digital infrastructures*" envisioned by Verdecchia et al. [70, p. 8], contributing to "*energy-aware software solutions*" and to the promotion of "*conscious software developers.*"

We keep several considerations in mind when designing such a tool. Measuring energy precisely is difficult because hardware configurations differ across devices, and measurement tools vary in accuracy [74]. Nonetheless, Fonseca et al. [18] argue that "*understanding the trend of energy efficiency is more important than knowing the raw logged values.*" Thus, it is more valuable to identify consistent trends than to achieve perfect precision.

1.1 Terminology

Feedback is described by Karlin et al. as "*the process of giving people information about their behaviour that can be used to reinforce behaviour and/or suggest behaviour change*" [25, p. 377]. They further define *energy feedback* as "*information about actual energy use that is collected in some way and provided back to the energy consumer*" [25, p. 381].

Awareness is defined by the Cambridge Dictionary as "*knowledge that something exists, or understanding of a situation or subject at the present time based on information or experience*" [3]. In the context of green software engineering, Fonseca et al. define being *energy aware* as "*taking into account the energy consumption of software across the software development lifecycle*" [18, p. 81], and *energy-aware software* as "*software that is consciously designed and developed to monitor and react to energy preferences and usage*" [18, p. 79].

For the purpose of this thesis, we operationalise a *code snippet* or a *code chunk* as a user-selected fragment of Java code, ranging from basic expressions to complex algorithmic logic, extracted directly from the IDE editor. It is not a whole application or a complete file, but a discrete block.

1.2 Research Questions

The goal that we envision is to create an approach in the form of a tool that provides energy feedback from snippets of code and thus increases energy awareness of developers. Our investigation is guided by the following research question (MRQ):

> **MRQ:** How technically and practically feasible is it to implement a lightweight, software-based energy measurement tool for Java code snippets that provides immediate, reliable energy measurements to support developer reasoning about Java code energy efficiency?

Measuring reliability at the code-snippet level requires establishing whether the tool can distinguish between implementations with meaningfully different energy profiles. At the same time, technical feasibility alone is insufficient since a tool that is difficult to use or

[3] https://dictionary.cambridge.org/dictionary/english/awareness

that does not integrate naturally into a developer's workflow will not be adopted, regardless of how accurately it measures. The MRQ is therefore further supported by two research questions (RQ1, RQ2) that address these complementary dimensions.

> **RQ1:** Does the proposed energy measurement tool provide sufficiently reliable measurements such that statistically significant energy consumption differences between semantically equivalent but syntactically distinct Java code snippets can be detected on the tested hardware configuration?

Although differences in execution time between implementations are often straightforward to detect, the relationship between time and energy is not always proportional. RQ1 therefore tests not only whether the tool detects differences, but also whether those differences are both statistically significant and practically meaningful, and whether energy consumption scales consistently with execution time across a set of algorithmic problems.

> **RQ2:** How does the proposed energy measurement tool affect developers' ability to understand, reason about, and make decisions regarding the energy efficiency of Java code snippets?

A technically accurate tool may fail to support developers if its output is difficult to interpret or does not fit into their workflow. RQ2 investigates the practical dimension of the tool, examining whether access to measured energy data improves identification accuracy, raises developer confidence, and integrates usefully into the reasoning process.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 surveys existing work on software energy consumption, energy measurement tooling, and developer awareness. Chapter 3 details the design and engineering of the proposed plugin, covering the JShell backend, the `powercap` integration, and the IntelliJ IDEA frontend. Chapter 4 describes the methodology used to evaluate the tool, which is divided into a verification phase and a validation phase. Chapter 5 presents the results of the verification study, assessing whether the tool can reliably detect statistically significant energy differences between semantically equivalent Java implementations. Chapter 6 presents the results of the validation study, examining how the tool affects developer reasoning and decision-making about energy efficiency. Chapter 7 discusses the findings of both studies, their implications, and the limitations of this work. Finally, Chapter 8 concludes the thesis and outlines directions for future work.

1.4 Replication Package

All materials required to replicate the studies presented in this thesis, including the plugin source code, statistical analysis code, anonymised participant data, and experimental scripts, can be found in our replication package [35].

Chapter 2

Related Work

To understand the current state of energy measurement, it is necessary to investigate the parallel evolution of computer systems and the measurement methodologies developed to monitor them. Historically, energy optimisation relied on hardware improvements such as more efficient transistors, better cooling, and power-gated circuits. However, the plateauing of Dennard Scaling [14] and the slowing of Moore's Law [37] have demanded a new approach [22], that of software-defined energy efficiency.

2.1 Taxonomy of Energy Measurement

Due to the previously stated rationale, a taxonomy of energy measurement systems can be constructed by categorising solutions based on their methodologies, either hardware or software. The primary distinction lies between systems that require physical monitoring devices and those that rely on software running on the host machine.

2.1.1 Hardware-based measurement

Hardware-based measurement can be further categorised into *external devices* and *intra-node devices* [24]. *External devices* are physical power meters usually placed between the wall socket and the power supply unit, measuring the global consumption of a computing node. These are accurate, but expensive, difficult to deploy at scale, and lack visibility into specific software components [39, 26]. *Intra-node devices* are embedded equipment placed inside computing nodes which report power consumption for the whole system or for individual computing node components [24]. Similar to external power meters, these also require a financial investment and are sometimes considered to lack user-friendliness [24].

2.1.2 Software-based measurement

Software-based power meters can be further classified into three distinct categories based on how they derive and present data: *energy calculators or estimators*, *energy measurement software*, and *power profiling software* [24].

Energy calculators or estimators estimate energy consumption using mathematical models based on hardware specifications, such as Thermal Design Power (TDP), and resource usage, such as Central Processing Unit (CPU) usage and total execution time. These often assume a linear relationship between usage and power, which can lead to inaccuracies because they fail to capture hardware inefficiencies or rapid fluctuations. However, they are useful when hardware interfaces are unavailable, but do depend heavily on the quality of the model [24, 26].

Energy measurement software reads directly from hardware interfaces, such as the Intel Running Average Power Limit (RAPL) [12] or the NVIDIA Management Library (NVML) [1], to report the total energy consumed by the computing node during the execution of a program [24].

Power profiling software is similar to energy measurement software in the sense that it reads hardware sensors, but it also provides a time-series profile given its tracking of consumption moment-by-moment rather than just a total sum [24]. Because it samples at high frequencies, it allows the correlation of power spikes with specific code execution.

One perhaps interesting aspect to note is that when considering software, the aforementioned categories are not mutually exclusive. It can be the case that specific software falls into multiple categories depending on the use case, as we will next see.

2.2 Intel RAPL

Intel RAPL [12] is a hardware feature included in Intel processors that allows for the monitoring and limiting of energy consumption across different components of the CPU and attached memory [26, 50]. It serves as the bridge between the physical hardware and the software layer and is oftentimes the fundamental data source for most modern software-based energy measurement tools.

The Intel RAPL was introduced in 2011 in the Sandy Bridge architecture and provides a set of counters that report energy usage [12, 24]. These counters are accessed via 32-bit Model-Specific Registers (MSRs), which accumulate energy consumption since the processor booted in multiples of model-specific energy units, such as 15.3 microjoules (μJ) for Sandy Bridge and 61 μJ for Haswell [26]. The registers are updated approximately once every 1 millisecond (ms), depending on the specific architecture. In early generations of Sandy Bridge and Ivy Bridge, the RAPL was primarily a software model that estimated energy based on event counters, such as cache misses or instruction retires. However, starting with the Haswell architecture, the RAPL began using fully integrated voltage regulators to measure actual consumption, significantly improving its accuracy to the point where it is considered comparable to physical power meters [26, 24].

The RAPL reports energy for specific domains within the hardware. A visual overview of these domains and their hierarchy, as described by Khan et al. [26], is presented in Figure 2.1. The Package (PKG) domain represents the entire CPU socket, including cores, integrated graphics, and uncore components. Inside the PKG, the Power Plane 0 (PP0) domain represents the CPU cores, and the Power Plane 1 (PP1) domain indicates the integrated

[1] https://developer.nvidia.com/management-library-nvml

graphics. The Dynamic Random Access Memory (DRAM) domain is the random access memory attached to the memory controller. The PSys (Platform) domain was introduced in the Skylake architecture and covers the entire System on Chip (SoC), measuring the package plus other system agents. The PSys is the specific domain that often has been found to match the total consumption of the computing node reported by external wall meters [50].

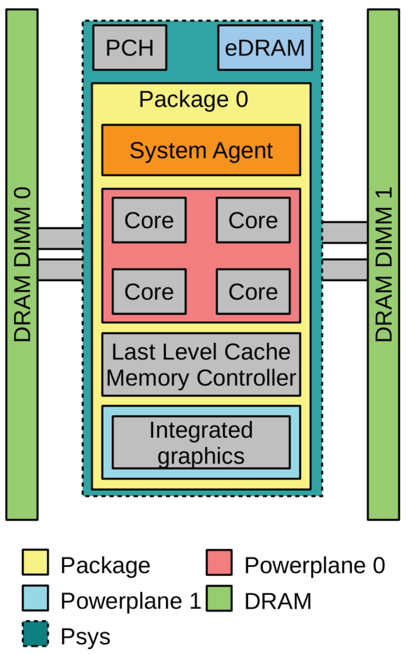


Figure 2.1: Hierarchy of Intel RAPL power domains as presented in Khan et al. [26].

In the taxonomy of energy measurement, the RAPL is the underlying mechanism that enables the category of software-based measurement. Unlike external hardware power meters which are expensive and difficult to scale, the RAPL is built into the existing hardware, requiring no extra equipment. Moreover, it has the advantage that it can isolate different power domains in a computing node. And unlike pure calculators that estimate energy based on general TDP values and CPU usage, the RAPL provides readings based on actual hardware activity and voltage measurements, offering ground truth data close to physical reality.

There are multiple ways software tools can access the information provided by the RAPL. In a Linux system, one way is to read the registers directly using the `msr` driver [2]. Another is to use the Linux Power Capping Framework `powercap` [3], which exposes RAPL data through the `sysfs` filesystem (e.g., reading files in `/sys/class/powercap/intel-rapl/`). This is a user-friendly method used by many tools because it organises domains into a readable tree structure, see Figure 2.2 for an example. However, perhaps the best option in terms of efficiency is the Linux `perf` subsystem [4], which handles counter overflows better and

[2]https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/reading-writing-msrs-in-linux.html

[3]https://docs.kernel.org/power/powercap/powercap.html

[4]https://perfwiki.github.io/main/

is often faster than reading from the MSR or through powercap, at least when it comes to servers rather than laptops [50]. Since the RAPL registers are 32-bit and can overflow quickly under load, the perf-events subsystem automatically corrects for these overflows within the kernel and returns a 64-bit integer to the user. This makes perf almost immune to overflow errors, whereas tools using MSR or powercap must poll frequently and implement manual correction logic, such as checking if the current value is lower than the previous one, which is prone to bugs. Nevertheless, Raffin et al. consider it is up to the developer to choose whether they prefer efficiency over usability or vice versa [50] . Although perf is technically superior regarding latency and system overhead, powercap offers ease of access and automatic unit conversion.

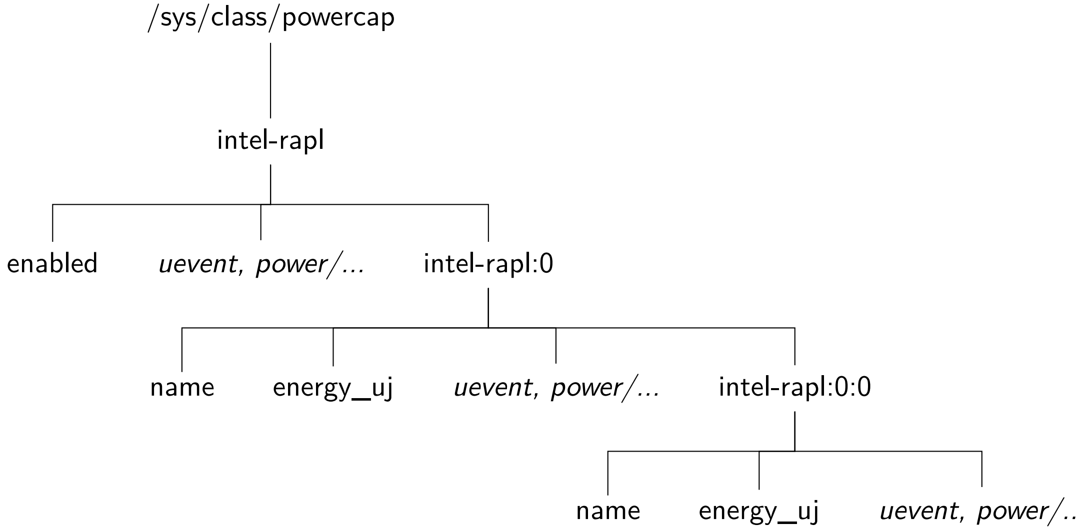


Figure 2.2: Visualisation of the RAPL sysfs directory structure on an AMD Ryzen 9 5900HX CPU using Kernel 6.14.0-36-generic. Generic kernel attributes (e.g., power/, uevent) are collapsed for clarity.

One interesting note regarding the Intel RAPL is that at the time of release, the RAPL interface permitted unprivileged access to the energy counter information, giving rise to novel software-based power side-channel attacks, known as PLATYPUS attacks [31]. Because the RAPL interface exposes values directly correlated with power consumption, which forms a low-resolution side channel, by observing changes in power consumption with a resolution of up to 20 kHz, attackers can statistically evaluate these variations to distinguish different instructions and the Hamming weights of operands and memory loads. This side channel enables an attacker to monitor the control flow of applications, infer data, and extract cryptographic keys. Due to the severe security risk that PLATYPUS poses, more specifically the ability to infer private keys, Intel acknowledged the findings and restricted RAPL MSR access in Linux systems. Starting with Linux kernel version 5.10 in 2020, access to RAPL MSRs was limited to root users. This restriction also applies, for instance, to reading powercap energy meters for non-privileged users in Intel CPUs and is also the direct reason why newer tools like PowerJoular require sudo or a system service to function.

2.3 Energy Measurement Tools

The most significant software energy measurement tools for the purpose of this thesis are to be presented in chronological order of emergence.

2.3.1 PowerAPI

PowerAPI [5] is a software library and open-source toolkit designed to monitor the energy consumption of software at the granularity of system processes. It was developed by Noureddine et al. in 2012 and emerged in the context of GreenIT research to address some specific gaps in energy monitoring [40]. At the time, accurate energy monitoring required expensive external physical wattmeters or specialised integrated circuits. These were difficult to deploy at scale and could not provide insights into specific software components. Similarly, the then existing software tools were too coarse-grained, prompting the researchers to apply the concept of performance profiling to energy, to allow developers to find energy hotspots within their source code.

PowerAPI originally appeared as the system-level component of a broader framework called e-Surgeon. This framework had a two-layer architecture. At the system layer, it had a library running at the Operating System (OS) level to estimate the power consumption of hardware components for specific processes. At the application layer, there was the Jalen Java agent which correlated the data from PowerAPI with bytecode instrumentation to determine which specific methods in the code were consuming energy. In this section, we will focus on the system layer.

Among the challenges of the software-centric approach of power consumption measurement identified by Noureddine et al., there were accuracy, overhead, granularity, and power modelling [40]. Of most interest is the latter, as the power models used for PowerAPI were among the first proposed specifically for software-based, fine-grained power consumption computation, and remain in use to this day. These models are primarily applied to estimate the power consumption of specific software components, particularly individual processes, by attributing a share of the globally measured hardware consumption to them. The fundamental function of PowerAPI is to estimate the CPU power consumed by a specific application process, denoted as $P^{PID}_{CPU}(d)$, during a given duration d. The goal is to compute the CPU percentage usage of the PID and multiply it by the overall CPU power. This is achieved using the following formula:

$$P^{PID}_{CPU}(d) = \underbrace{P_{CPU}(d)}_{\text{Overall CPU Power}} \times \underbrace{U^{PID}_{CPU}(d)}_{\text{Process CPU Usage}}$$

where
$P^{PID}_{CPU}(d)$ is the CPU power consumed by the specific process (PID) during duration d,
$P_{CPU}(d)$ is the overall CPU power consumed globally during duration d,
$U^{PID}_{CPU}(d)$ is the process CPU usage percentage during duration d.

[5] https://powerapi.org/

To calculate $P_{CPU}(d)$, PowerAPI uses models based on hardware characteristics. Early versions of PowerAPI implemented formulas to estimate overall CPU power based on the standard Complementary Metal-Oxide-Semiconductor (CMOS) equation where power (P) is related to frequency (f) and voltage (V) by $P_{f,v}^{CPU} = c \times f \times V^2$ [49]. Frequencies and voltages are both known, but to address the often unspecified nature of the constant c, the authors correlated the overall CPU power with the TDP value provided by the manufacturer. The TDP represents the maximum heat the cooling system is designed to dissipate. Then, PowerAPI estimated the overall CPU power as being proportional to the TDP [51]:

$$P_{CPU}^{f_{TDP},V_{TDP}} \approx 0.7 \times TDP$$

Using this estimated baseline, the model then accounts for Dynamic Voltage and Frequency Scaling (DVFS), recognising that power consumption is not always linearly dependent on CPU utilisation because power depends on the voltage and frequency of the processor.

Next, to calculate the process CPU usage $U_{CPU}^{PID}(d)$, PowerAPI proposes calculating the ratio between CPU time for the PID (t_{CPU}^{PID}) and global CPU time (t_{CPU}) during duration d:

$$U_{CPU}^{PID}(d) = \frac{t_{CPU}^{PID}}{t_{CPU}}(d)$$

This approach was inspired by the Linux Top[6] program [41].

Last, putting everything together, we get:

$$P_{CPU}^{PID}(d) = \underbrace{(0.7 \times TDP)}_{\text{Max Power Baseline}} \times \underbrace{\left(\frac{f \times V^2}{f_{TDP} \times V_{TDP}^2}\right)}_{\text{DVFS Scaling Ratio}} \times \underbrace{U_{CPU}^{PID}(d)}_{\text{Process Usage}}$$

equivalent to

$$P_{CPU}^{PID}(d) = \underbrace{\left(\frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2}\right)}_{\text{Hardware Constant } c} \times \underbrace{(f \times V^2)}_{\text{Current State}} \times \underbrace{U_{CPU}^{PID}(d)}_{\text{Process Usage}}$$

Newer versions of PowerAPI and its affiliated tools transitioned away from proprietary TDP-based models and began utilising Intel RAPL, NVML and other hardware interfaces for more accurate measurements.

Together with PowerJoular and JoularJX, PowerAPI and Jalen make up part of the Joular project by being the tools which established the early foundation and methodology for software-based process and method attribution.

[6] https://linux.die.net/man/1/top

2.3.2 Jalen

Jalen [7] originated as the application-level complement to the system-level energy monitoring library, PowerAPI, together making up the e-Surgeon fine-grained runtime energy monitoring framework [40]. Jalen has a software-level profiling architecture designed to monitor resource utilisation and estimate the energy consumption of running applications, specifically at the granularity of Java classes and methods. In itself, it was a proof-of-concept for fine-grained energy profiling methodology, particularly method-level energy attribution through a Java Agent connecting to a system monitor, being a direct conceptual predecessor and functional ancestor of JoularJX.

2.3.3 jRAPL

jRAPL [8] is an open-source library designed to profile the energy consumption of Java programs running on Intel CPUs. Introduced by Liu et al. in 2015, it serves as a bridge between application-level software and hardware-level energy management by providing a software wrapper to access the underlying Intel RAPL interface [32]. In the taxonomy of energy measurement, jRAPL is classified as energy measurement software.

Similar to the majority of the software presented in this chapter, jRAPL operates by accessing the MSRs of the processor. It uses the `msr` kernel module in Linux to access these registers together with the `energy metering` RAPL mode. One can use jRAPL by inserting API calls to enclose the specific block of code they wish to measure. For example, calling `EnergyCheck.statCheck()` at the beginning and end of a method allows the library to report the energy consumed during that specific execution interval. An example code snippet can be seen in Listing 2.1. Beyond just total energy, jRAPL can report consumption broken down by specific hardware components supported by RAPL, such as the CPU cores, the uncore, and DRAM. In addition to energy, it can collect CPU time, User Mode time, Kernel Mode time, and Wall Clock time.

```
1 double beginning = EnergyCheck.statCheck();
2 doWork();
3 double end = EnergyCheck.statCheck();
```

Listing 2.1: Code snippet showing how to use jRAPL, as shown in Liu et al. [32].

One major difficulty with external meters is aligning the timestamps of the meter with the execution clock of the software. jRAPL solves this because the measurement demarcation coincides exactly with the program execution, so no synchronisation is required. However, it only works in a Linux system with the `msr` kernel module installed and accessible. Moreover, it generally cannot measure energy in virtual machines (VMs) because it requires direct access to the real hardware registers, which are often hidden or virtualised in a VM environment, but that is a shortcoming of most software measurement tools based on the RAPL and not only of jRAPL. Moreover, the act of collecting energy data itself incurs a small over-

[7] https://www.noureddine.org/research/jalen
[8] https://github.com/kliu20/jRAPL

head, though this is generally orders of magnitude lower than the execution time of most experiments.

One other perhaps relevant aspect to note is that the jRAPL repository has not been maintained or updated since 2017, so it would almost certainly fail on modern systems patched against the PLATYPUS vulnerability. However, there are some active forks of jRAPL that have been updated to support newer architectures and permission handling if one specifically wants to use jRAPL.

2.3.4 PowerJoular

PowerJoular [9] is a command line multi-platform tool published by Noureddine et al. in 2021 to monitor the power consumption of software and hardware components across heterogeneous architectures, including x86 servers and ARM-based devices like the Raspberry Pi [39]. It represents an evolution in the approach to monitoring runtime power consumption.

Unlike the early versions of PowerAPI which relied heavily on TDP-based estimation formulas, PowerJoular adopts a hybrid approach depending on the hardware availability. For standard PCs and servers, it acts as a software wrapper for hardware interfaces, reading CPU power data from Intel RAPL, via `powercap`, and GPU power from the NVIDIA System Management Interface (SMI). However, for devices lacking these hardware sensors, such as Raspberry Pi, PowerJoular reverts to software-based estimation. This is where the previous comment about the categories in the taxonomy of energy measurement not being mutually exclusive becomes relevant.

PowerJoular provides power consumption data through a command-line interface and can write to CSV files, such that it enables real-time monitoring and historical retracing of power consumption. It also reports total energy consumption in Joules (J) at the end of a monitoring session. It can run automatically as a `systemd` service on Linux boot, storing data in the `/tmp` folder. This is a workaround to bypass restrictions imposed by the kernel due to the previously mentioned PLATYPUS vulnerability that complicate non-privileged user access to the `powercap` energy meters.

PowerJoular is explicitly designed to integrate with other tools, most importantly JoularJX.

2.3.5 JoularJX

JoularJX [10] is a source-code power monitoring tool in the form of a Java agent first published in 2021 by Noureddine et al., for Java applications, using the power data provided by PowerJoular [39]. It is designed specifically for monitoring the energy consumption of Java source code at the method level. JoularJX is the direct successor of Jalen, which was originally the application-level agent in the previously mentioned two-layer e-Surgeon framework, correlating low-level metrics collected by PowerAPI, to specific Java methods. JoularJX carries forward this methodology but replaces the underlying low-level data source that is PowerAPI with the specialised runtime monitoring tool PowerJoular. It adopts a similar statistical

[9] https://www.noureddine.org/research/joular/powerjoular
[10] https://www.noureddine.org/research/joular/joularjx

sampling approach, but one that is architected to take advantage of the multi-platform capabilities of PowerJoular.

JoularJX functions as a Java agent that hooks directly into the Java Virtual Machine (JVM) at startup. It does not measure power directly, but instead manages PowerJoular to monitor the PID of the Java process and consumes the resulting data. To attribute this process-level energy to specific software methods, it then monitors CPU utilisation every second (s) for each Java thread and employs statistical sampling by observing the top of the thread stack trace every 10 ms to determine how to allocate that power consumption to individual methods. Therefore, unlike earlier tools like Jalen which provided only total energy consumption at the end of a run, JoularJX can provide runtime power consumption for individual methods every second, allowing developers to detect power variations live.

Furthermore, because it relies on PowerJoular for the underlying measurements, JoularJX can profile Java applications running on heterogeneous operating systems, devices and architectures, allowing for Linux, Mac and Windows, but also Raspberry Pi for example.

2.4 The Missing Pieces

Despite the richness of the software-based energy measurement ecosystem described until this point and summarised in Table 2.1, a significant gap remains when viewing energy efficiency from the perspective of the daily developer workflow. Although tools like PowerJoular, JoularJX, and jRAPL provide robust measurement capabilities, they are primarily designed as post-mortem profiling tools or system administrator utilities, rather than interactive development aids.

Tool	Taxonomy	Language Support	Granularity	Underlying Technology	Platform	Usage	Relationship
PowerAPI [40]	Calculator/Estimator[1]	Any (Process)	Process	TDP & CMOS Models[1]	Linux	Library	Backend for Jalen
Jalen [40]	Profiling	Java	Java Method	Bytecode Instrumentation	Linux	Agent	Frontend for PowerAPI
jRAPL [32]	Measurement	Java	Code Block	RAPL via MSR	Linux	Library	Standalone
PowerJoular [39]	Measurement[2]	Any (Process)	System	RAPL via powercap, SMI & Custom Models	Linux, RPi	CLI	Backend for JoularJX
JoularJX [39]	Profiling	Java	Java Method	Statistical Sampling	Multi-OS, RPi	Agent	Frontend for PowerJoular

[1] Legacy versions relied on TDP models, but newer versions support RAPL.
[2] Acts as a Calculator/Estimator on Raspberry Pi using regression models.

Table 2.1: Comparison of software-based energy measurement tools.

In this section, we outline the specific limitations of applying these existing tools directly within an Integrated Development Environment (IDE) context and justify the architectural decision to implement a custom Java IDE plugin that interfaces directly with the Linux `powercap` framework.

The primary limitation of the current state-of-the-art is workflow friction. Tools such as PowerJoular and PowerAPI are predominantly Command Line Interface (CLI) utilities. To use them, a developer must:

1. Leave the coding environment (IDE).
2. Compile the application.
3. Construct a specific command line string, often requiring root privileges.
4. Execute the tool alongside the application.
5. Parse the resulting CSV or text output to find the relevant data.

This context switching disrupts the flow of software development. It has been found that if measuring energy consumption requires a manual, multi-step process, developers are unlikely to perform it frequently [42] and it is considered that "*best practices for energy-aware software engineering should end up being embedded in the tools, packages, and frameworks we create*" [18, p.80]. Surveyed developers agree that a green energy tool should be minimal, seamless, and transparently integrated to avoid being distracting [39]. Moreover, research shows that immediate feedback is fundamental for behavioural change [69, 42]. The gap, therefore, is the lack of immediate, in-situ energy feedback that lives where the code is written.

One next logical counter-argument might be to ask why one should not simply build an IDE plugin that wraps PowerJoular, JoularJX, or jRAPL. And while it is theoretically possible, this approach introduces some major architectural disadvantages.

First, wrapping an external tool like PowerJoular creates many dependency problems. The user would not only need to install the IDE plugin but also manually install PowerJoular and its dependencies on their system and ensure it is in the system PATH. This raises the barrier to entry significantly compared to a self-contained plugin.

Next, if the plugin were to invoke a CLI tool like PowerJoular for every measurement, it would need to spawn a subprocess, wait for it to initialise, and then parse its standard output. This introduces latency and synchronisation issues, particularly when trying to measure short-lived code snippets.

Then, tools like JoularJX and Jalen function as Java Agents. Although they are powerful, attaching an agent requires modifying the JVM startup arguments (e.g., `-javaagent:joularjx.jar`). Integrating this into the test runner of an IDE transparently is complex and prone to conflicts with other agents, such as debuggers or code coverage tools. Furthermore, instrumentation can introduce overhead that skews results for very small code blocks [40].

Lastly, unlike jRAPL, which requires the developer to modify their source code to add energy probes, polluting the code with measurement logic, a direct-access plugin can run the code as is, applying the measurement externally but tightly coupled to the execution timeline.

To bridge these gaps, the proposed solution is a plugin that bypasses the middleware layer, that is PowerAPI or PowerJoular, and interfaces directly with the Linux `powercap` subsystem. The framework exposes energy metrics as virtual files (e.g., `energy_uj`). Reading a file is a lightweight system call compared to the overhead of inter-process communication required to talk to an external tool. This allows the plugin to sample energy at a high frequency with minimal impact on the system. By controlling the reading of the sensor directly,

the plugin can synchronise measurement with the code execution. It can snapshot the energy counter immediately before the method starts and immediately after it ends, isolating the energy usage of the code from the surrounding system noise. As for energy attribution, it can use a similar approach as the one presented by Noureddine et al. when proposing the PowerAPI power model [40]. That is, it can attribute a precise share of the global CPU energy to the code execution based on its relative CPU time and resource usage.

However, let us not forget or favourably omit the fact that `powercap` now also requires permissions to access the energy information to mitigate security risks. But here, the gap between current tools and the proposed solution lies in the managing of said permissions. External tools require the user, for example, to manage sudo sessions in a terminal. An IDE plugin can handle this gracefully by checking permissions on startup, prompting for a password once, or guiding the user to set a persistent `chmod` rule, essentially directing the developer to configure the environment once.

In conclusion, while the existing ecosystem provides excellent tools for system-level monitoring and long-running application profiling, a gap exists for a lightweight, low-dependency, and high-granularity tool integrated directly into a developer's edit-compile-run cycle. The specific architectural design and implementation details of this proposed plugin, including how it addresses the permission and synchronisation challenges outlined above, are to be presented in Chapter 3.

Chapter 3

Implementation

This chapter details the design and engineering of the energy measurement plugin proposed in previous chapters. To this end, the solution is architected as a decoupled system consisting of a standalone Java application that wraps JShell to provide a REPL environment with energy measurement capabilities, and a UI (User Interface) in the form of an IntelliJ plugin that captures code snippets, invokes the backend, and displays results. The complete source code for this implementation can be found in the accompanying replication package [35].

The design and implementation of the energy measurement plugin were conducted on specific hardware and software configurations to ensure consistency and reproducibility, and are displayed in Table 3.1. Since accessing MSR drivers is a major security risk and therefore restricted on the Windows OS unless working in a signed kernel mode, we decided all development should be conducted on a Linux OS, namely Ubuntu. However, Linux access to RAPL MSRs was also eventually restricted starting with kernel version 5.10 in 2020. The workaround we ultimately implemented was to modify the permissions of the relevant files while in root mode to allow for unprivileged access to the `powercap` Intel RAPL energy counters. Nonetheless, this is not a permanent solution, as it means we need to perform this modification on each system boot and we also need to pay attention not to open any potentially malicious scripts during operation. A more lasting solution could be setting up a `udev` rule to ensure the settings persist after a system reboot.

3.1 Backend

The backend component of the system is a standalone Java application that wraps the JShell API to provide a REPL environment augmented with energy measurement capabilities. This section describes its architecture and the energy measurement mechanism, together with the decisions that shaped their implementation.

Component	Specification
Processor (CPU)	AMD Ryzen™ 9 5900HX
Topology	8 Cores, 16 Threads
Clock Speed	3.3 GHz Base, 4.6 GHz Boost
Cache	20 MB Total
Instruction Set	x86_64
Memory (RAM)	16 GB DDR4
Operating System	Ubuntu 24.04.3 LTS
Kernel	Linux 6.14.0-37-generic
Desktop Environment	GNOME
IDE	IntelliJ IDEA 2025.3.2
Java Runtime	OpenJDK 21.0.9

Table 3.1: Development environment specifications.

3.1.1 Architecture Overview

The backend is structured as a Maven [1] project with the package organisation seen in Figure 3.1. The design follows a layered approach where the `EnergyREPL` class serves as the entry point, delegating code execution to `JavaREPLEngine`, input processing to `CommandHandler`, and output styling to `ConsoleFormatter`. The `JavaREPLEngine` class is used through `CommandHandler` inside `EnergyREPL` and wraps the JShell API introduced in Java 9 to execute code snippets. The `EnergyMonitor` class contains all energy measurement logic, providing methods that the `JavaREPLEngine` invokes before and after code execution. It returns its measurements encapsulated in an `EnergyMetrics` object which is then attached to an `ExecutionResult` inside the `JavaREPLEngine` and ultimately consumed by the `CommandHandler` instance in `EnergyREPL` to format and display the energy cost of the executed code. A visualisation of the backend workflow is available in Figure 3.7b.

The backend supports two main modes of operation, interactive and single-shot, selectable via command-line arguments. Interactive mode is the default when no arguments are provided. Here, the application starts a persistent REPL session where users can enter multiple code snippets sequentially in a CLI. Variables, methods, and classes defined in one snippet remain available in subsequent evaluations. Some basic imports (e.g., `util`, `io`, `time`, `math`) are provided by default, but more specific ones need to be provided by the user. This mode is suitable for exploratory programming and iterative experimentation, with an example shown in Figure 3.2. Alternatively, single-shot execution mode is activated by the `--execute` flag. Together with some context flags (`--project-cp`, `--project-dir`), the single-shot execution mode is designed for integration with the IntelliJ plugin. An example

[1] https://maven.apache.org/

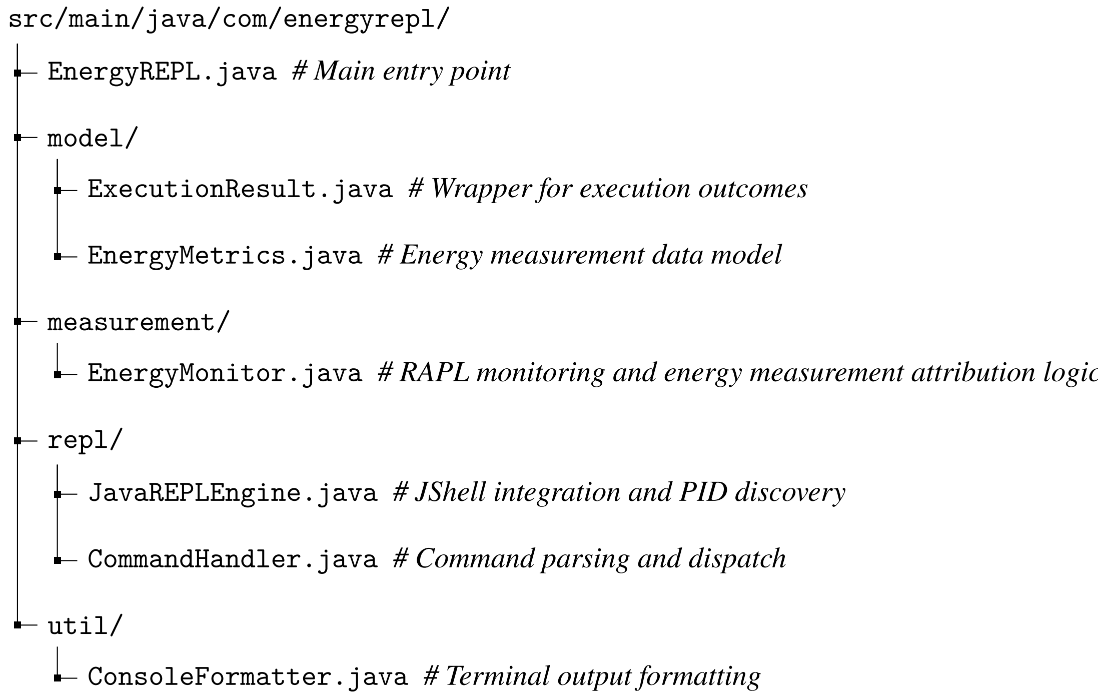


Figure 3.1: The backend structured as a Maven project and its respective package organisation.

single-shot backend invocation is provided in Listing 3.1. In this mode, the backend executes a single code snippet, outputs the result with energy metrics, and terminates. An example compact single-line metrics output returned by the backend in either of the two modes is available in Listing 3.2. Each output typically contains: the process identifier (PID) of the host application (i.e., the CLI or the IDE running the plugin), the PID of the JShell execution engine, the average system-wide CPU load during the window the code was running (using the Operating System and Hardware Information Java library [2]), the memory allocated during the execution of the snippet, the estimated average power the system was drawing while the code ran, an estimated energy consumption, and the real measured energy consumption. For the purpose of our overall tool, the real measured energy is the only relevant metric. Further implementation details regarding the other metrics are not provided, as they were only used during development as placeholders or for debugging purposes.

A critical aspect for the upcoming energy attribution discussion is knowing and correctly identifying what it is that we aim to measure. One initial oversight was mistakenly measuring the energy consumption of the main REPL process instead of the energy consumption of the code snippets. This flaw first became apparent after evaluating code snippets with known different energy profiles, only to get the same approximate consumption each time. Another obvious tell was querying the PID together with code snippet and getting the same value back in each run, meaning we were targeting the consumption of some constant process and

[2] https://github.com/oshi/oshi

```
java -jar /path/to/energy-repl.jar \
  --execute "import java.util.stream.*; IntStream.range(1, 1_000_000).sum();" \
  --project-cp "/path/to/out/production/my-module:/path/to/dependencies/*" \
  --project-dir "/path/to/my-project"
```

Listing 3.1: Example single-shot backend invocation used by the IDE integration.

```
[Main PID: 182744, Worker PID: 182761, CPU: 34.7%, Mem: 6.4MB, Power: 32.371W, Energy: Est: 0.214J, Real: 0.063J]
```

Listing 3.2: Example compact single-line metrics output returned by the backend.

```
  Java Energy-Aware REPL v1.0.0
  Real-time Energy Consumption Measurement

Type /help for available commands or /exit to quit.

System: System Information:
  Processor: AMD Ryzen 9 5900HX with Radeon Graphics
  Physical Cores: 8
  Logical Cores: 16
  Max Frequency: 4892 MHz
  Total Memory: 15.0 GB

java[1]> System.out.println("Hello world!");
Hello world!
[Main PID: 81301, Worker PID: 81320, CPU: 0.0%, Mem: 11.4MB, Power: 15.004W, Energy: Est: 2.500J, Real: 0.019J]

java[2]> int x = 1;
[Main PID: 81301, Worker PID: 81320, CPU: 0.0%, Mem: 1.9MB, Power: 15.001W, Energy: Est: 2.005J, Real: 0.019J]

java[3]> int y = 1;
[Main PID: 81301, Worker PID: 81320, CPU: 0.0%, Mem: 0.8MB, Power: 15.000W, Energy: Est: 1.944J, Real: 0.000J]

java[4]> int z = x + y;
[Main PID: 81301, Worker PID: 81320, CPU: 0.0%, Mem: 0.9MB, Power: 15.000W, Energy: Est: 2.019J, Real: 0.000J]

java[5]> System.out.println(z);
2
[Main PID: 81301, Worker PID: 81320, CPU: 0.0%, Mem: 2.0MB, Power: 15.001W, Energy: Est: 2.124J, Real: 0.000J]

java[6]> /quit

Exiting Energy REPL. Goodbye!
```

Figure 3.2: Example use of the backend interactive mode of operation.

not that of our target snippet. Therefore, we realised JShell executes code in a separate JVM process which is distinct from the main REPL process. To attribute energy consumption correctly, we must discover the PID of the worker. Our solution takes advantage of the fact that code evaluated by JShell runs inside the worker process. By evaluating a snippet that queries its own PID, the `JavaREPLEngine` obtains the identifier of the worker. This approach ensures that subsequent energy measurements track the process where user code actually executes, rather than the orchestrating REPL process.

3.1.2 Energy Measurement

The `EnergyMonitor` class implements direct hardware readings via Intel RAPL to provide real measurements. To briefly revisit the RAPL discussed in Section 2.2, it is an Intel technology that provides energy consumption data through hardware counters. Although we are working with an AMD processor, starting with the first Zen architecture in 2017, AMD processors, such as our Ryzen 9 5900HX, feature a RAPL-compatible interface [50]. Moreover, on Linux systems, the `powercap` subsystem abstracts the underlying hardware manufacturer and exposes the energy counters as virtual files in `/sys/class/powercap/` for both Intel and AMD processors. Accordingly, in our implementation we read from the package-level energy counter which reports cumulative energy consumption in µJ since system boot. This package-level scope was deliberately chosen over a core-only measurement because evaluating Java snippets, particularly those involving algorithmic memory access patterns and object allocation, requires capturing the energy consumed by the integrated memory controller and Last Level Cache. To measure the consumption of a specific code snippet, the monitor records the counter value before and after execution, computing the difference.

One of the challenges encountered in computing the energy delta was occasionally obtaining a negative value. To investigate and identify the source of this fault, we looked back at the underlying technology (Section 2.2) and at the energy measurement implementation of other tools (Section 2.3). Even widely-used utilities like EnergiBridge are reported to sometimes return negative energy values [3]. However, after recalling that the Intel RAPL energy counter is stored in µJ in a 32-bit register, it becomes apparent how overflow can occur during extended system operation. To handle this exception in our solution, we read the maximum counter range from `/sys/class/powercap/intel-rapl/intel-rapl:0/max_energy_range_uj` and apply a wraparound correction, as shown in Listing 3.3.

Another important aspect is that RAPL measures energy at the CPU package level, including all processes running on the system. To attribute a portion of this energy to a specific code snippet execution, we use a CPU time-based proportional allocation model inspired by PowerAPI [40], and previously explained in Section 2.3.1. To give a high-level overview of how it is applied in our context, the system first reads the hardware RAPL counter to measure the total system energy consumed by the entire chip during the execution window. We track the process CPU time (i.e., user mode and kernel mode), which represents the actual time the CPU spent executing instructions for the target PID. Then, we calculate the total available CPU time by multiplying the wall-clock duration of the execution by the number

[3] `https://luiscruz.github.io/course_sustainableSE/2024/p1_measuring_software/g3_energy_consumption_ides.html`

of logical cores. This product represents the maximum potential processing time available on the machine during that window. Finally, we calculate an attribution ratio by dividing the specific process CPU time by this total available time. This ratio is applied to the total RAPL energy measurement to filter out system noise and isolate the energy consumed specifically by the target code. Pseudocode for the described workflow is provided in Listing 3.4, while the general formulas are the following:

$$U_{core}^{PID}(d) = \frac{t_{CPU}^{PID}(d)}{t_{wall}(d) \cdot N}$$

$$E_{CPU}^{PID}(d) = \underbrace{\Delta E_{RAPL}(d)}_{\text{Package Energy Delta over } d} \times \underbrace{U_{core}^{PID}(d)}_{\text{Process CPU-Time Usage}}$$

$$\Delta E_{RAPL}(d) = \begin{cases} E_{end} - E_{start}, & E_{end} \geq E_{start} \\ (E_{max} - E_{start}) + E_{end}, & E_{end} < E_{start} \ \wedge \ E_{max} > 0 \\ -1, & \text{otherwise} \end{cases}$$

where
$t_{CPU}^{PID}(d)$ is the CPU time consumed by the target process during duration d,
$t_{wall}(d)$ is the wall-clock duration of the measurement window,
N is the number of logical CPU cores,
$U_{core}^{PID}(d)$ is the process CPU-time usage,
$\Delta E_{RAPL}(d)$ is the total package energy delta measured by RAPL over d,
$E_{CPU}^{PID}(d)$ is the energy attributed to the target process.

3.2 Frontend

The frontend component is an IntelliJ IDEA plugin that provides integration of energy measurement into the developer workflow. Rather than requiring developers to leave their IDE and interact with command-line tools, the plugin allows them to select code directly in the editor and measure its energy consumption with a single IDE action [4]. This section describes the plugin architecture and user interface design.

We note that both a CLI and a Graphical User Interface (GUI) were options considered for the frontend. Although a CLI satisfies the need for a lightweight tool, and a GUI can provide convenient visualisations, we deem both to be too disruptive to the developer workflow. As previously mentioned in Section 2.4, we believe a plugin is the type of interface that best addresses the shortcomings of state-of-the-art energy measurement tools.

[4] Here, the term *action* is used as defined by the IntelliJ Platform Plugin SDK.

```
// 1. CAPTURE START STATE
// Read global hardware energy counter (microjoules)
long startSystemEnergy = readRaplEnergy();
// Read CPU time specifically used by this process (user + kernel)
long startProcessCpuTime = getProcessCpuTime(targetPid);
// Read the current wall time
long startWallTime = System.nanoTime();

// --- EXECUTE CODE SNIPPET ---

// 2. CAPTURE END STATE
long endWallTime = System.nanoTime();
long endSystemEnergy = readRaplEnergy();
long endProcessCpuTime = getProcessCpuTime(targetPid);

// 3. CALCULATE DELTAS
long wallClockDuration = endWallTime - startWallTime;
long processCpuDuration = endProcessCpuTime - startProcessCpuTime;

// Calculate total energy consumed (handling overflow)
long totalSystemEnergyDelta = calculateRaplDelta(startSystemEnergy
    , endSystemEnergy);


// 4. CALCULATE ATTRIBUTION RATIO
// Calculate total CPU time available across all cores
// Formula: Wall Time * Number of Logical Cores
int coreCount = getLogicalProcessorCount();
double totalAvailableCpuTime = wallClockDuration * coreCount;

// Calculate the share of the CPU resources this process used
double attributionRatio = processCpuDuration /
    totalAvailableCpuTime;

// Safety check: Ensure ratio is between 0.0 (0%) and 1.0 (100%)
attributionRatio = Math.max(0.0, Math.min(1.0, attributionRatio));

// 5. CALCULATE FINAL ATTRIBUTED ENERGY
// Apply the ratio to the global energy measurement
double attributedEnergyJoules = (totalSystemEnergyDelta *
    attributionRatio) / 1_000_000.0;
```

Listing 3.4: Pseudocode showing the implemented energy attribution logic, inspired by Noureddine et al. [40].

```
// Inputs derived from reading /sys/class/powercap/.../energy_uj
long startEnergy = readRaplEnergy();
// ... Code executes here ...
long endEnergy = readRaplEnergy();

// Constant derived from /sys/class/powercap/.../
    max_energy_range_uj
long maxRange = readRaplMaxRange();
long deltaEnergy;

// LOGIC FLOW
IF (endEnergy >= startEnergy) THEN
    // Usual Case: The counter increased
    deltaEnergy = endEnergy - startEnergy

ELSE IF (maxRange > 0) THEN
    // Overflow Case: The counter hit the max and reset to 0
    // Formula: (Distance from Start to Max) + (Distance from 0 to
        End)
    deltaEnergy = (maxRange - startEnergy) + endEnergy

ELSE
    // Error Case: Overflow occurred but max range is unknown
    deltaEnergy = -1

END IF
```

Listing 3.3: Pseudocode showing the handling of RAPL counter overflow with the aim of accurate energy measurement.

3.2.1 Architecture Overview

The plugin follows the standard IntelliJ Platform plugin architecture, consisting of actions, tool windows, and extension points declared in the plugin configuration file. The implementation consists of two primary classes. `RunEnergyAnalysisAction` handles user interaction, code extraction, backend invocation, and result parsing. `EnergyToolWindowFactory` creates and manages the persistent console for displaying results. The capabilities are declared in `plugin.xml`, which registers the action and tool window with the IDE.

First, the `EnergyToolWindowFactory` class creates the persistent console. `RunEnergyAnalysisAction` is the main controller, from extracting the selected code to displaying results. When triggered, the action first retrieves the user's text selection from the active editor, compiles the project, and launches the backend in the background. Finally, it parses the backend's structured output using regular expressions and displays the formatted report to the user in the console window, as seen in Figure 3.3. A visualisation of the frontend workflow is presented in Figure 3.7a. Typically, if the execution does not encounter any errors or warnings, each run output contains the following two sections: "ENERGY ANALYSIS" and

```
Energy Monitor
ENERGY ANALYSIS
Snippet:
  | int x = 10 + 20 * 3;
  | ...
  | System.out.println("Area: " + result);
Run output:
  | Area: 78.53981633974483

ENERGY METRICS
  • Real:     0.047 J   (Actual measured consumption)

  Process IDs: Main[53224] Worker[53265]
```

Figure 3.3: Example measurement output in the IDE tool window, as shown in light theme.

"ENERGY METRICS". Under "ENERGY ANALYSIS", we find "Snippet", where we see a preview containing the selected code in order to facilitate the identification of measurement provenience while making relative comparisons, and the "Run output", where the user can check if the evaluation went as expected. The "Run output" catches all print statements and error outputs if any, otherwise it remains empty. Under "ENERGY METRICS" we find the real energy value in J alongside a message detailing if it is the actual measured consumption. Otherwise, the user is informed if the snippet execution skipped the RAPL counter increase or if the system crashed. The PIDs are also included for debugging purposes. An overview of all possible warning and error messages is available in Table 3.2.

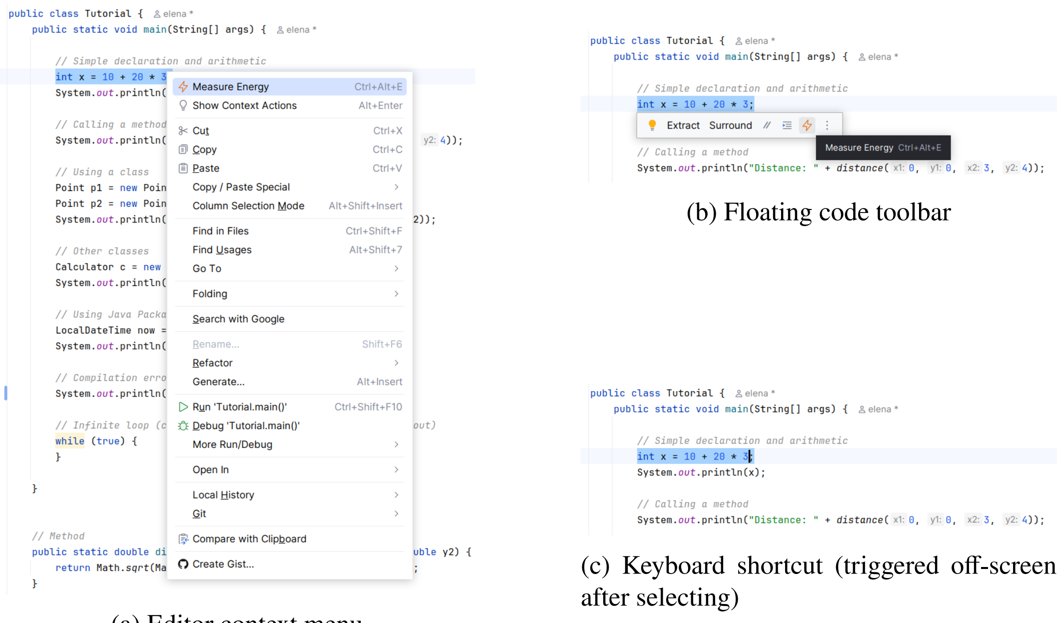
Type	Message Displayed	Trigger Condition	Content
Error	`Compilation failed with [N] error(s). Fix them before running energy analysis.`	Syntax errors or missing dependencies found during module build.	Error
Error	`Execution timed out after 5 minutes.`	Process exceeds maximum allowed time.	Snippet, Error
Error	`ERROR: [System Exception Messages]`	IO errors, missing JAR[1] or process start failure.	Snippet, Run output, ENERGY METRICS
Error	`RAPL hardware sensor is not available or permission denied. Please enable counter reading to use this plugin: sudo chmod o+r /sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj`	RAPL sensor is inaccessible or lacks read permissions.	Snippet, Run output, Error
Warning	`Compilation aborted.`	Build process is cancelled by the system/user.	Warning
Warning	`Could not resolve module. Skipping compilation. Results may use stale classes.`	`ModuleUtil` cannot find a module for the file.	Warning, Snippet, Run output, ENERGY METRICS
Warning	`Execution Cancelled.`	User manually cancels the measurement.	Snippet, Warning

[1] The frontend connects to the backend using a .env path to the compiled JAR file.

Table 3.2: Overview of all possible error and warning messages.

One other important aspect to note is that the energy measurement action can be triggered through three mechanisms once the desired code snippet is selected: the editor context menu (right-click, see Figure 3.4a), the floating code toolbar (see Figure 3.4b), and the keyboard shortcut `Ctrl+Alt+E` (see Figure 3.4c). This varied access ensures developers can invoke energy measurement in whatever manner better fits their workflow.

Although seemingly simple and straightforward, the development of the frontend did not come without its own challenges. We made the console display a preview of the executed snippet, and we noticed instances where the tool was using outdated versions of the code instead of the most recently selected version. To prevent the execution of stale data, we integrated automated file saving and triggered a background compilation of the module prior to execution, ensuring that the measurement safely aborts if the code does not compile. We also encountered a significant usability issue where triggering the energy measurement action would cause the entire IDE to freeze until the measurement completed. This freeze was resolved by adding support for the IntelliJ non-modal dialog progress bar to appear during execution, featuring a cancel button (see Figure 3.5) which allows users to manually abort processes. Moreover, after accidentally attempting to measure a snippet containing an infinite loop, we realised we need more termination mechanisms. As a secondary fail-safe to complement the manual cancellation button, an automatic execution timeout of five minutes was introduced.



(a) Editor context menu

(b) Floating code toolbar

(c) Keyboard shortcut (triggered off-screen after selecting)

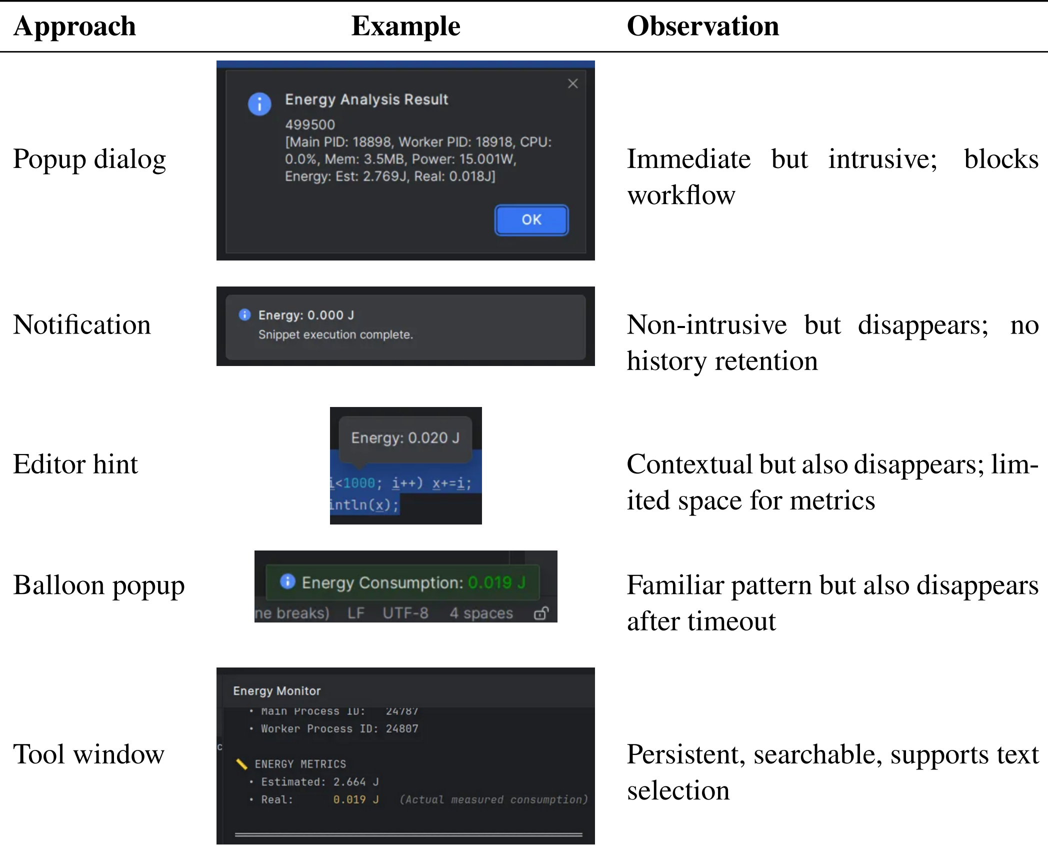
Figure 3.4: Primary interaction entry points for triggering the energy measurement action within the IntelliJ IDEA editor, as shown in light IDE theme.

3.2.2 UI Design Evolution

The user interface underwent several iterations before arriving at the final design. Initial iterations explored various feedback mechanisms available in the IntelliJ Platform Plugin SDK [5], which are presented in Table 3.3 with their respective examples, good points, and drawbacks. The final design offers several advantages aligned with and driven by the IntelliJ

[5]https://plugins.jetbrains.com/docs/intellij/welcome.html

Platform UI Guidelines [6]. The proposed approach uses a dedicated tool window containing a `ConsoleView` component. This implementation ensures that execution results remain persistent to facilitate comparative analysis across multiple measurement cycles. The integration inherently supports standard text manipulation features, including clipboard operations and the `Ctrl+F` search shortcut. To ensure a cohesive user experience, the interface mirrors the visual structure of the standard `Run` and `Debug` tool windows and employs `JBColor` attributes to dynamically adapt to both light and dark IDE themes. An example of all the user interface components together, as shown in IntelliJ, can be seen in Figure 3.5.



Approach	Example	Observation
Popup dialog	Energy Analysis Result 499500 [Main PID: 18898, Worker PID: 18918, CPU: 0.0%, Mem: 3.5MB, Power: 15.001W, Energy: Est: 2.769J, Real: 0.018J] OK	Immediate but intrusive; blocks workflow
Notification	Energy: 0.000 J Snippet execution complete.	Non-intrusive but disappears; no history retention

Editor hint	Energy: 0.020 J
	i<1000; i++) x+=i;
	intln(x);
		Contextual but also disappears; limited space for metrics

Balloon popup	Energy Consumption: 0.019 J ne breaks) LF UTF-8 4 spaces	Familiar pattern but also disappears after timeout
Tool window	Energy Monitor • Worker Process ID: 24807 ENERGY METRICS • Estimated: 2.664 J • Real: 0.019 J (Actual measured consumption)	Persistent, searchable, supports text selection

Table 3.3: Comparison of user interface approaches with visual examples in dark IDE theme.

3.3 Putting Everything Together

The preceding sections have described the backend and frontend as independent components, each with its own responsibilities and design rationale. This section synthesises these components into a unified system, explaining how they communicate and detailing the complete measurement pipeline from user action to displayed result.

[6]https://plugins.jetbrains.com/docs/intellij/ui-guidelines-welcome.html

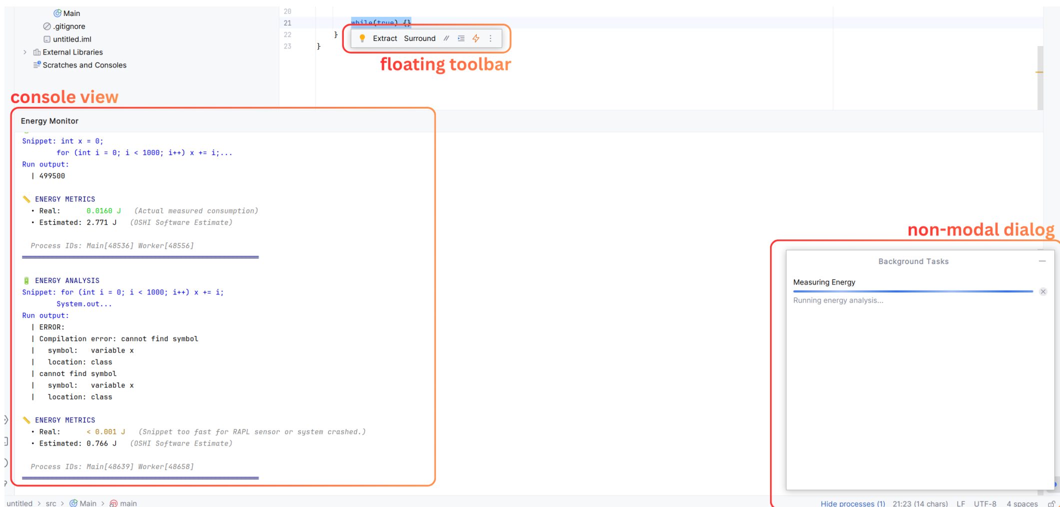


Figure 3.5: The plugin user interface components, showing the floating toolbar for action triggering, the non-modal progress dialog for background execution, and the persistent console view displaying execution results and energy metrics in light IDE theme.

3.3.1 Integration Architecture

The integration between the IntelliJ plugin and the REPL backend follows a subprocess invocation pattern. Rather than maintaining a persistent connection to a long-running backend service, the plugin spawns a fresh instance of the backend JAR for each measurement request. This design decision prioritises simplicity and robustness over performance as each measurement operates in complete isolation, eliminating concerns about state corruption, memory leaks, or zombie processes accumulating over extended IDE sessions.

In the very early iterations of the system, running relatively simple code succeeded because the backend automatically provided a few baseline libraries (e.g., `util`, `io`, `time`, `math`). However, when attempting to execute more complex snippets that relied on project-specific packages or external libraries, the execution would fail. To address this issue, one of the first solutions we implemented was to automatically append the import statements from the top of the selected code's parent file into the execution context, using IntelliJ's Program Structure Interface (PSI). We also later introduced IntelliJ's `OrderEnumerator` to gather the entire compiled output of the module, along with all dependencies (e.g., Apache Commons), to construct a complete classpath to provide to the backend. Furthermore, we had a specific situation where we attempted to run code that required file input from different locations within the project, and so the relative paths failed to resolve. To address this issue, we decided to also pass the project's root directory to the backend.

Therefore, in the final design, communication between the components occurs through the following two channels. In the forward direction, the plugin passes information to the backend via command-line arguments. The `--execute` flag signals single-shot mode, the code snippet is passed as a string argument, the `--project-cp` flag provides the classpath necessary for resolving project-specific types, and the `--project-dir` passes the project

root directory. In the reverse direction, the backend writes its results to standard output in a structured format that the plugin parses using regular expressions. Although this text-based communication protocol is simple, it proves sufficient for the current use case and avoids the complexity of binary serialisation or inter-process communication frameworks.

We present an example of what the data could look like at different points in the communication flow in Figure 3.6. When a user highlights a code fragment within a source file (Figure 3.6a), the surrounding package and file-level imports are prepended to the user's selection to create a compilable execution context (Figure 3.6b). This constructed string is then injected into the `--execute` argument of the backend invocation command, alongside the resolved module classpath and project directory (Figure 3.6c). In this example, the user attempts to add an integer to a `String` list. The backend executes the snippet, catches the JShell compilation failure, and returns a structured response (Figure 3.6d) containing both the error diagnostics and the energy data gathered during the failed execution attempt, which the plugin subsequently parses and displays.

To verify the robustness of our system and to check if it can handle the complexities of development, the system was subjected to a suite of over 40 distinct code scenarios, available in our replication package [35]. These code snippets were designed to check both the plugin's context extraction capabilities, and the backend's energy measurement under varying loads. The validation suite confirmed that the system supports the following capabilities:

- *Core features*: execution of basic expressions, boolean logic, string operations, and complex control flows (e.g., nested loops, `while` loops).
- *Advanced features*: Java Collections and Streams, recursive method calls, and localised class definitions within the selected snippet.
- *Context and dependency resolution*: cross-file dependencies within the same package, instantiation of nested/inner classes, and resolution of relative file I/O operations (e.g., `java.nio.file.Path.of(...)`) using the injected project directory context.
- *Edge cases*: runtime exceptions (e.g., division by zero, `NullPointerException`), syntax typos, and high-volume standard output.

3.3.2 Data Flow

The complete data flow for a single energy measurement traverses both frontend and backend components, as illustrated in Figure 3.7. The process begins when the user selects code in the editor and triggers the measurement action. The plugin's `RunEnergyAnalysisAction` class extracts the selected text along with contextual information, including import statements and the module compiled classpath. This information is assembled into a command-line invocation of the backend JAR. The plugin spawns this process on a background thread to avoid blocking the IDE event dispatch thread, displaying a progress indicator that allows the user to cancel measurements. The backend's `EnergyREPL` class parses the arguments and delegates to `JavaREPLEngine` for execution. Within the backend, the measurement proceeds in three phases. First, the `EnergyMonitor` captures the initial state by reading the RAPL energy counter and recording the process CPU time for the JShell worker process.

```
package com.example;

import java.util.List;
import java.util.ArrayList;

public class MyClass {
    // User selects lines:
    List<String> list = new
        ArrayList<>();
    list.add(1);
}
```

(a) Source file with selection.

```
import com.example.*;
import com.example.MyClass
    .*;
import static com.example.
    MyClass.*;
import java.util.List;
import java.util.ArrayList;

List<String> list = new
    ArrayList<>();
list.add(1);
```

(b) Constructed string.

```
java -jar /path/to/energy-repl.jar \
   --execute "import com.example.*; import com.example.MyClass.*;
       \
import static com.example.MyClass.*; import java.util.List; \
import java.util.ArrayList; List<String> list = new ArrayList<>();
    list.add(1);" \
   --project-cp "/path/to/out/production/my-module:/path/to/
       dependencies/*" \
   --project-dir "/path/to/my-project"
```

(c) Constructed backend invocation command.

```
ERROR:Compilation error: incompatible types: int cannot be
    converted to java.lang.String
[Main PID: 71290, Worker PID: 71310, CPU: 0.0%, Mem: 2.0MB, Power:
    15.001W, Energy: Est: 0.403J, Real: 0.000J]
```

(d) Terminal output showing compilation error.

Figure 3.6: Communication flow of the energy measurement process. The plugin extracts the user's selection (a), constructs a context-aware string (b), and passes it as an argument to the backend via a command-line single-shot invocation (c). The backend executes the snippet and returns a structured output string (d).

Second, the `JavaREPLEngine` evaluates the code snippet, potentially processing multiple statements if the input contains imports followed by executable code. Third, the `EnergyMonitor` captures the final state and computes the attributed energy using the CPU time-based proportional allocation model. The resulting `ExecutionResult` object, containing both the code output and the `EnergyMetrics`, is formatted and written to standard output. The plugin captures this output, parses it to extract the relevant metrics, and formats the information for display in the tool window console, which retains all measurement results from the current session, unless cleared by the user.

3.3.3 User Interaction Flow

From the user's perspective, the interaction flow is designed to be minimally disruptive to the normal development workflow, as shown in Figure 3.8. The developer writes or navigates to code of interest, selects the relevant lines, and triggers the measurement through their preferred method, either the context menu, the floating toolbar, or the keyboard shortcut `Ctrl+Alt+E`. A progress indicator appears briefly while the measurement executes, after which the results appear in the dedicated tool window which persists across measurements, accumulating results that the developer can scroll through, search, or copy. This persistence is particularly valuable for comparative analysis, where a developer might measure several implementations of the same algorithm to identify the most energy-efficient approach. Since the console interface is similar to the IntelliJ standard `Run` and `Debug` windows, it reduces the learning curve and integrates naturally into existing workflows.

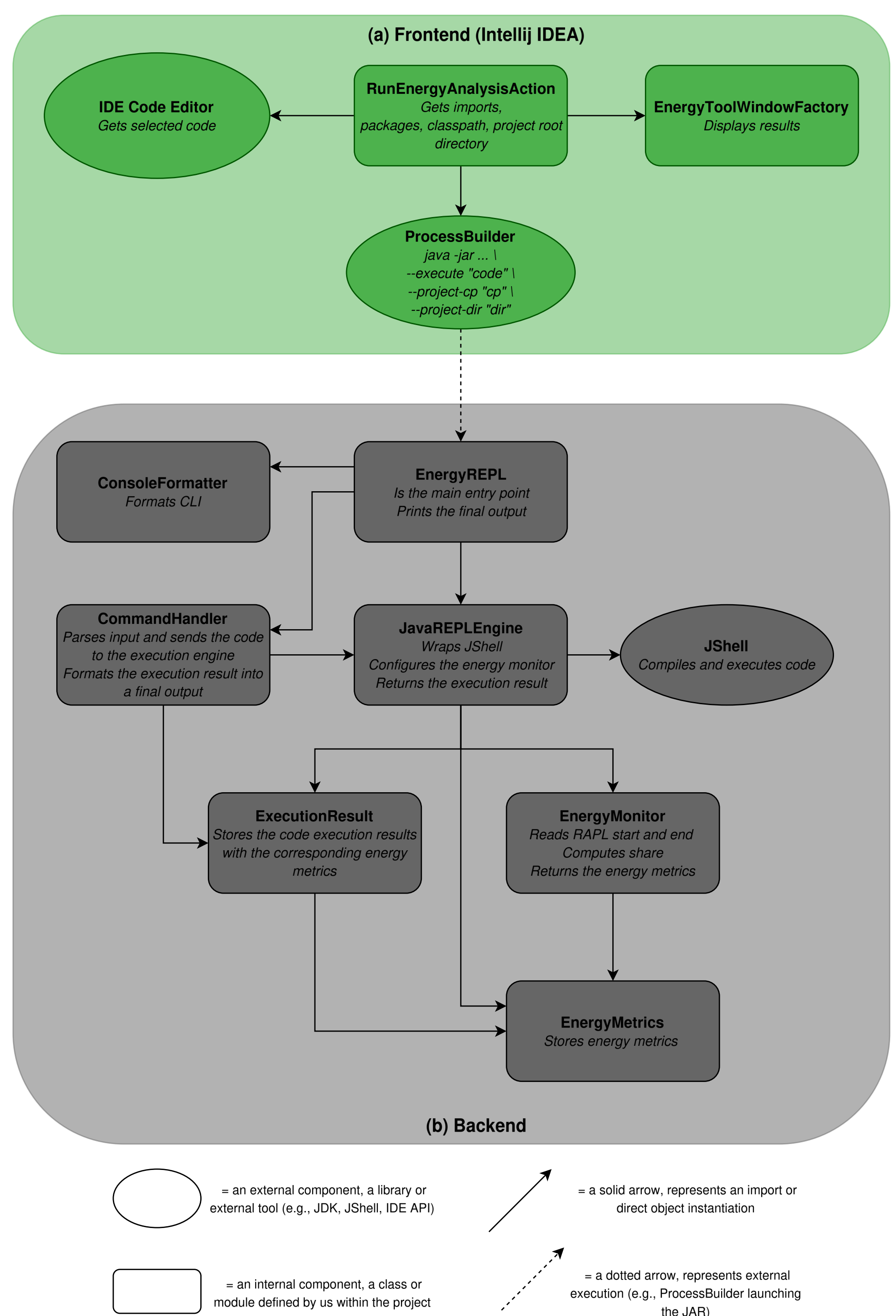


Figure 3.7: High-level data flow diagram showing the interaction between the IntelliJ IDEA frontend and the backend components. (a) The flow of data in the IntelliJ IDEA frontend components, starting from the code editor selection. (b) The energy monitoring backend, showing the data flow from the start of the process to the collection of energy metrics.

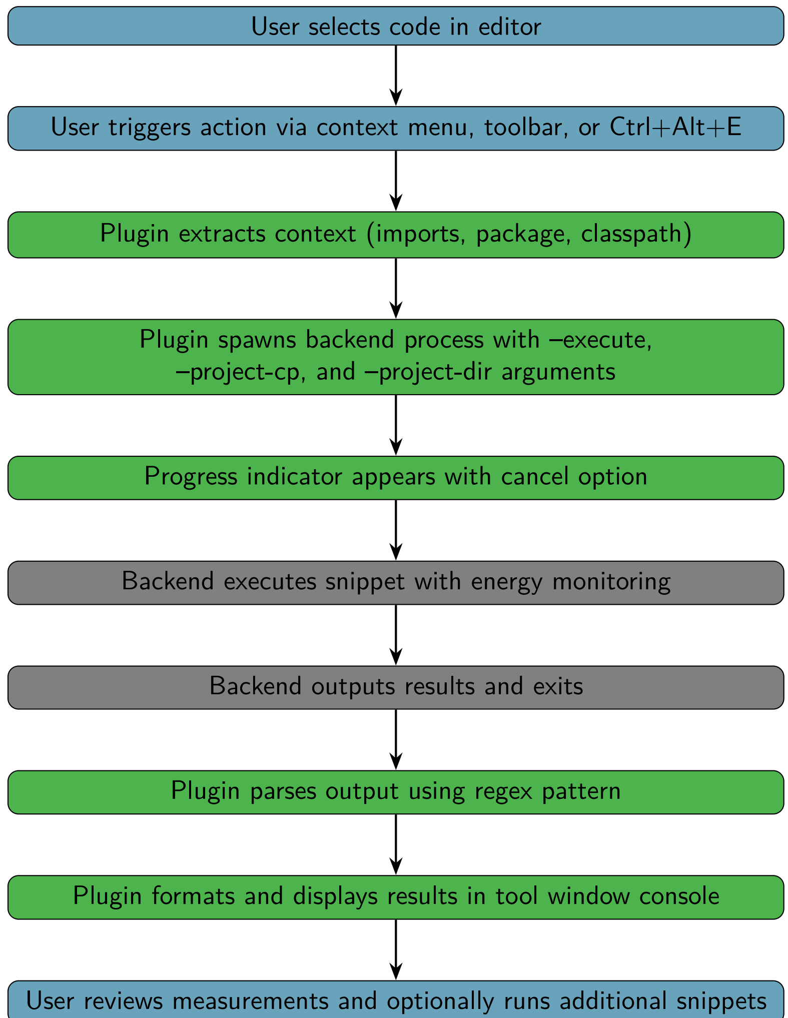


Figure 3.8: The execution workflow of the energy analysis process, illustrating the sequence of steps from user input to the display of results.

Chapter 4

Study Setup

This chapter outlines the methodological approach used to evaluate the proposed energy measurement plugin. The methodology is divided into two primary components: verification and validation.

4.1 Verification

The primary objective of the verification part is to answer the general questions: “Did we build the product *right*? Does the tool measure what it claims to measure?”. We aim to show that the tool can successfully detect statistically significant energy differences between distinct implementations. For our study setup we start from the assumption that better runtime performance usually translates to better energy consumption as well [47]. Following this assumption, we can then compare pairs of semantically equivalent but syntactically different programming solutions that are documented to show improvements in runtime performance, be it that the performance gain comes from improved algorithmic complexity (e.g., $O(n)$ versus $O(n \log n)$) or from hardware resource optimisation (e.g., improving spatial locality to reduce cache misses). We therefore investigate the following research question:

> **RQ1:** Does the proposed energy measurement tool provide sufficiently reliable measurements such that statistically significant energy consumption differences between semantically equivalent but syntactically distinct Java code snippets can be detected on the tested hardware configuration?

To answer this question comprehensively, we further decompose it into four sub-questions that guide the design of our verification experiment:
Baseline (RQ1.1): Does the local experimental setup consistently replicate expected execution time discrepancies between the chosen faster and slower implementations?
Sensitivity (RQ1.2): Can the tool detect statistically significant differences in energy consumption between the paired implementations?
Magnitude (RQ1.3): Are the detected energy savings practically meaningful?

Correlation (RQ1.4): Does the energy consumption measured by the tool scale with the execution time across the different implementations?

4.1.1 Data Collection

Publicly available datasets (e.g., CodeComplex [1]) were taken into consideration, but were found not to have sufficient metadata (e.g., runtimes, varied time complexity) to support the experiment. Therefore, to establish a baseline for comparison, we use a ground truth of problem solutions sourced from LeetCode [2] with reported performance discrepancies given by the platform's Submissions Runtime analysis. To build our dataset, a total of 30 problems are manually [3] selected and for each problem, a pair of two distinct implementations is chosen: one reported by LeetCode as *better-performing* (or *faster*) in terms of runtime, and one reported as *worse-performing* (or *slower*). Because LeetCode does not make the input for their Submissions Runtime analysis publicly available, for each pair, we choose specific input adhering to the respective problem's description constraints. The aim of the input is to demonstrate the different approach of the two chosen solutions, as shown through execution time. Following recommendations for energy measurement research [30, 11, 10], each code snippet (containing input declaration, solution run, and result printing) is run 30 times using the plugin, resulting in a total of 1800 runs across the 30 pairs. After each code snippet run, the plugin reports two primary metrics which we record: time (in s) [4] and energy consumption (in J). An example code snippet from one of the chosen problems together with corresponding plugin output is available in Figure 4.1. We note that the "ENERGY METRICS" results cover input declaration, solution run, and printing, while the time measurements reported through the plugin's "Run output" only cover the solution run. This is due to the fact that for the plugin to work on a code snippet, it requires both input declaration and solution run (see Chapter 3), whereas for the time measurement we are only aiming to showcase the time discrepancies given by implementation differences across the solutions, therefore not needing to record the time taken to declare input. Lastly, the reason we also include the printing of execution time in the code snippet is to facilitate data collection. One potential risk is that the inclusion of printing can skew the absolute energy measurement results, but we mitigate this by only being interested in relative comparisons and by including these prints across all runs. The entire data collection pipeline is illustrated in Figure 4.2.

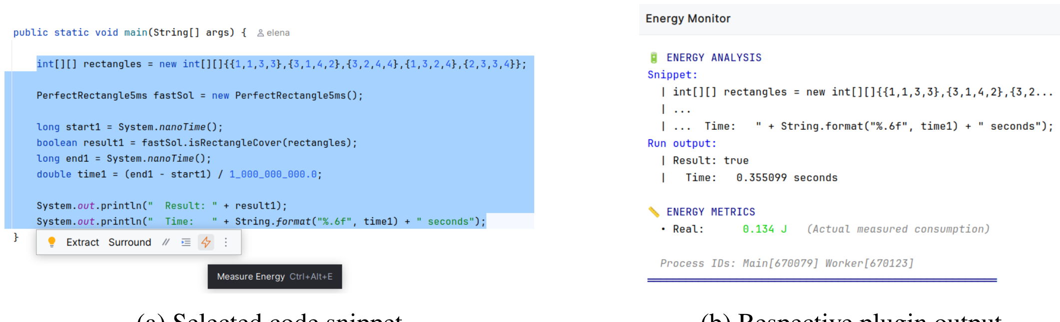
To make the measurements as reliable as possible, we aim to conduct all runs under consistent experimental conditions. This includes keeping the background programmes running on the machine, the battery percentage, and the charging state the same. Furthermore, measurements are recorded sequentially, in batches of 10, to allow for cooldown periods. More on the topic of measurement reliability is available in Section 7.4. To facilitate transparency and reproducibility, the complete dataset comprising all 1,800 measurements, alongside the evaluated problem implementations, is available in the our replication package [35].

[1] https://huggingface.co/datasets/codeparrot/codecomplex

[2] https://leetcode.com/

[3] Manual selection was preferred to automation in order to adhere to the LeetCode Terms of Service (TOS).

[4] While reported and analysed in seconds to maintain adherence to the International System of Units (SI) alongside Joules, the raw execution time is captured at the nanosecond level using Java's `System.nanoTime()` to guarantee precision without data loss.



(a) Selected code snippet

(b) Respective plugin output

Figure 4.1: Selected code snippet used for the energy and time measurement of the *fast* solution for the *PerfectRectangle* problem alongside a respective measurement output given by the plugin.

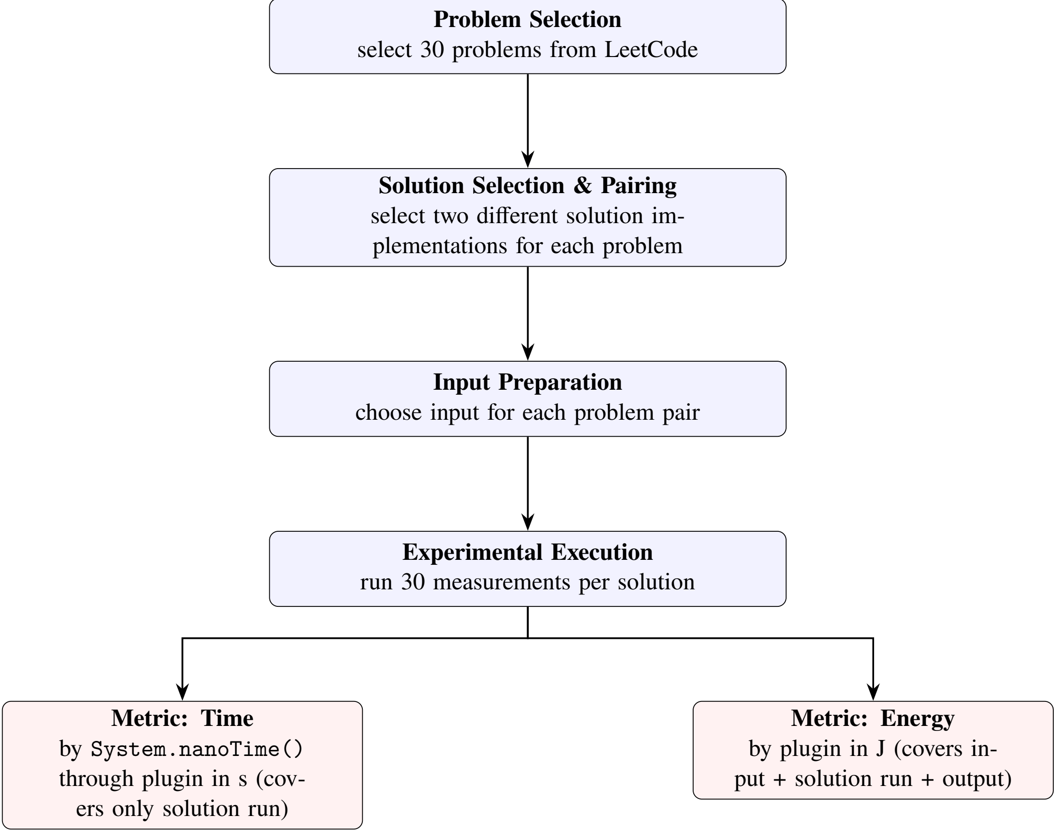


Figure 4.2: The data collection pipeline from problem selection to metric capture.

4.1.2 Experimental Design

The overall reasoning for the experiment follows a logical progression that starts by establishing the aforementioned baseline and then moves through four specific steps to reach a conclusion. The first step is to perform a statistical test on the execution time. This is done to address *Baseline (RQ1.1)* by answering whether the chosen inputs actually show the expected measurable difference between the better and worse implementations. Once the time difference is shown, the same statistical test is applied to the energy data. This step addresses *Sensitivity (RQ1.2)* to evaluate whether the proposed plugin actually reports lower energy consumption for the faster algorithms. If significant energy differences are detected, we address *Magnitude (RQ1.3)* by analysing the effect size to determine if the achieved energy savings are practically meaningful. Finally, a correlation test is conducted to address *Correlation (RQ1.4)* and check if the measured energy accurately reflects the performed work. The question here is whether the energy consumption scales with the execution time across the dataset. We conclude with a final logical conclusion:

> ***IF*** the time differences are significant (*RQ1.1*, shows that the inputs chosen for the case study code solutions show the implementation difference),
> ***AND*** the energy differences are significant (*RQ1.2*, shows that the plugin successfully detects energy differences across the different solutions in a problem pair),
> ***AND*** the effect size is meaningful (*RQ1.3*, shows that the achieved energy savings are practically meaningful),
> ***AND*** the energy correlates with time (*RQ1.4*, shows that energy reflects the optimisation work),
> ***THEN*** it can be stated that the plugin correctly measures energy consumption (*RQ1*).

4.1.3 Statistical Analysis

In order to conduct the four steps outlined in the experimental design, the data is processed through a statistical pipeline to determine the most appropriate tests.

We begin by stating that in this experimental setup, the execution time serves as the independent variable, whilst energy consumption acts as the dependent variable. The next step in the statistical analysis involves determining the distribution of the collected data to guide the selection of appropriate evaluation methods. Dudley [15] mentions that "*given a sample $X_1, \ldots, X_n$ of n real-valued observations, the Shapiro–Wilk test [57] is a test of the composite hypothesis that the data are i.i.d. (independent and identically distributed) and normal, i.e. $N(\mu, \sigma^2)$ for some unknown real μ and some $\sigma > 0$. This test of a parametric hypothesis relates to nonparametrics in that a lot of statistical methods (such as t-tests and analysis of variance) assume that variables are normally distributed. If they are not, then some nonparametric methods may be needed.*" We therefore use the Shapiro-Wilk test [57] to evaluate the null hypothesis that a given sample originated from a normally distributed population and we do so for each of the 60 solutions, for both energy and time measurements, totalling 120 distributions. For each one, if the resulting p-value is less than the chosen alpha level ($p < 0.05$), the null hypothesis is rejected, indicating the data is not normally

distributed. We also establish a strict constraint that both the time and energy distributions for a given problem pair must be normally distributed to justify the use of parametric statistical methods. However, the methodological choice between parametric and non-parametric tests is not based solely on this normality test, as it has been observed that additional data characteristics must be evaluated, including the presence of heavy tails, skewed distributions, outliers, and the overall sample size [16, 11, 10]. If the analysis reveals that the majority of the 30 problem pairs fail to satisfy these constraints, the methodology dictates defaulting to non-parametric tests for all pairs to maintain analytical consistency.

If the constraints are met, we proceed with parametric methods. One option could be to use the Two-Sample t-test, also known as the independent or Student's t-test [65], to compare the means of two groups to determine if they are significantly different. Specifically, for each of the 30 problem pairs, we conduct a one-tailed comparison of the means. We evaluate the 30 time measurements of the *better* implementation against the 30 time measurements of the *worse* implementation to determine if the former is significantly lower, addressing *Baseline (RQ1.1)*, and then repeat this process for the 30 energy measurements, addressing *Sensitivity (RQ1.2)*. Walpole et al. [71] note that this test is appropriate "*if the scientist involved is willing to assume that both distributions are normal and that* $\sigma_1 = \sigma_2 = \sigma$". To check for the equality of variances we use the F-test [17, 58]. If it is the case that we have normally distributed data but unequal population variances we can use Welch's t-test [72]. One other observation is that because the measurement samples collected during the experimental design are independent of one another, paired methods like the Paired t-test [65] which is a special case of the Student's t-test [36], are explicitly excluded, as it would be statistically incorrect to assume the measurements come in dependent pairs.

If the constraints are not met, we proceed with non-parametric methods. We can use the Mann–Whitney U test [33] to compare the ranks of two groups to determine if they are significantly different. Mirroring the parametric approach, we perform a one-tailed comparison of the ranks for both time, addressing *Baseline (RQ1.1)*, and energy, addressing *Sensitivity (RQ1.2)*, across all 30 problem pairs. This allows us to determine if the distribution of measurements from the *better-performing* implementation is stochastically less than that of the *worse-performing* implementation. With this test, the data must also be checked for duplicates, as duplicate values can alter the rank calculations and skew the final results.

To address *Magnitude (RQ1.3)*, beyond identifying statistical significance, we analyse the magnitude of the differences between the implementations using either Cohen's d [8], Glass's delta [19] or Cliff's delta [6]. Cohen's d gives the difference between two means divided by a standard deviation for the data assuming normality and equal variances. Glass's delta also assumes normality but is meant for unequal variances and uses only the standard deviation of the second group which is taken as a control group. Lastly, Cliff's delta measures how often values in one distribution are larger than those in a second distribution, while not making any assumptions regarding the shape or spread of the data. For Cliff's delta we use the thresholds defined by Romano et al. [52] for the following categories: *negligible* ($|\delta| < 0.147$), *small* ($0.147 \leq |\delta| < 0.33$), *medium* ($0.33 \leq |\delta| < 0.474$), *large* ($|\delta| \geq 0.474$). As with the significance testing, these effect sizes are calculated independently for each of the 30 problem pairs. Because the baseline energy consumption and algorithmic complexity vary drastically across different LeetCode problems, a highly sensitive

measurement environment might detect a statistically significant difference ($p < 0.05$) that is actually negligible. Calculating the isolated effect size per pair aims to quantify whether the energy savings achieved by the *better-performing* implementation are actually meaningful.

Finally, to address *Correlation (RQ1.4)* and validate the relationship between energy and time, we perform a correlation analysis. Unlike the previous verification steps which compare distributions within isolated problem pairs, this analysis evaluates the trend across the entire dataset. To achieve this, we calculate the mean or median execution time and mean or median energy consumption for each of the 60 distinct implementations (comprising the 30 *better* and 30 *worse* variants). The correlation is then applied to these 60 aggregated data points to determine how reliably energy scales with time. We use the Pearson correlation coefficient [4, 61, 46] for continuous, normally distributed data exhibiting a linear relationship without significant outliers. Conversely, we use the Spearman rank correlation coefficient [59] for ordinal, ranked, or non-normally distributed data, as it measures monotonic relationships and is significantly more robust to outliers due to using ranks.

To recapitulate, we aim to address *RQ1* and its respective sub-questions by employing the statistical methodology outlined above and visualised in Figure 4.3. As mentioned, the pipeline begins by assessing the underlying characteristics of the collected samples, testing for normal distribution with the Shapiro-Wilk test and accounting for outliers and skewness. Should the majority of the problem pairs fail to meet these constraints, we choose analytical consistency by defaulting to non-parametric methods, specifically the Mann-Whitney U test (for *RQ1.1* and *RQ1.2*), Cliff's delta (for *RQ1.3*), and Spearman rank correlation (for *RQ1.4*). Conversely, if the distributions are demonstrably normal, the evaluation proceeds to parametric methods, dividing between the Two-Sample (Student's) t-test and Welch's t-test (for *RQ1.1* and *RQ1.2*), respectively between Cohen's d and Glass's delta (for *RQ1.3*), depending on the equality of population variances, and proceeding with Pearson's correlation (for *RQ1.4*). This progression aims for mathematical systematicity to be maintained throughout the verification process.

4.2 Validation

The primary objective of the validation part is to answer the general questions: "Did we build the *right* product? Is the tool useful for developers?". We aim to show whether the proposed tool achieves its goal of increasing energy awareness and fitting into the developer workflow by conducting a user study where we investigate the following research question:

> **RQ2:** How does the proposed energy measurement tool affect developers' ability to understand, reason about, and make decisions regarding the energy efficiency of Java code snippets?

To answer this question comprehensively, we further decompose it into five sub-questions that guide the design of our user study:
Interpretation (RQ2.1): Do developers correctly interpret the metrics produced by the tool?
Identification (RQ2.2): Does the tool help developers identify more energy-efficient imple-

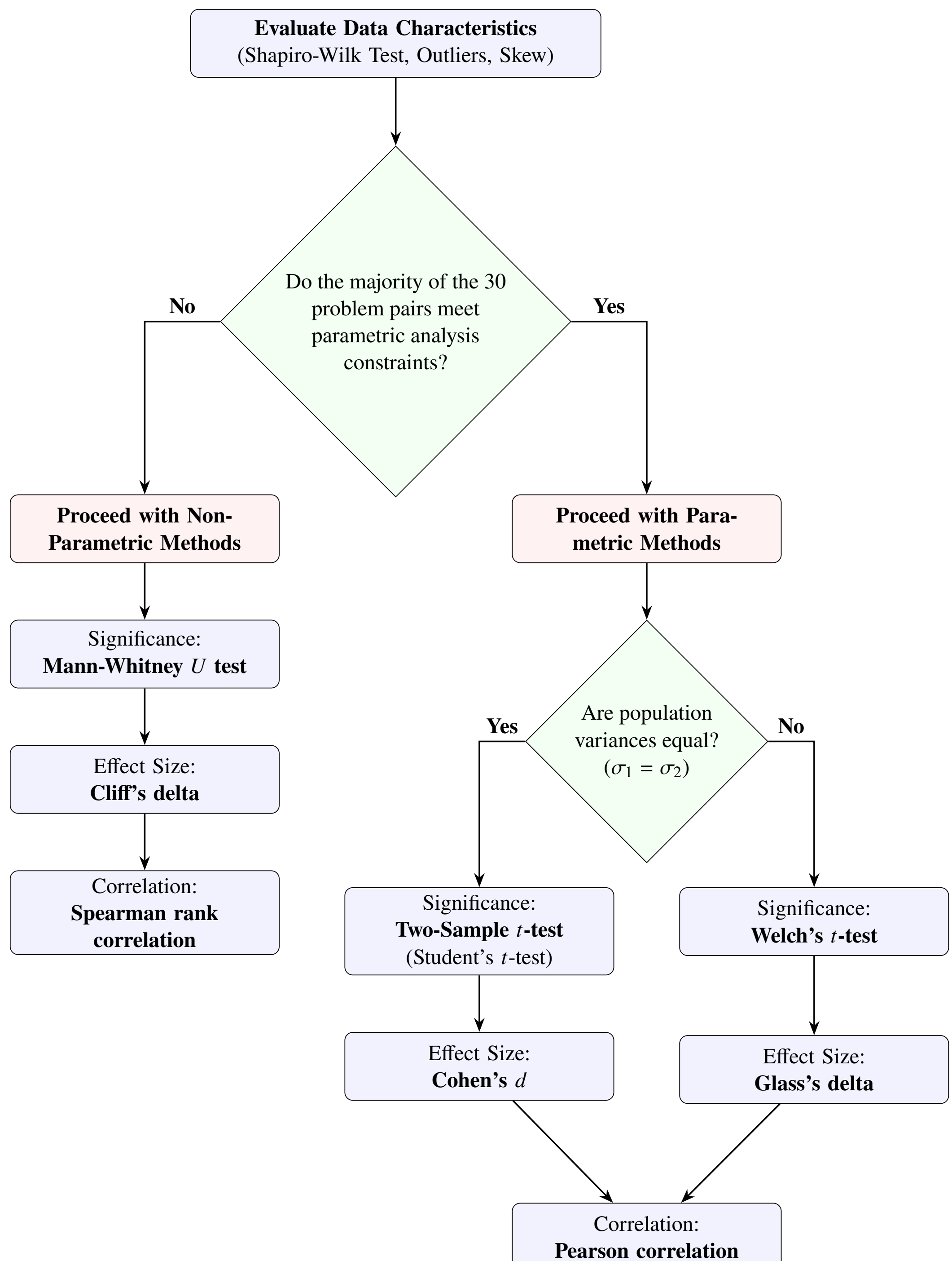


Figure 4.3: Decision flow for selecting the appropriate statistical tests based on data distribution constraints.

mentations compared to reasoning from source code alone?
Integration (RQ2.3): How do developers integrate the tool into their reasoning process?
Usability (RQ2.4): What usability problems appear when developers use the tool in IntelliJ?
Confidence (RQ2.5): Does the tool make developers more confident in their reasoning regarding energy efficiency?

To address these questions, we design a mixed-methods, within-subject user study with task-based evaluation. The study captures both quantitative metrics, such as identification accuracy and self-reported confidence levels, and qualitative insights gathered through think-aloud protocols, screen and audio recordings, and a post-task questionnaire. Each session consists of a background questionnaire, a pre-recorded tutorial with live explanations, the participant experimenting with the plugin, some code energy analysis tasks, and a post-task questionnaire, all of which we elaborate upon next.

We further mention that since our study involves human research subjects, research ethics were considered and incorporated into the research design, and approval for the study to proceed was given by the TU Delft Human Research Ethics Committee (HREC)[5].

4.2.1 Study Design

The target cohort for the study consists of 20 to 30 participants, recruited from our own personal network. For the collected data to accurately reflect the intended user base of the tool, participants are favoured to fit a specific profile of being Computer Science students or working software developers with prior experience in the Java programming language and, preferably, familiarity with the IntelliJ IDEA environment.

Every session takes place in person in the presence of the researcher and on the setup provided by the researcher. During each session, we switch between three contexts: a Microsoft form where the participant fills in their responses (see Appendix D), a video player where we show a pre-recorded tutorial video [6], and IntelliJ where the code energy analysis tasks and interaction with the plugin happen. All the resources used are available in our replication package [35]. Each session is designed to last 30 to 40 minutes and follows a structured sequence:

1. *Introduction (5 min)*: the researcher explains the study purpose, gives a brief overview of the plugin, provides instructions on the think-aloud protocol, and has the participant sign a consent form (see Appendix E).

2. *Background questionnaire (2 min)*: the participant reports their programming experience, Java and IntelliJ proficiency, their occupation, and previous familiarity with performance or energy optimisation in the Microsoft form.

3. *Tutorial (5 min)*: the researcher gives a video demonstration together with live explanations, showing how to properly select code, trigger the energy analysis, interpret the metrics, and avoid common misuses of the tool.

[5]https://www.tudelft.nl/en/about-tu-delft/strategy/integrity-policy/human-research-ethics

[6]https://www.youtube.com/watch?v=2E2z0PDPe6w

4. *Plugin experimentation (5 min)*: after being walked through the tutorial, the participant is given the code used in the tutorial within the IntelliJ environment such that they can familiarise themselves with the plugin and ask any questions they might have.

5. *Tasks (10-20 min)*: the participant completes the four code analysis tasks explained below.

6. *Post-task questionnaire (5 min)*: the participant answers four 1-5 Likert-scale questions regarding whether the tool was useful, easy to interpret, and practical for real-world development, alongside open-ended questions about what they found confusing or would like to improve and a chance to share any other feedback they might have.

The tutorial is conducted using a pre-recorded video demonstration alongside a live explanation. The video introduces participants to the IDE and a specific `Tutorial` class, which contains a `main` method, a `distance` method, and a `Point` class. We then cover multiple specific steps and features in detail. We start by selecting a simple variable declaration to demonstrate the three ways to trigger the energy measurement: clicking the lightning icon on the floating toolbar, right-clicking (using a mouse or touchpad), or pressing the `Ctrl + Alt + E` keyboard shortcut. We show the progress bar appearing upon measurement trigger and how to use the tool window toggle to open and close the interface. The researcher explains the four components of the measurement output: the snippet, the run output, the energy metric, and the process IDs. It is mentioned that the tool only displays the beginning and end of the selected code snippet to save space in the console when dealing with long selections. Participants are taught that they must check the run output to verify the code's expected behaviour, as it captures any print statements or errors. This is practically demonstrated by running the previous variable initialisation, but this time including a print statement of the variable in the selection. We then proceed with a `distance` method call where we explain that we only need to select the method call and not its declaration body too. Similar to the method, we have the `Point` class example, but the difference here is that because we need to call its own distance method through an instantiation of the class, we need to declare at least two `Point` entities. The video also demonstrates how to successfully measure a class declared in a different part of the project, as well as how to use standard Java packages like `java.time.LocalDateTime`. The next part of the tutorial deals with common misuse of the plugin and system limitations. The video shows a deliberate typo to demonstrate how a compilation error appears. The researcher explains that these errors persist in the tool window to allow for relative comparisons between different runs unless the user explicitly clears them. Once the typo is fixed, the measurement is shown to proceed successfully if triggered again. To demonstrate cancellation, an infinite `while(true)` loop is selected. The video shows that hovering over the ⊗ symbol on the progress bar expands it to display a "Cancel" message. Clicking this ⊗ symbol stops the execution and produces a specific warning message in the tool window alongside the snippet. The same infinite loop is run again, but the video is sped up 30 times to demonstrate that the tool will automatically time out after five minutes, resulting in a specific error in the console. To demonstrate improper selections, the video shows what happens if a user tries to print a variable without including the variable's declaration in their highlighted selection. It also shows the error

caused by improperly highlighting a line, emphasising once again why users must check the run output rather than relying solely on the energy metrics. The video portion concludes by showing how to wipe the console clean by right-clicking and selecting "Clear output". Following the video, the researcher switches to a live instance of IntelliJ IDEA. The participant is given control of the `Tutorial` class and is encouraged to experiment with the tool, modify the code, and ask any questions before beginning the actual study tasks.

For the task-based evaluation, a set of four distinct algorithmic problems was formulated to measure the impact of the tool on developer reasoning. We choose to include only four problems in the task part, such that the study does not become exhausting for the participants, while also getting enough insight for analysis [68]. For each problem, we created two semantically equivalent but syntactically different Java solution implementations, available in our replication package [35]. These pairs were specifically chosen to represent common, recognisable performance scenarios in Java development, ensuring a clear ground truth regarding which solution is more energy-efficient. The four tasks include:

- *Task 1: String manipulation*: comparing basic string concatenation in a loop (see Figure 4.4a) versus using `StringBuilder` (see Figure 4.4b) to append 100000 words.
- *Task 2: Collection lookup*: comparing an $O(N)$ lookup using a nested loop over an `ArrayList` (see Figure 4.4c) versus an $O(1)$ lookup using a `HashSet` (see Figure 4.4d) to identify duplicates among 100000 integers.
- *Task 3: Recursive vs. iterative execution*: comparing a recursive approach versus (see Figure 4.4e) an iterative programming approach (see Figure 4.4f) to calculate the 45th Fibonacci number.
- *Task 4: Memory access patterns*: comparing column-by-column (see Figure 4.4g) versus row-by-row traversal (see Figure 4.4h) when multiplying two 1500×1500 matrices, showcasing the impact of cache locality and of how Java stores matrices.

As for how the task evaluation is actually carried out, we apply a within-subject, mixed-methods experimental design. The term *mixed methods research (MMR)* is described by Storey et al. as "*a research approach where multiple methods are used to collect, analyze, and integrate both qualitative and quantitative data to address a research problem and produce novel insights*" [62]. This type of research approach is preferred as it has been found to "*bring insights on the nuanced nature of the problem and/or solution being studied*" [62] in a software engineering context. Then, for the within-subject design part, we decide that every participant should be exposed to both conditions of the study, which serve as our independent variable: completing a task *without the tool* relying solely on reading the source code, and completing a task *with the tool* by using the plugin to measure energy consumption. This approach is selected over a between-subject design where participants only see one condition because it requires a smaller number of participants to achieve sufficient statistical power [13] and allows individuals to act as their own baselines when comparing the dependent variables of confidence and accuracy. A significant risk in within-subject studies is the learning or carryover effect, where a participant's experience in the first task influences their performance in the second task [68]. To mitigate this, we employ AB/BA

```java
public class Pair1A {
    public static void main(String[] args) {

        // FIRST OPTION
        //build a result string by concatenating 100,000 words
        String result = "";
        for (int i = 0; i < 100000; i++) {
            result += "word" + i;
        }

    }
}
```

(a) Task 1A: String concatenation in a loop

```java
public class Pair1B {
    public static void main(String[] args) {

        //SECOND OPTION
        //build a result string by concatenating 100,000 words
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 100000; i++) {
            sb.append("word").append(i);
        }
        String result = sb.toString();

    }
}
```

(b) Task 1B: Using StringBuilder

```java
public class Pair2A {

    public static void main(String[] args) {

        //FIRST SOLUTION
        //check whether a list of 100,000 integers contains any duplicates
        List<Integer> numbers = new ArrayList<>();
        for (int i = 0; i < 100000; i++) numbers.add(i % 90000);

        boolean hasDuplicate = false;
        for (int i = 0; i < numbers.size(); i++) {
            for (int j = i + 1; j < numbers.size(); j++) {
                if (numbers.get(i).equals(numbers.get(j))) {
                    hasDuplicate = true;
                    break;
                }
            }
        }

    }
}
```

(c) Task 2A: ArrayList nested loop

```java
public class Pair2B {

    public static void main(String[] args) {

        //SECOND SOLUTION
        //check whether a list of 100,000 integers contains any duplicates
        List<Integer> numbers = new ArrayList<>();
        for (int i = 0; i < 100000; i++) numbers.add(i % 90000);

        Set<Integer> seen = new HashSet<>();
        boolean hasDuplicate = false;
        for (Integer n : numbers) {
            if (!seen.add(n)) {
                hasDuplicate = true;
                break;
            }
        }

    }
}
```

(d) Task 2B: HashSet

```java
public class Pair3A {
    public static void main(String[] args) {

        //FIRST OPTION
        //calculate the Fibonacci number at index 45
        //a Fibonacci number is part of a specific integer
        // sequence where each number is the sum of the two
        // preceding ones, starting from 0 and 1
        fib(45);

    }

    public static int fib(int n) {
        if (n <= 1) return n;
        return fib(n-1) + fib(n-2);
    }
}
```

(e) Task 3A: Recursive Fibonacci

```java
public class Pair3B {
    public static void main(String[] args) {

        //SECOND OPTION
        //calculate the Fibonacci number at index 45
        //a Fibonacci number is part of a specific integer
        // sequence where each number is the sum of the two
        // preceding ones, starting from 0 and 1
        int a = 0, b = 1;
        for (int i = 0; i < 45; i++) {
            int temp = a + b;
            a = b;
            b = temp;
        }

    }
}
```

(f) Task 3B: Iterative programming

```java
public class Pair4A {
    public static void main(String[] args) {

        // FIRST OPTION
        //multiply two 1500×1500 matrices
        int n = 1500;
        int[][] A = new int[n][n];
        int[][] B = new int[n][n];
        int[][] C = new int[n][n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = i + j;
                B[i][j] = i - j;
            }
        }

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    C[i][j] += A[i][k] * B[k][j];

    }
}
```

(g) Task 4A: Column-by-column traversal

```java
public class Pair4B {
    public static void main(String[] args) {

        //SECOND OPTION
        //multiply two 1500×1500 matrices
        int n = 1500;
        int[][] A = new int[n][n];
        int[][] B = new int[n][n];
        int[][] C = new int[n][n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = i + j;
                B[i][j] = i - j;
            }
        }

        for (int i = 0; i < n; i++)
            for (int k = 0; k < n; k++)
                for (int j = 0; j < n; j++)
                    C[i][j] += A[i][k] * B[k][j];

    }
}
```

(h) Task 4B: Row-by-row traversal

Figure 4.4: Experimental Java implementation pairs used for task-based evaluation, representing common performance and energy-efficiency scenarios. The highlights (the yellow-shaded regions) pinpoint the lines of code that represent the scenario being tested.

crossover design with counterbalancing [20]. Therefore, the four algorithmic problems are divided into two task blocks. Block A contains *Task 1: String manipulation* and *Task 2: Collection lookup* problems, while Block B contains *Task 3: Recursive vs. iterative execution* and *Task 4: Memory access patterns* problems. Moreover, participants are divided into two groups: Group AB completes Block A *without the tool*, followed by Block B *with the tool*. Group BA completes the reverse: Block B *without the tool*, followed by Block A *with the tool*. During the tasks, participants are shown the two different implementations of each specific problem and are asked to determine which one is more energy efficient, how confident they are in their choice and to explain their reasoning. *Without the tool*, participants must rely purely on their own reasoning by reading the code, but *with the tool*, they can use the plugin to measure the energy consumption of code. It is up to each individual participant how they choose to integrate the plugin into their reasoning process. This counterbalancing is also designed to prevent the difficulty of the specific algorithmic problem from skewing the results of the *with tool* versus *without tool* conditions. As noted in research regarding crossover designs in software engineering experiments [68], while a washout period (i.e., time between tasks to unlearn strategies) is often not feasible because reasoning strategies cannot easily be unlearnt, this alternating sequence is an appropriate method for mitigating validity threats associated with practice and fatigue.

We also mention that prior to proceeding with the user study, we conduct a pilot study to find out if instructional materials are clear enough, if the software and hardware work correctly, and if the duration of the experiment is appropriate.

4.2.2 Data Analysis

To capture a comprehensive view of the participants' experience, we collect both quantitative and qualitative data. To get quantitative data, for each task, we record the participant's accuracy in identifying the more energy-efficient implementation. Alongside this, participants self-report their confidence level in their choice on a 5-point Likert scale (ranging from "Extremely not confident" to "Extremely confident"). Additionally, the post-task questionnaire utilises 5-point Likert scales to quantify the tool's perceived usefulness, ease of interpretation, and likelihood of adoption in real-world practice. For qualitative data, during the tasks, participants are asked to think aloud, verbalising what they are looking at, what they are trying to figure out, and how they arrive at their conclusions. This feedback is captured using both screen and audio recordings. Furthermore, participants provide brief written explanations for their choices during the tasks, and answer open-ended questions in the post-task questionnaire regarding what they found confusing, when the tool was most useful, and what features they felt were missing.

Quantitative Data

Due to the relatively small target sample size of 20 to 30 participants, conducting inferential statistical tests (such as t-tests) would lack sufficient statistical power and could lead to unreliable or misleading conclusions [7], as they require an appropriate number of participants to yield reliable p-values and confidence intervals. Consequently, our quantitative results

are reported through descriptive statistics that characterise the observed patterns within this specific sample, without making claims about the broader developer population.

For *Identification (RQ2.2)* accuracy, we compute, per condition (*without tool* and *with tool*), the proportion of participants who correctly identified the more energy-efficient implementation for each task. These proportions are reported both per task and as an overall accuracy rate across all tasks in the respective condition. Because the crossover design assigns each participant to both conditions, accuracy can be compared at the individual level. We therefore also report how many participants performed better, worse, or equivalently when using the tool compared to the no-tool condition.

For *Confidence (RQ2.5)*, participants self-reported their confidence on a 5-point Likert scale immediately after selecting an implementation, with responses ranging from 1 ("Extremely not confident") to 5 ("Extremely confident"). We report the median of confidence ratings for each condition, as the median is a more robust summary statistic for ordinal data than the mean [38, 66]. Where relevant, we additionally report the frequency distribution of each scale point to convey the full shape of participants' responses. As with accuracy, individual-level differences in confidence between the *without tool* and *with tool* conditions are noted to support the within-subject comparison.

For perceived *Usability (RQ2.4)*, the post-task questionnaire contains several 5-point Likert-scale items assessing the tool's usefulness, ease of interpretation, and likelihood of adoption in real-world practice. Again, we report the median per item, as well as the frequency distribution across scale points, to characterise how participants collectively rated each dimension of usability.

Across all quantitative measures, results are presented using descriptive summaries, i.e., frequencies, proportions, and medians, accompanied by visualisations where appropriate. These statistics serve to describe the observed tendencies in the data and are used to contextualise and triangulate the qualitative findings, rather than to make generalisable or statistically significant claims.

Qualitative Data

To analyse the qualitative data gathered during the study, particularly the participants' written reasoning during tasks and their responses to the open-ended questionnaire questions, we employ Thematic Analysis (TA). TA is a qualitative research method designed for identifying, analysing, and reporting patterns (themes) within qualitative data [2, 21]. Although it is considered to be impossible to estimate a required minimum sample size for TA [73], Braun and Clarke [3] consider that "*a sample size of between 15 and 30 individual interviews tends to be common in research which aims to identify patterns across data*", making our aim of 20 to 30 participants convenient.

To improve the credibility of our data interpretation, given that Inter-Coder Reliability (ICR) is "*a somewhat controversial topic in the qualitative research community, with some arguing that it is an inappropriate or unnecessary step within the goals of qualitative analysis*" [43, p. 1], we instead incorporate a triangulation strategy [45, 5]. Triangulation is "*using two or more data sources, methods, or researchers to try to gain a fuller or multi-faceted understanding of a topic*" [3, p. 223]. While the primary textual analysis is conducted

on the written answers provided by participants, we utilise the screen and audio recordings as a supplementary data source. Specifically, the recordings are consulted to cross-reference and clarify any instances where the written answers are unclear, confusing, or contradictory. We also triangulate with the quantitative data [3], as mentioned before. Moreover, we include an additional analytic auditor "*for a 'verification step' of reviewing the data for discrepancies, overstatements or errors*" [3], as recommended by Braun and Clarke [3].

The analytical process follows the six-phase TA framework described by Braun and Clarke [2], which guides us from initial data familiarisation to the final reporting of themes, as seen in Table 4.1. For Phase 1 (Familiarising yourself with the data), the primary method is active, repeated reading of the data to search for meanings and patterns. For our specific study, this means reading and re-reading the participants' written open-ended questionnaire responses and written reasonings. During Phase 2 of the TA approach (Generating initial codes), we apply the initial coding technique [64, 63]. According to Saldaña, initial coding involves "*breaking down qualitative data into discrete parts, closely examining them, and comparing them for similarities and differences*" [53, p.100]. We select this specific coding method because it provides an open-ended starting point for data review and, as Saldaña notes, it "*is appropriate for virtually all qualitative studies, but particularly for beginning qualitative researchers learning how to code data*" [53, p.101]. Additionally, we note that initial coding is sometimes referred to as open coding, and while the term open coding is associated with grounded theory [34, 64], we employ initial coding here strictly as a data labelling technique within the TA framework, not as part of a grounded theory methodology. The codes generated are therefore not intended to be theory-generative but to serve as the building blocks for theme development in subsequent phases. Moreover, the developed codes should have a consistently similar level of abstraction.

Following Phase 2, the initial codes are collated into broader candidate themes (Phase 3), gathering all relevant data extracts under each potential theme. These candidate themes are then reviewed against both the coded extracts and the full dataset to check if they accurately and coherently represent the participants' meanings (Phase 4), producing a thematic map of the analysis. Each retained theme is subsequently defined and named to capture its analytical essence and its relationship to the research questions (Phase 5). We note that our application of TA follows an inductive orientation [2], in that themes are derived from the data itself and are strongly linked to the data [1], rather than from a pre-existing theoretical framework or codebook [9]. Deductive coding approaches are sometimes misunderstood as coding driven by the research questions or by the data collection questions. In our case, although the five research sub-questions provide a thematic focus that delimits the domain of analysis, they do not predetermine the content of the themes, as patterns are identified from the participants' own words and reasonings. At last, the analysis is synthesised into the final analytical report presented in Section 6.3 (Phase 6), where representative extracts from participant responses are selected to illustrate each theme and the findings are related back to the five research sub-questions.

Phase	Description of the process
1. Familiarizing yourself with your data:	Transcribing data (if necessary), reading and re-reading the data, noting down initial ideas.
2. Generating initial codes:	Coding interesting features of the data in a systematic fashion across the entire data set, collating data relevant to each code.
3. Searching for themes:	Collating codes into potential themes, gathering all data relevant to each potential theme.
4. Reviewing themes:	Checking if the themes work in relation to the coded extracts (Level 1) and the entire data set (Level 2), generating a thematic 'map' of the analysis.
5. Defining and naming themes:	Ongoing analysis to refine the specifics of each theme, and the overall story the analysis tells, generating clear definitions and names for each theme.
6. Producing the report:	The final opportunity for analysis. Selection of vivid, compelling extract examples, final analysis of selected extracts, relating back of the analysis to the research question and literature, producing a scholarly report of the analysis.

Table 4.1: Phases of thematic analysis, as presented in Braun and Clarke [2].

Triangulation

We aim to address *RQ2* and its respective sub-questions by employing the aforementioned methodology on the quantitative and qualitative data. To summarise, *Interpretation (RQ2.1)* is evaluated through the data gathered from the participants' think-aloud verbalisations during the tasks, combined with their written reasoning explanations. *Identification (RQ2.2)* is addressed by analysing the actual choices participants make between the two implementations alongside their written reasoning for those choices. *Integration (RQ2.3)* is also evaluated through the written explanations participants provide regarding their decision-making process during the code analysis tasks. *Usability (RQ2.4)* is answered using the post-task questionnaire, which asks participants to agree or disagree with statements about the tool's ease of use and allows them to highlight confusing aspects or suggest improvements. *Confidence (RQ2.5)*, which was introduced based on pilot study feedback, is evaluated by asking participants to rate their confidence level on a 5-point scale (from "Extremely confident" to "Extremely not confident") immediately after selecting the most efficient implementation. Finally, *RQ2* is answered by triangulation, synthesising the data collected from the five targeted sub-questions.

Chapter 5

Verification

As outlined in the study setup (see Section 4.1), the verification phase aims to determine whether the proposed plugin accurately measures energy consumption by detecting statistically significant differences between pairs of different implementations. This section presents the empirical results of our data collection and the subsequent statistical analysis.

5.1 Results

To check the accuracy of the proposed energy measurement plugin, we created a dataset of 30 distinct algorithmic problems sourced from LeetCode, available in our replication package [35]. As specified in the study setup, each problem was associated with two implementations: one characterised as *fast* (or *faster* or *better*) and another as *slow* (or *slower* or *worse*) based on the runtime performance reported by the LeetCode Submissions Runtime analysis. The selected problems span a wide range, including dynamic programming, string manipulation, and graph theory. For each solution, we took its respective problem name given by LeetCode, removed the whitespaces, and appended the LeetCode-reported runtime to create the names for our dataset (e.g., *Candy2ms*). The complete list of algorithm pairs, along with their execution times reported by LeetCode at the time of collection, is detailed in Table 5.1. There were some cases where the *best* solution would show to have "*used 0ms of runtime*" and we identified two possible explanations for this. If the solution genuinely executed faster than the minimum measurement of the platform, we kept it in the dataset, but if the original submitter used tricks to manipulate the runtime results (e.g., overriding the `display_runtime.txt` file that LeetCode uses to store and display the runtime on the website), we discarded the cheated one and proceeded with another top-performing solution.

We manually ran each of the 60 solutions 30 times and recorded both time and energy measurements as explained in Section 4.1.1. The measurements were conducted manually because the tool can only be triggered through an interactive UI and because we needed per-run supervision (failed run detection, environment reset, cooldown and warm-up handling). The experiment is also small enough that manual overhead is manageable.

For uniform precision across the dataset prior to statistical evaluation, all raw execution times were mathematically standardised to four decimal places, while energy measurements

Algorithm	**Fast Implementation**	**Slow Implementation**
BestTimeToBuyAndSellStockIV	0ms	15ms
BestTimetoBuyandSellStockIII	1ms	167ms
Candy	2ms	19ms
CountOfRangeSum	47ms	748ms
ExpressionAddOperators	3ms	166ms
FirstMissingPositive	1ms	28ms
IntegerToEnglishWords	1ms	9ms
LongestBalancedSubarrayI	9ms	445ms
LongestBalancedSubarrayII	145ms	456ms
LongestIncreasingPathInAMatrix	6ms	90ms
LongestPalindromicSubstring	45ms	2386ms
LongestValidParentheses	1ms	7ms
MaxPointsOnALine	6ms	47ms
MedianOfTwoSortedArray	1ms	12ms
MinimumWindowSubstring	5ms	269ms
PalindromePairs	104ms	1431ms
PerfectRectangle	5ms	79ms
PermutationSequence	8ms	1279ms
RegularExpressionMatching	6ms	1036ms
RemoveInvalidParentheses	1ms	423ms
RussianDollEnvelopes	10ms	74ms
ScrambleString	2ms	79ms
SelfCrossing	1ms	10ms
ShortestPalindrome	4ms	343ms
SlidingWindowMaximum	6ms	318ms
SubstringWithConcatenation	8ms	358ms
SudokuSolver	7ms	402ms
ValidNumber	1ms	12ms
WordSearchII	35ms	1905ms
ZigzagConversion	2ms	39ms

Table 5.1: Selected LeetCode problems and their reported runtime for the two implementations. Times are as reported by the LeetCode platform at the time of data collection.

were reported to three decimal places by default through the plugin.

Following the established methodology, we first checked the data for normality. In accordance with energy measurement research, for each batch of 30 measurements, we removed "*all data points that deviate from the mean more than 3 standard deviations – i.e.,* $|\bar{x}-x| > 3s$*, where* $\bar{x}$ *is the sample mean,* x *is the value of the measurement and* s *is standard deviation of the sample*" [10], resulting in 11 removed outliers. The Shapiro-Wilk test revealed that the measurements were largely non-normally distributed, particularly for energy, where 45 out of 60 solutions (75%) failed the normality assumption. Execution time showed non-

normality in 12 out of 60 cases (20%). These results are confirmed by the visualisations of these distributions. Histograms with Gaussian curve overlay and Kernel Density Estimation (KDE) plots for each of the 60 implementations are provided in Appendix B, and Appendix C respectively. The detailed numerical results of the Shapiro-Wilk test for each of the 60 implementations are presented in Table 5.2. There are only two projects (*Candy* and *SubstringWithConcatenation*) where both energy and time are normally distributed across both implementations. Given these results, we proceeded with non-parametric methods for the remainder of the analysis.

To test the hypothesis on time and energy differences, we used the one-tailed Mann-Whitney U test, addressing *Baseline (RQ1.1)* and *Sensitivity (RQ1.2)*. The results (see Table 5.3) showed that the local execution time measurements aligned with *fast* versus *slow* reported runtimes by LeetCode in 28 out of 30 cases (93.3%). Furthermore, the *fast* implementations consumed significantly less energy in 25 out of 30 cases (83.3%). Considered together, both metrics followed the expected direction in 24 out of 30 pairs. We identify four possible combinations of time and energy significance to classify the 30 problems, including the six exception problems, and display them in Table 5.5.

Although p-values show the sensitivity of the plugin, they do not quantify the magnitude of the energy savings. To investigate this, we calculated Cliff's delta for each of the 30 problem pairs, addressing *Magnitude (RQ1.3)*. The analysis revealed a distinction between time and energy savings, as seen in Table 5.4. For 12 problem pairs (e.g., *ExpressionAddOperators*, *LongestBalancedSubarrayI*, *MedianOfTwoSortedArray*, *ScrambleString*), both time and energy effect sizes were completely maximised at +1.000 (*large*). In these cases, the *better-performing* algorithm is indeed preferable, as it executes faster and translates that time saving into a notable energy reduction. However, the effect size table also showed some nuances. Across all pairs, energy effects were distributed as 19/30 *large*, 3/30 *medium*, 3/30 *small*, and 5/30 *negligible*, whereas time effects were 25/30 *large*, 3/30 *medium*, 1/30 *small*, and 1/30 *negligible*. Algorithms such as *CountOfRangeSum* and *IntegerToEnglishWords* remained statistically significant for energy ($p = 0.0217$ and $p = 0.0295$ respectively), but with only *small* practical effects ($\delta = +0.304$ and $\delta = +0.287$ respectively). Moreover, in the case of *SlidingWindowMaximum*, the time difference was *large* ($\delta = +0.849$), signifying a faster algorithm, but the energy effect size was *negligible* ($\delta = +0.122$). This finding shows that a large reduction in execution time does not guarantee a proportional, or even meaningful, reduction in energy consumption. To address *Correlation (RQ1.4)*, this interpretation is further supported by cross-solution association: median execution time and median energy were strongly positively correlated (Spearman $\rho = 0.7992$, see Figure 5.1), with fitted relation Energy = $1.1716 \times$ Time $- 0.0730$.

5.2 Analysis of Exceptions

The six exceptions that we will elaborate on further are *ValidNumber*, *Candy*, *ShortestPalindrome*, *SlidingWindowMaximum*, *SelfCrossing*, and *MaxPointsOnALine*.

ValidNumber. The *ValidNumber* problem states that when given a string s, the solution should return whether s is a valid number. The constraints are that the length of the input

Name	Energy p	Normal?	Time p	Normal?	Both Normal?
BestTimeToBuyAndSellStockIV0ms	0.0679	Yes	0.3600	Yes	Yes
BestTimeToBuyAndSellStockIV15ms	0.0119	No	0.0487	No	No
BestTimetoBuyandSellStockIII167ms	0.0005	No	0.7909	Yes	No
BestTimetoBuyandSellStockIII1ms	0.0035	No	0.7667	Yes	No
Candy19ms	0.1387	Yes	0.8544	Yes	Yes
Candy2ms	0.2773	Yes	0.7578	Yes	Yes
CountOfRangeSum47ms	0.5262	Yes	0.3698	Yes	Yes
CountOfRangeSum748ms	0.0087	No	0.9209	Yes	No
ExpressionAddOperators166ms	0.0001	No	0.6178	Yes	No
ExpressionAddOperators3ms	0.0345	No	0.7140	Yes	No
FirstMissingPositive1ms	0.0244	No	0.0025	No	No
FirstMissingPositive28ms	0.2853	Yes	0.1373	Yes	Yes
IntegerToEnglishWords1ms	0.0030	No	0.0050	No	No
IntegerToEnglishWords9ms	0.0019	No	0.0003	No	No
LongestBalancedSubarrayI445ms	0.0001	No	0.2539	Yes	No
LongestBalancedSubarrayI9ms	0.0000	No	0.8325	Yes	No
LongestBalancedSubarrayII145ms	0.0001	No	0.2408	Yes	No
LongestBalancedSubarrayII456ms	0.0000	No	0.0545	Yes	No
LongestIncreasingPathInAMatrix6ms	0.0002	No	0.4130	Yes	No
LongestIncreasingPathInAMatrix90ms	0.0000	No	0.0803	Yes	No
LongestPalindromicSubstring2386ms	0.5082	Yes	0.0004	No	No
LongestPalindromicSubstring45ms	0.0004	No	0.6784	Yes	No
LongestValidParentheses1ms	0.7127	Yes	0.1166	Yes	Yes
LongestValidParentheses7ms	0.0402	No	0.0796	Yes	No
MaxPointsOnALine47ms	0.0005	No	0.3413	Yes	No
MaxPointsOnALine6ms	0.0944	Yes	0.0278	No	No
MedianOfTwoSortedArray12ms	0.0406	No	0.0400	No	No
MedianOfTwoSortedArray1ms	0.0284	No	0.2117	Yes	No
MinimumWindowSubstring269ms	0.0156	No	0.7332	Yes	No
MinimumWindowSubstring5ms	0.0026	No	0.3426	Yes	No
PalindromePairs104ms	0.0018	No	0.8922	Yes	No
PalindromePairs1431ms	0.0000	No	0.4253	Yes	No
PerfectRectangle5ms	0.0063	No	0.0157	No	No
PerfectRectangle79ms	0.0006	No	0.1072	Yes	No
PermutationSequence1279ms	0.7336	Yes	0.0001	No	No
PermutationSequence8ms	0.0005	No	0.1950	Yes	No
RegularExpressionMatching1036ms	0.8084	Yes	0.0074	No	No
RegularExpressionMatching6ms	0.2346	Yes	0.3955	Yes	Yes
RemoveInvalidParentheses1ms	0.0008	No	0.8155	Yes	No
RemoveInvalidParentheses423ms	0.0024	No	0.3173	Yes	No
RussianDollEnvelopes10ms	0.0095	No	0.9052	Yes	No
RussianDollEnvelopes74ms	0.0008	No	0.2816	Yes	No
ScrambleString2ms	0.0480	No	0.6463	Yes	No
ScrambleString79ms	0.0109	No	0.4562	Yes	No
SelfCrossing10ms	0.0115	No	0.3672	Yes	No
SelfCrossing1ms	0.0234	No	0.9054	Yes	No
ShortestPalindrome343ms	0.0178	No	0.9092	Yes	No
ShortestPalindrome4ms	0.3228	Yes	0.5222	Yes	Yes
SlidingWindowMaximum318ms	0.0091	No	0.6686	Yes	No
SlidingWindowMaximum6ms	0.0010	No	0.7584	Yes	No
SubstringWithConcatenation358ms	0.2026	Yes	0.1948	Yes	Yes
SubstringWithConcatenation8ms	0.1957	Yes	0.5771	Yes	Yes
SudokuSolver402ms	0.0233	No	0.5050	Yes	No
SudokuSolver7ms	0.0089	No	0.7997	Yes	No
ValidNumber12ms	0.0007	No	0.8233	Yes	No
ValidNumber1ms	0.0017	No	0.4533	Yes	No
WordSearchII1905ms	0.8413	Yes	0.0002	No	No
WordSearchII35ms	0.0000	No	0.0000	No	No
ZigzagConversion2ms	0.0018	No	0.6696	Yes	No
ZigzagConversion39ms	0.0012	No	0.0910	Yes	No

Table 5.2: Shapiro-Wilk normality tests ($p < 0.05$ indicates not normally distributed).

Algorithm	Energy p	Time p	Energy Sig.?	Time Sig.?
BestTimeToBuyAndSellStockIV	0.0380	0.0000	Yes	Yes
BestTimetoBuyandSellStockIII	0.0076	0.0000	Yes	Yes
Candy	0.2209	0.0103	No	Yes
CountOfRangeSum	0.0217	0.0021	Yes	Yes
ExpressionAddOperators	0.0000	0.0000	Yes	Yes
FirstMissingPositive	0.0000	0.0000	Yes	Yes
IntegerToEnglishWords	0.0295	0.0000	Yes	Yes
LongestBalancedSubarrayI	0.0000	0.0000	Yes	Yes
LongestBalancedSubarrayII	0.0000	0.0000	Yes	Yes
LongestIncreasingPathInAMatrix	0.0001	0.0000	Yes	Yes
LongestPalindromicSubstring	0.0000	0.0000	Yes	Yes
LongestValidParentheses	0.0000	0.0000	Yes	Yes
MaxPointsOnALine	0.0000	1.0000	Yes	No
MedianOfTwoSortedArray	0.0000	0.0000	Yes	Yes
MinimumWindowSubstring	0.0000	0.0000	Yes	Yes
PalindromePairs	0.0001	0.0000	Yes	Yes
PerfectRectangle	0.0038	0.0000	Yes	Yes
PermutationSequence	0.0000	0.0000	Yes	Yes
RegularExpressionMatching	0.0000	0.0000	Yes	Yes
RemoveInvalidParentheses	0.0000	0.0000	Yes	Yes
RussianDollEnvelopes	0.0032	0.0000	Yes	Yes
ScrambleString	0.0000	0.0000	Yes	Yes
SelfCrossing	0.1874	0.3057	No	No
ShortestPalindrome	0.3950	0.0130	No	Yes
SlidingWindowMaximum	0.2101	0.0000	No	Yes
SubstringWithConcatenation	0.0000	0.0000	Yes	Yes
SudokuSolver	0.0000	0.0179	Yes	Yes
ValidNumber	0.7079	0.0000	No	Yes
WordSearchII	0.0000	0.0000	Yes	Yes
ZigzagConversion	0.0000	0.0000	Yes	Yes

Table 5.3: Statistical comparison of energy and time across all algorithm pairs using Mann-Whitney U test ($p < 0.05$ indicates significant).

Algorithm	Energy δ	Energy Size	Time δ	Time Size
BestTimeToBuyAndSellStockIV	+0.268	small	+0.991	large
BestTimetoBuyandSellStockIII	+0.366	medium	+0.679	large
Candy	+0.117	negligible	+0.349	medium
CountOfRangeSum	+0.304	small	+0.432	medium
ExpressionAddOperators	+1.000	large	+1.000	large
FirstMissingPositive	+0.913	large	+1.000	large
IntegerToEnglishWords	+0.287	small	+0.700	large
LongestBalancedSubarrayI	+1.000	large	+1.000	large
LongestBalancedSubarrayII	+0.778	large	+0.986	large
LongestIncreasingPathInAMatrix	+0.564	large	+0.998	large
LongestPalindromicSubstring	+1.000	large	+1.000	large
LongestValidParentheses	+0.723	large	+1.000	large
MaxPointsOnALine	+0.976	large	-0.936	large
MedianOfTwoSortedArray	+1.000	large	+1.000	large
MinimumWindowSubstring	+1.000	large	+1.000	large
PalindromePairs	+0.544	large	+0.996	large
PerfectRectangle	+0.402	medium	+0.800	large
PermutationSequence	+1.000	large	+1.000	large
RegularExpressionMatching	+1.000	large	+1.000	large
RemoveInvalidParentheses	+1.000	large	+1.000	large
RussianDollEnvelopes	+0.415	medium	+0.606	large
ScrambleString	+1.000	large	+1.000	large
SelfCrossing	+0.136	negligible	+0.078	negligible
ShortestPalindrome	+0.041	negligible	+0.336	medium
SlidingWindowMaximum	+0.122	negligible	+0.849	large
SubstringWithConcatenation	+1.000	large	+1.000	large
SudokuSolver	+0.716	large	+0.317	small
ValidNumber	-0.081	negligible	+0.987	large
WordSearchII	+1.000	large	+1.000	large
ZigzagConversion	+1.000	large	+1.000	large

Table 5.4: Effect size analysis using Cliff's delta (δ).

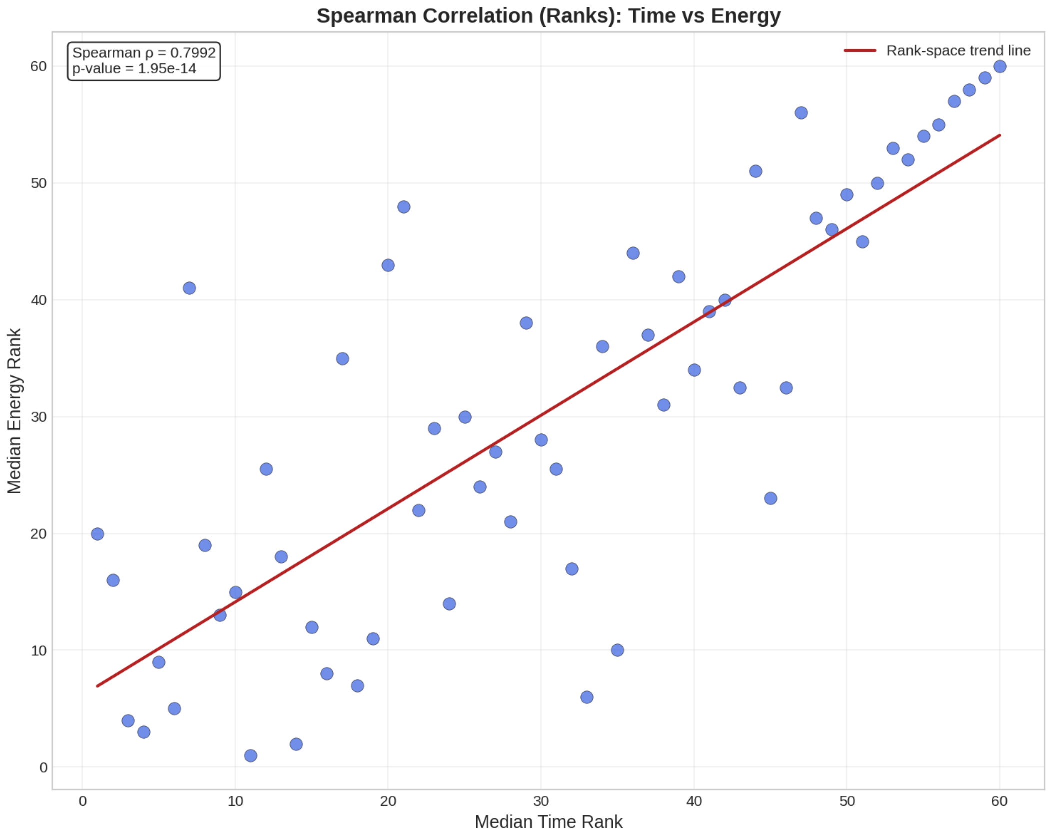


Figure 5.1: Spearman rank correlation between median execution time and median energy consumption across algorithm implementations. Each point represents one implementation (*fast* or *slow* variants) in rank space. The fitted rank-space trend line and reported ρ (with p-value) indicate a strong positive monotonic association.

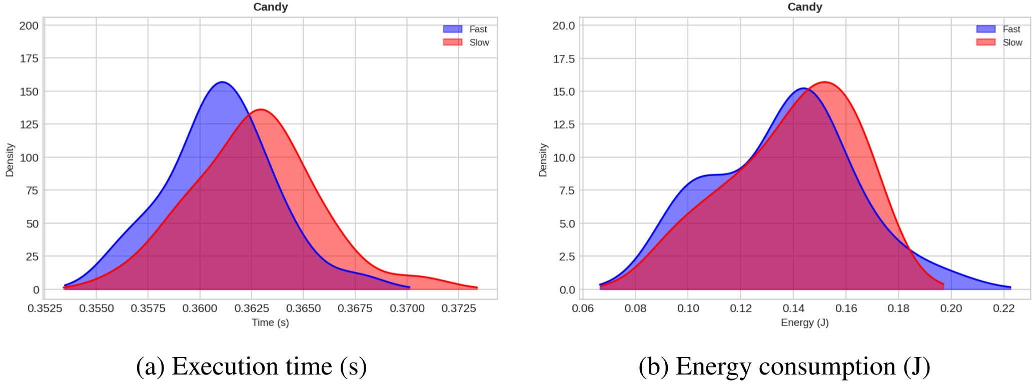
string must be at most 20 characters, and that only specific characters are allowed. The *fast* solution uses a one-pass linear scan over the input string in $O(n)$ time, while the *slow* solution uses a one-line regular expressions approach. Here, time turned out to be statistically different ($p = 0.0000$), but energy did not ($p = 0.7079$). However, this problem should be discarded from the results discussion, since we mistakenly violated the problem input constraints by choosing an input string longer than permitted and by including disallowed characters.

Candy. The *Candy* problem states that given an array of size n of candy ratings by children, one needs to distribute the minimum total number of candies to a line of children such that every child receives at least one, and any child with a higher rating than an immediate neighbour receives more candies than that neighbour. The *fast* solution uses a greedy approach of $O(n)$ time and $O(n)$ space, while the *slow* solution attempts to solve the problem by processing children in order of their ratings, resulting in $O(n \log n)$ time and $O(n)$ space. As input, we provided a long sequence of decreasing values followed by a long sequence

Time Sig.?	Energy Sig.?	Count	Algorithms
✓	×	4	Candy, ShortestPalindrome, SlidingWindowMaximum, ValidNumber
×	✓	1	MaxPointsOnALine
×	×	1	SelfCrossing
✓	✓	24	BestTimeToBuyAndSellStockIV, BestTimetoBuyandSellStockIII, CountOfRangeSum, ExpressionAddOperators, FirstMissingPositive, IntegerToEnglishWords, LongestBalancedSubarrayI, LongestBalancedSubarrayII, LongestIncreasingPathInAMatrix, LongestPalindromicSubstring, LongestValidParentheses, MedianOfTwoSortedArray, MinimumWindowSubstring, PalindromePairs, PerfectRectangle, PermutationSequence, RegularExpressionMatching, RemoveInvalidParentheses, RussianDollEnvelopes, ScrambleString, SubstringWithConcatenation, SudokuSolver, WordSearchII, ZigzagConversion

Table 5.5: Classification of the 30 algorithmic problems into four possible combinations based on the statistical significance ($p < 0.05$) of execution time and energy consumption.

of increasing values. Time turned out to be statistically different ($p = 0.0103$), but energy did not ($p = 0.2209$), and these results become visually understandable upon looking at the KDE distribution plots in Figure 5.2. Since *Candy* has all four distributions of time and energy for the *fast* and *slow* implementations normally distributed, we made an exception at the end of the analysis to apply the parametric pipeline to this problem only, to check if it would yield different results than the non-parametric methods. However, even on the parametric pipeline, *Candy* still had significant time differences, but no significant energy differences.



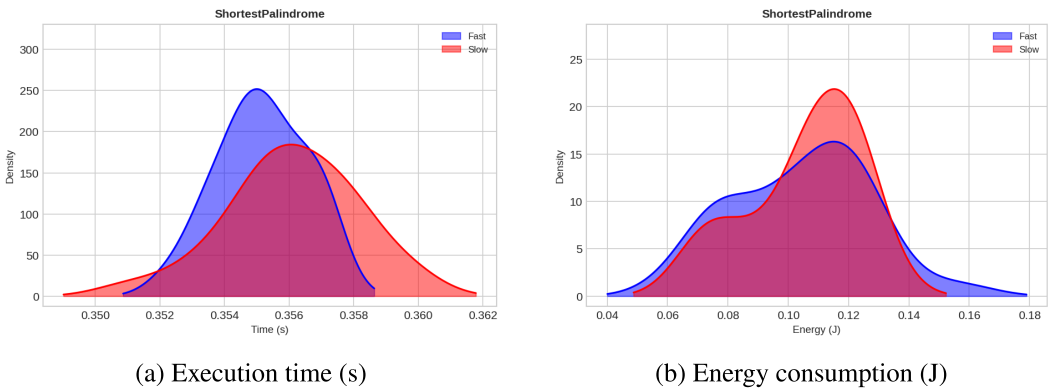
(a) Execution time (s)

(b) Energy consumption (J)

Figure 5.2: Density plots for the *Candy* problem comparing execution time and energy consumption between the *fast* and *slow* solutions.

ShortestPalindrome. The goal of the *ShortestPalindrome* problem is to find the longest

palindromic prefix already present in a given string of size n. The *fast* implementation uses a recursive reduction approach where it compares the string to its own reverse to isolate the longest palindromic prefix in $O(n^2)$ time, while the *slow* one brute forces by creating and comparing new substring objects for every possible overlap taking $O(n^2)$ time as well. The constraints are that the given string s should satisfy the relation $0 <= s.length <= 5 * 10^4$ and consist of only English lowercase letters. We used $s =$ ''racecar'' as input, which yielded significant time differences ($p = 0.0130$), but no significant energy differences ($p = 0.3950$), with the KDE plot visualised in Figure 5.3.



(a) Execution time (s)

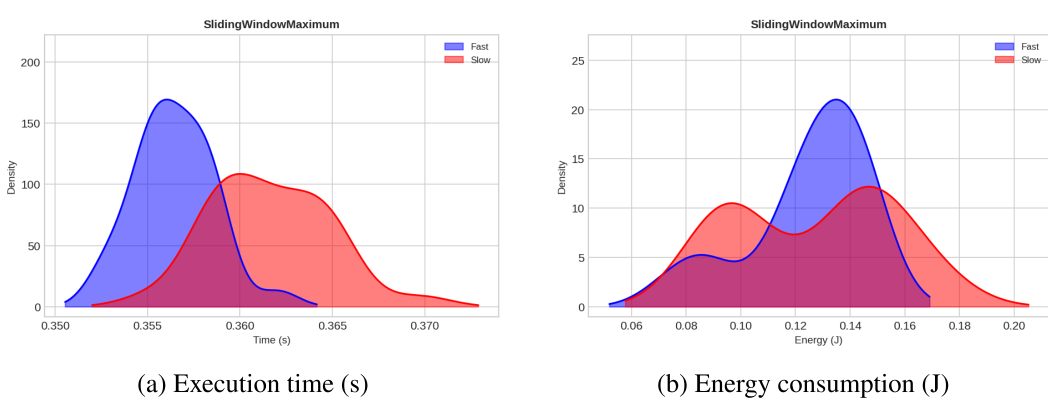
(b) Energy consumption (J)

Figure 5.3: Density plots for the *ShortestPalindrome* problem comparing execution time and energy consumption between the *fast* and *slow* solutions.

SlidingWindowMaximum. The *SlidingWindowMaximum* problem requires finding the maximum value within a sliding window of size k as it moves across an array of size n from left to right. The *fast* solution uses a block-partitioning strategy with $O(n)$ space and time, while the *slow* one uses a frequency map implemented with `TreeMap` to keep track of the numbers currently inside the window in $O(n \log k)$ time and with $O(n)$ space. As input, we used repeating sequences of $0 - 99$ with $k = 500$, and this resulted in significant time differences ($p = 0.0000$), but no energy differences ($p = 0.2101$), with the KDE distribution seen in Figure 5.4.

SelfCrossing. For the *SelfCrossing* problem, given an array of n movement lengths, the goal is to determine if a path that consistently turns counter-clockwise intersects its own previous segments. The *fast* implementation uses state machine logic taking $O(n)$ time with $O(1)$ space, while the *slow* one uses pattern matching on three crossing cases, taking $O(n)$ time with $O(1)$ space as well. As input, we used a 1 to 500 array in perfectly increasing order, such that the path never crosses itself. This setup yielded neither significant time differences ($p = 0.3057$), nor significant energy differences ($p = 0.1874$). The results are consistent with the respective time and energy KDE distribution plots available in Figure 5.5.

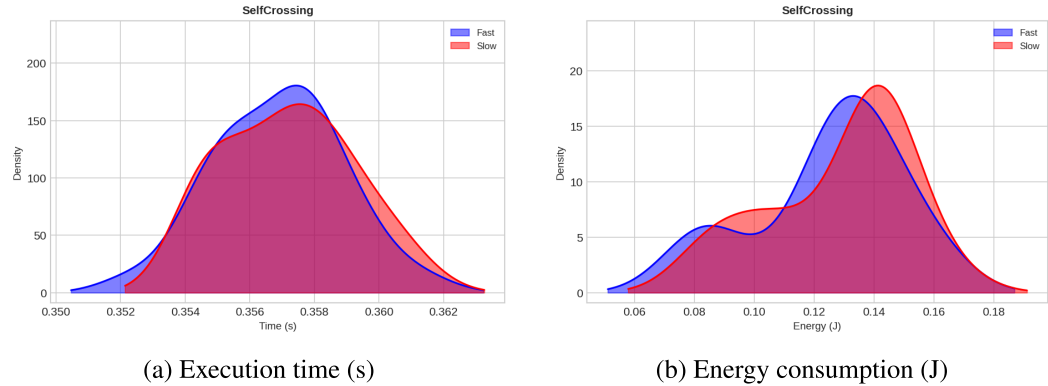
MaxPointsOnALine. For the *MaxPointsOnALine* problem, given a set of size n of coordinates on a 2D plane, one needs to determine the maximum number of points that reside on the same infinitely extending straight line. The *fast* solution uses a nested loop to calculate floating-point slopes between pairs of points, storing and counting them in primitive arrays to find the most frequent trajectory, using $O(n^2)$ time and $O(n)$ space. The *slow* implemen-



(a) Execution time (s)

(b) Energy consumption (J)

Figure 5.4: Density plots for the *SlidingWindowMaximum* problem comparing execution time and energy consumption between the *fast* and *slow* solutions.

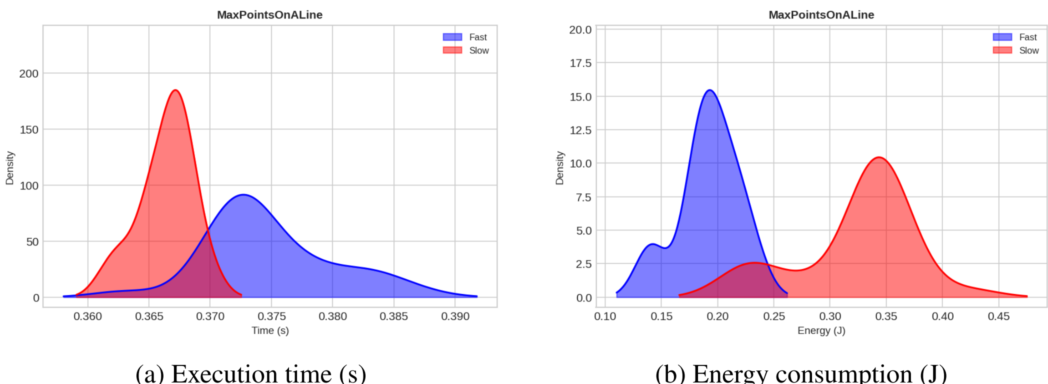


(a) Execution time (s)

(b) Energy consumption (J)

Figure 5.5: Density plots for the *SelfCrossing* problem comparing execution time and energy consumption between the *fast* and *slow* solutions.

tation reduces every slope to a fraction via the Greatest Common Divisor (GCD) and stores these as unique string keys in a `HashMap` that manages sets of associated points, using $O(n^2)$ time and space. The *MaxPointsOnALine* problem is an interesting case because it is the only one where we have no significant time differences ($p = 1.0000$), but we do have significant energy differences ($p = 0.0000$). The respective time and energy KDE distribution plots that show this exception are available in Figure 5.6.



(a) Execution time (s) (b) Energy consumption (J)

Figure 5.6: Density plots for the *MaxPointsOnALine* problem comparing execution time and energy consumption between the *fast* and *slow* solutions.

5.3 Summary

RQ1 Summary: The verification phase tested the tool across 30 LeetCode algorithm pairs. Energy measurements were largely non-normally distributed (75% of solutions), so non-parametric methods were applied throughout. For *Baseline (RQ1.1)*, local execution time aligned with LeetCode-reported runtimes in 28 of 30 cases (93.3%), showing the plugin captures meaningful performance differences. Regarding *Sensitivity (RQ1.2)*, the faster implementation consumed significantly less energy in 25 of 30 cases (83.3%), with both metrics following the expected direction jointly in 24 of 30 pairs. Six exceptions were identified: *ValidNumber* (time significant, energy not, but invalid input), *Candy* and *ShortestPalindrome* and *SlidingWindowMaximum* (time significant, energy not), *MaxPointsOnALine* (energy significant, time not), and *SelfCrossing* (neither significant). For *Magnitude (RQ1.3)*, effect sizes via Cliff's delta showed that energy effects were often smaller than time effects, suggesting a faster algorithm does not guarantee a proportional energy reduction. Lastly, addressing *Correlation (RQ1.4)*, median execution time and median energy were strongly positively correlated (Spearman $\rho = 0.7992$), with a near-linear fitted relation, but the scatter showed that time alone is an imperfect proxy for energy consumption.

Chapter 6

Validation

As stated in the study setup (see Section 4.2), the validation phase aims to show whether the proposed tool achieves its goal of increasing energy awareness and fitting into the developer workflow. This section presents the results and subsequent analysis of our user study.

6.1 Participant Overview

We had 22 participants who were evenly split between the two crossover groups, with 11 participants in Group AB and 11 in Group BA. The majority of participants had substantial general programming experience, with 15 participants (68.2%) reporting five or more years of experience and a further six (27.3%) reporting three to five years. Java-specific experience was more limited: 11 participants (50.0%) had one to three years of Java experience, five (22.7%) had less than one year, four (18.2%) had three to five years, and only two (9.1%) had five or more years. When asked about the context of their experience, 12 participants (54.5%) stated they use Java in their education, six (27.3%) use it both in and outside of education, four (18.2%) selected no experience, and three (13.6%) consider Java one of their main programming languages of choice. For IntelliJ familiarity, eight participants (36.4%) reported using IntelliJ occasionally, seven (31.8%) considered it their primary IDE for Java development, and the remaining seven (31.8%) had never used it before.

With respect to prior exposure to energy and performance topics, 17 participants (77.3%) had taken courses or conducted research related to energy and/or performance optimisation. However, practical experience with energy optimisation specifically was rare: 16 participants (72.7%) reported having never performed energy optimisation, and only one participant (4.5%) did so on a daily basis. Performance optimisation was considerably more common, with nine participants (40.9%) performing it weekly.

In terms of occupation, the sample was predominantly composed of Computer Science, Data Science, or Artificial Intelligence students (18 participants, 81.8%), with five software engineers or developers (22.7%) among them.

This background confirms that the sample, although technically literate, largely lacked prior experience with energy-aware development, making it a realistic representation of the tool's intended user base. The full results are available in Figure 6.1.

Background questionnaire results

Programming experience
4.5%
27.3%
68.2%
5+ years (15)
1-3 years (1)
3-5 years (6)

Java experience
9.1%
18.2%
50.0%
22.7%
1-3 years (11)
3-5 years (4)
less than 1 year (5)
5+ years (2)

IntelliJ experience
31.8%
36.4%
31.8%
use IntelliJ sometimes (8)
never used IntelliJ (7)
IntelliJ is my IDE of choice for Java development (7)

Energy and/or performance courses
22.7%
77.3%
Yes (17)
No (5)

Energy optimisation frequency
4.5%
9.1%
13.6%
72.7%
Never (16)
Monthly (2)
Once or twice a year (3)
Daily (1)

Performance optimisation frequency
4.5%
9.1%
13.6%
40.9%
31.8%
Never (1)
Weekly (9)
Once or twice a year (3)
Daily (2)
Monthly (7)

Occupation (multi-select)
54.5%
6.1%
15.2%
CS/DS/AI Student (18, 81.8%)
Software Engineer/Developer (5, 22.7%)
Data/AI Engineer/Developer (2, 9.1%)
DevOps (1, 4.5%)
Frontend Engineer/Developer (1, 4.5%)
Full-stack Engineer/Developer (1, 4.5%)
Game Technology Student (1, 4.5%)
Geomatics (1, 4.5%)
IT Infrastructure Manager (1, 4.5%)
Nanobiology Student (1, 4.5%)
Post-doc researcher (Earth System Modelling) (1, 4.5%)

Figure 6.1: Overview of participant demographics and technical background.

6.2 Quantitative Data Results

This section reports the quantitative findings of the user study using descriptive statistics, as motivated in Section 4.2.2. No inferential claims are made, as the results only characterise the observed patterns within this specific sample of 22 participants.

To address *Identification (RQ2.2)*, we examine the proportion of correct answers across tasks and conditions. Table 6.1 summarises the per-task accuracy collapsed across both conditions, showing the number and percentage of participants who correctly identified the more energy-efficient implementation for each of the four tasks. Tasks 2 and 3 yielded the highest overall accuracy (95.5% and 86.4% respectively), while Tasks 1 and 4 seemed more challenging, with only 68.2% and 59.1% of participants answering correctly when both conditions are pooled together.

Task	Description	Correct (n, out of 22)	Correct (%)
Task 1	*String manipulation*	15	68.2
Task 2	*Collection lookup*	21	95.5
Task 3	*Recursive vs. iterative*	19	86.4
Task 4	*Memory access patterns*	13	59.1

Table 6.1: Per-task accuracy across all 22 participants (both conditions combined).

The primary comparison of interest is between the *without tool* and *with tool* conditions. Table 6.2 and Figure 6.2 present accuracy broken down by both task and condition. Without the tool, participants correctly identified the more energy-efficient implementation in 25 out of 44 cases (56.8%), which is only marginally above chance for a binary choice task. With the tool, accuracy is 43 out of 44 cases (97.7%), with the single incorrect response occurring on Task 4. Using the tool produced perfect accuracy on Tasks 1, 2, and 3. Notably, Task 1 (*String manipulation*) saw the largest improvement, rising from 36.4% without the tool to 100% with it, suggesting that this particular scenario is difficult to reason about from source code alone but trivially resolvable with measured energy data. Task 4 (*Memory access patterns*) remained the most challenging based on accuracy even with the tool, with one participant choosing to trust their own intuition and still selecting the less efficient implementation despite having measurement data available.

At the individual level, the within-subject comparison reveals that 16 out of 22 participants (72.7%) achieved higher accuracy when using the tool than when not using it, one participant performed worse with the tool, and five participants showed equivalent accuracy in both conditions, because they answered correctly in both.

To address *Confidence (RQ2.5)*, participants rated their confidence in their implementation choice immediately after each selection using a 5-point Likert scale, ranging from "Extremely not confident" to "Extremely confident". The median confidence rating across all four tasks was "Somewhat confident" / "Extremely confident" for Task 1, "Extremely confident" for Task 2, and "Somewhat confident" for Tasks 3 and 4. When separated by condition, the contrast is more pronounced. Table 6.3 presents the full frequency distribution of confidence ratings per condition, and the median shifts from "Somewhat confident"

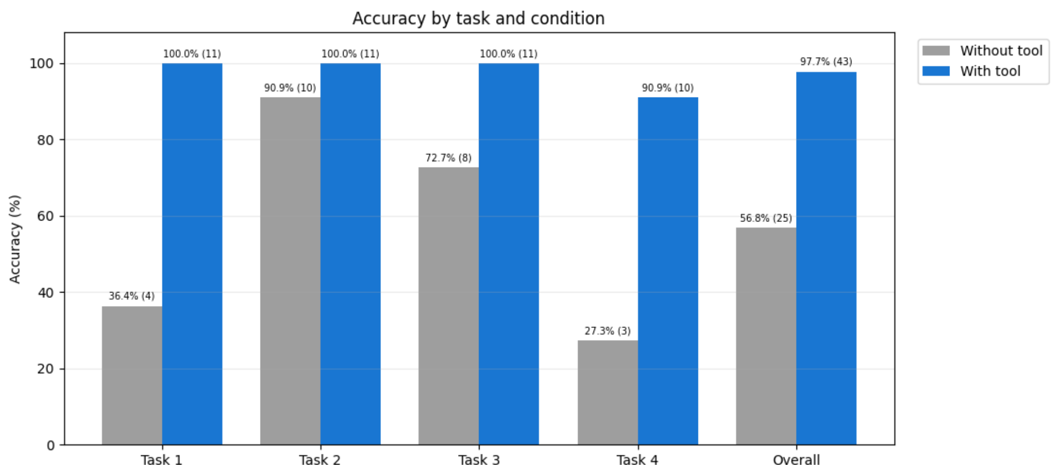


Figure 6.2: Performance accuracy scores broken down by specific tasks and the overall average, comparing participants working *with the tool* to those working *without the tool*. Data labels show the percentage accuracy and absolute number of correct responses.

	Without tool (n = 11)		With tool (n = 11)	
Task	Correct (n)	Correct (%)	Correct (n)	Correct (%)
Task 1	4	36.4	11	100.0
Task 2	10	90.9	11	100.0
Task 3	8	72.7	11	100.0
Task 4	3	27.3	10	90.9
Overall	25	56.8	43	97.7

Table 6.2: Accuracy by task and condition.

without the tool to "Extremely confident" with it. Without the tool, confidence was dispersed across all five scale points, with 11.4% of responses at the lowest level ("Extremely not confident") and only 18.2% at the highest ("Extremely confident"). With the tool, this distribution compressed toward the top of the scale: no participant reported being "Extremely not confident", and 77.3% of responses were at the "Extremely confident" level. This suggests that access to measured energy improved accuracy and gave participants a more concrete basis for high-confidence reasoning.

At the individual level, nine out of 22 participants (40.9%) reported a higher median confidence when using the tool compared to not using it. No participant reported lower median confidence with the tool. The remaining 13 participants (59.1%) showed equivalent median confidence across both conditions, likely attributable to ceiling effects among participants who were already highly confident without the tool, or to those who maintained a stable mid-range confidence in both conditions.

To address *Usability (RQ2.4)*, participants completed four 5-point Likert-scale items

	Without tool (n = 44)		With tool (n = 44)	
Response	Count (n)	%	Count (n)	%
Extremely not confident	5	11.4	0	0.0
Somewhat not confident	4	9.1	2	4.5
Neutral	6	13.6	2	4.5
Somewhat confident	21	47.7	6	13.6
Extremely confident	8	18.2	34	77.3
Median	Somewhat confident		Extremely confident	

Table 6.3: Confidence distribution by condition (all tasks combined).

in the post-task questionnaire assessing different dimensions of the tool's usability. One participant did not respond to three of the four items, yielding $n = 21$ scored responses for those items. Table 6.4 summarises the results. The results indicate a broadly positive reception across all four usability dimensions. The item with the strongest endorsement was "The output was easy to interpret", for which all 21 scored respondents agreed or strongly agreed, with a median of "Strongly agree" and 81.0% at the highest rating. This suggests that the tutorial and interface design were effective in helping participants make sense of the energy metrics. "The tool helped me understand energy consumption" also received strong agreement, with 90.9% of all 22 participants responding "Agree" or "Strongly agree" and a median of "Agree / Strongly agree". Both "The tool would be useful in real development" and "I would use this tool in practice" had a median of "Agree". For the real-development usefulness item, all 21 scored participants rated it as "Agree" or "Strongly agree" (57.1% and 42.9% respectively). However, the personal adoption item showed slightly more variability: while 76.2% of scored participants indicated they would use the tool in practice, four selected "Neutral" and one selected "Strongly disagree", suggesting that perceived utility does not uniformly translate to personal adoption intent.

Statement	SD	D	N	A	SA	(Miss.)	Median
The tool helped me understand energy consumption.	0	1	1	9	11	0	Agree / Strongly agree
The output was easy to interpret.	0	0	0	4	17	1	Strongly agree
The tool would be useful in real development.	0	0	0	12	9	1	Agree
I would use this tool in practice.	1	0	4	8	8	1	Agree

SD = Strongly disagree, D = Disagree, N = Neutral, A = Agree, SA = Strongly agree, Miss. = Missing data points

Table 6.4: Post-task Likert usability ratings (n = 21–22).

Comparing the two crossover groups on the usability items, the AB and BA groups showed consistent medians across all four statements, with no meaningful divergence. This suggests that the order of conditions likely did not influence participants' overall perceptions of the tool.

Taken together, the quantitative results show a consistent trend across all three RQs. In terms of *Identification (RQ2.2)* accuracy, the tool raised correct identification rates from 56.8% to 97.7% overall, with 16 out of 22 participants individually performing better when using the tool. In terms of *Confidence (RQ2.5)*, the median confidence rating shifted from "Somewhat confident" to "Extremely confident", with 77.3% of *with tool* responses at the highest level compared to 18.2% without it, and no participant reported lower confidence when using the tool. In terms of *Usability (RQ2.4)*, all four post-task Likert items received median ratings of "Agree" or higher, with the ease-of-interpretation item achieving a "Strongly agree" median and unanimous agreement among scored respondents. These descriptive patterns are triangulated with the qualitative findings in Section 6.3 and discussed in relation to the main research question in Section 7.1.

6.3 Qualitative Data Results

This section presents the findings of the thematic analysis conducted on participants' written task reasoning and open-ended post-task questionnaire responses, supplemented where necessary by the screen and audio recordings. Following the inductive six-phase TA framework described in Section 4.2.2, four themes were identified from the data. These themes are not mutually exclusive, as they represent overlapping dimensions of participants' experience with the tool.

During Phase 1, the complete corpus of written task reasoning and open-ended questionnaire responses from all 22 participants was read and re-read in full, with no analytical coding yet applied, in order to develop familiarity with the range and texture of the data. For Phase 2, initial codes were generated by working through each participant's written responses and labelling discrete units of meaning. To support this process, the coded extracts were transferred onto digital Post-it notes in Microsoft Whiteboard[1], enabling a visual and spatial overview of the full set of initial codes across the dataset, as can be seen in Figure 6.3. This approach to organising qualitative data is consistent with established practice in TA, where the manipulation of data fragments as discrete, movable units facilitates the pattern recognition required in subsequent phases [2]. The initial coding pass produced a set of labels including, among others: "use tool to get final choice without assumptions", "use tool to get confirmation on initial reasoning", "change reasoning based on tool feedback", "tool is good for learning", "tool is straightforward", "something is not clear enough for me", "I see areas for improvement", "tool enhanced my confidence", and "experience is fun". Initially, 14 codes were produced in the first coding pass. These were subsequently reviewed by an additional analytic auditor, who examined the coded extracts and identified instances where distinct patterns had been collapsed under a single label. Following this feedback, five further codes were added, bringing the total to 19. This step served as a form of enhancing trustworthiness in qualitative research [2], by trying to make the coding framework reflect the data more completely before thematic grouping began. In Phase 3, the Post-it notes were grouped spatially on the whiteboard by affinity, collating codes that shared an underlying meaning into candidate themes regardless of which RQ or task they originated

[1] https://www.microsoft.com/en-us/microsoft-365/microsoft-whiteboard/digital-whiteboard-app

from. In Phase 4, each candidate theme was reviewed against both the individual coded extracts grouped within it and the full dataset to verify coherence and coverage, resulting in the consolidation of several closely related codes, particularly those concerning the tool's role as a decision-making aid, into a single theme. Phases 5 and 6 involved defining and naming the four retained themes, the results being presented in the final thematic map in Figure 6.4, and writing the analytical report presented below, selecting representative extracts to illustrate each theme and relating the findings back to the five research sub-questions. Throughout the analysis, the screen and audio recordings were consulted on a targeted basis to clarify written responses that were ambiguous or insufficiently detailed, consistent with their designated role as a supplementary data source.

In the next paragraphs, we discuss the four themes that emerged from our analysis. Each one is presented with its definition, illustrated by representative participant extracts, and related back to the relevant research sub-questions. Where a qualitative finding directly corroborates or elaborates on a quantitative pattern, this is noted explicitly in support of the triangulation strategy outlined in Section 4.2.2.

Theme 1: Measurement and Reasoning. This theme captures the way participants positioned the tool as an objective source on energy efficiency, supporting or challenging their own prior assumptions with measured data. It encompasses three related but distinct patterns: participants who used the tool to obtain a definitive answer in situations where they had no prior expectation, participants who used it to validate an existing hypothesis, and participants whose reasoning was changed by the tool's output. In all cases, the distinguishing feature is the shift from subjective reasoning to grounding choices objectively, as a result of using the tool. This theme as a whole contributes to *Identification (RQ2.2)*.

The most prominent expression of this theme was participants describing the tool as resolving uncertainty that source code inspection alone could not. Several participants noted that they used the tool precisely because they felt they could not make a reliable judgment from reading the code: "*The tool showcased that it used less energy. And looking better at it I would not be able to confidently tell the difference myself*". Or that they deferred entirely to the measurement output rather than attempting to reason from the code: "*i [sic] used the tool, tried isolating the case, used the output to form judgement*", "*The tool showed 13.3J vs 0.2J for A and B respectively, which leads me to think B is more efficient*", and "*The tool said so. Makes sense if the read operation work on contiguous blocks.*" This pattern is reflected in the quantitative accuracy data for Tasks 1 and 4, where without-tool accuracy was 36.4% and 27.3% respectively. The qualitative data suggests that participants knew they were guessing, and for some of them the tool replaced the guesswork with certainty. This is a contribution to *Integration (RQ2.3)* as well, revealing that one of the integration patterns was to measure first, since some participants chose to directly use the tool rather than attempting to reason themselves.

A second, equally important pattern within this theme involved participants who did hold a prior expectation about which implementation would be more efficient and used the tool to test that expectation. This occurred most clearly in Tasks 2 and 3, where algorithmic complexity reasoning pointed clearly toward the correct answer. Participants articulated this explicitly: "*set insertion is very efficient. the [sic] problem can be solved with just set insertion, until we find the duplicate. that's [sic] much better than using a list (o (n^2)).*

Figure 6.3: Visualisation of the inductive initial (or open) coding results.

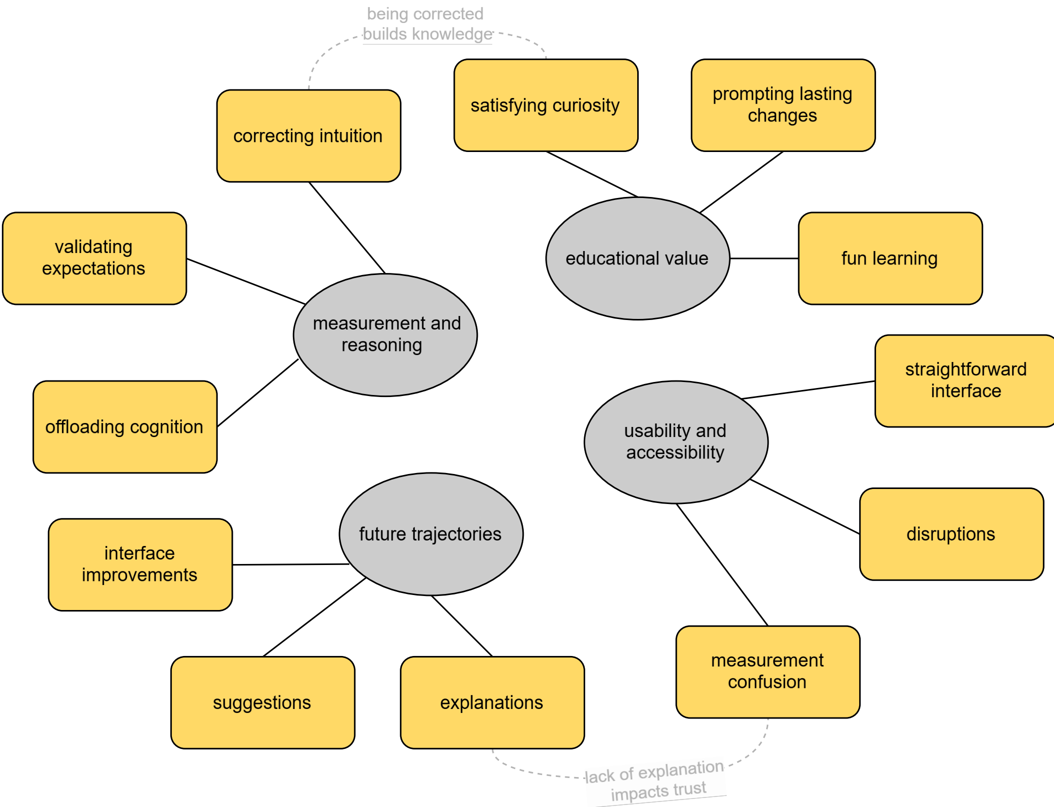


Figure 6.4: Final thematic map of participant experiences with the energy-analysis tool, illustrating core themes, sub-patterns, and inter-thematic relationships.

the [sic] tool confirms this. i [sic] trust the tool", "*I found the tool useful in confirming my initial analysis. For example, I was able to see when one solution has quadratic time complexity and another linear, but being able to run them made me even more certain*", and "*Usually (at least for big numbers) algorithmic runtime should still dominate the actual execution. In this case O(n) vs O(n^2), as the tool also reassures me I'm certain of this*". Validating expectations also came up in the other tasks: "*So I suspected that using string builder would be more energy efficient, and this was confirmed when I used the plugin*" and "*i [sic] already had a very strong idea of version b [sic] being better, as I remember string copying is absolutely terrible and you should always use a better library (like String Builder). the [sic] tool confirmed my suspicion and made me feel very smart. thank [sic] you tool!*". This pattern carries significance for *Confidence (RQ2.5)* where the quantitative results show that 13 participants maintained equivalent median confidence across both conditions, which could suggest the tool had no effect on confidence. However, the qualitative data offers a more nuanced interpretation in the sense that for participants who were already highly confident because their knowledge pointed them in the right direction, the tool's role was not to increase confidence but to validate it, reinforcing trust in both their own reasoning and the tool's output simultaneously. This also speaks to *Interpretation (RQ2.1)* as the tool's

output was legible enough that participants could recognise when it aligned with theoretical predictions, suggesting that the metrics were being correctly understood.

A third expression of this theme arose when the tool's output contradicted participants' initial reasoning, prompting them to revise their understanding. This occurred in tasks where intuitive heuristics, such as assuming simpler-looking code is more efficient, or that a recursive implementation implies performance, led participants to say: "*I thought that simpler code is more efficient, but most probably the string class of java [sic] is very well optimized since it used a lot less energy to run*", "*Even though I'd think that the building/initializations of objects is more costly in Java, Java might use compiler magic to still make it more efficient. I trust the measurements*", and "*I thought recursion would be more memory efficient, but in hindsight, after using the tool I get that the repeated calls would probably make it less efficient*". This explains the quantitative pattern observed in Task 1, where *without the tool* accuracy was lowest (36.4%) despite participants in many cases feeling confident in their incorrect answer. The tool provided information and had a corrective function. Notably, one participant also described a second-order learning effect from this correction, as having been wrong once, they applied more scepticism to their reasoning in subsequent tasks: "*I believe my response to the second exercise was heavily influenced by the result of the first one. My intuition was off and that made me think that I was more likely going to be wrong on the second as well, but that wasn't the case. But overall the fact that I was wrong made me learn something new and interesting, which is something that I didn't expect to happen when I learned about the tool*", which speaks directly to Theme 2 below.

Trust in the tool appears to have been calibrated against participants' existing knowledge. Those whose prior expectations were confirmed by the measurements trusted it, but those who encountered unanticipated output experienced doubt: "*PairA always access one element before finishing the calculation of one k loop. I am not really convinced by the tool*". This suggests that the tool's perceived trustworthiness is currently dependent on the user already possessing a frame of reference for what a plausible result looks like, which can be a meaningful limitation for a tool intended to serve developers who lack that prior knowledge.

Theme 2: Educational Value. This theme captures feedback in which participants described the tool as having an educational function that extended beyond the immediate task of choosing between two implementations. It encompasses the insight participants gained about specific algorithms or data structures during the study, and the broader effect the tool had on their approach to writing code and evaluating efficiency assumptions in the future.

Several participants described the tool as satisfying a kind of intellectual curiosity about program behaviour that they had not previously had a means to explore: "*I was curious about the results, so I used the tool to see the actual consumption, and it was a nice learning opportunity*", "*Yes, it has a clear energy indication and is easy to use. it also helps me learn more about how a small change has a big impact on the energy consumption*", and "*it was very useful to satisfy my curiosity about program behaviour*". This educational dimension is relevant to both *Interpretation (RQ2.1)* and *Identification (RQ2.2)* since participants were reading off a result but also possibly internalising the relationship between code structure and energy cost. The tool made abstract efficiency concepts more tangible by attaching values to specific implementation choices, which is a qualitatively different form of feedback from only asymptotic complexity analysis.

A particularly notable subset of responses described the tool as having a lasting effect on how participants would approach coding decisions in the future, suggesting that the learning was not confined to the study session: "*I thought recursion would be more memory efficient, but in hindsight, after using the tool I get that the repeated calls would probably make it less efficient*" and "*I found the tool to be useful because just having in mind that such a tool exists and that I have access to it when needed helps me make more conscious decision about every implementation details*". This connects directly to *RQ2*, since the tool seems to help developers understand energy efficiency during a task but also prompts a change in how they approach efficiency reasoning, extending its value beyond the individual use instance.

Several participants described their experience in affectively positive terms: "*It is pretty cool, especially for learning purposes*", "*Cool tool!*", and "*it was fun*". And although enjoyment is not a primary research outcome, it is relevant to *Usability (RQ2.4)* and adoption because a tool that participants find engaging is more likely to be voluntarily integrated into the development workflow.

Theme 3: Usability and Accessibility. This theme captures participants' overall assessment of the tool's usability, which was predominantly positive, alongside some specific points of confusion or friction that disrupted an otherwise smooth experience. The theme reflects a division in the usability feedback: the energy metric output and its interpretation was broadly considered clear and easy to use, but the mechanics of measurement, indicator visibility, and output reliability generated a distinct subset of opinions.

The dominant usability experience was one of accessibility and low learning curve. Participants consistently described the tool as intuitive and straightforward to operate after the tutorial: "*Yes, it has a clear energy indication and is easy to use*", "*I liked how intuitive the tool was, didn't feel it had a big learning curve*", "*it was very easy to use and intuitive*", "*I would change nothing*", "*I found the tool to be very usable and easy to understand*", and "*The tool was clear after the tutorial phase*".

Despite the broadly positive reception, a recurring source of friction was uncertainty about what the energy measurements actually represented and whether they could be trusted across different contexts: "*Whether the results would differ on different PCs or even on clusters*" and "*How exactly the energy consumption is calculated and if it is system specific*". These findings are relevant to *Interpretation (RQ2.1)* since the metric format was considered clear, the meaning of absolute values and their variability was not always fully interpretable.

Beyond the conceptual uncertainty around measurement, participants identified several concrete interface issues that disrupted the experience: "*It is a bit difficult to see when the tool has started. as [sic] it only has the line indication on the bottom right. and im [sic] impatient usually so i click it alot [sic]*", "*Error outputs could be more distinct. The icon is weird*", "*Not much, just that it sometimes closes the window without me having pressed anything. I could see myself getting annoyed if I were to use it all the time then.*" "*maybe just a bit of confusion around the tool slowness and if it was gonna succesfully [sic] run or not. only [sic] seeing the blue, nondescript loading bar did create some confusion*", and "*I think there is no typographic hierarchy introduced in the output design, making it intuitive to use only when the user knows what to look for. As a non-programmer I would like to see the energy output not only as a different color (as it is now) but also bigger and bolder text that stands out. Additionally I think the loading that happens when the*

tool is working is quite small, what might have to do with the UI design of the compilation program, but I would prefer to be more aware of it running instead of having to look for the loading bar in the thin right corner". One participant articulated observing unexpected tool behaviour: "*i [sic] thought that i [sic] can queue the tasks, instead they run in parallel which would mess with the results because it would compile one while measuring for the other*". The friction points of indicator visibility, error handling, and intermittent instability are consistent with the *Usability (RQ2.4)* finding that four participants selected "Neutral" and one selected "Strongly disagree" on the personal adoption item, even while agreeing the tool was useful in principle. The gap between perceived usefulness and personal adoption intent, visible in the quantitative data, could therefore be explained in part by these localised but impactful usability concerns.

Theme 4: Future Trajectories. This theme captures the forward-looking feedback participants provided: what they wished the tool already did, what would make it more useful in real development contexts, and what additional information or functionality they felt was missing. Unlike the preceding themes, which describe participants' experience of the tool as it currently exists, this theme reflects their vision of what it could become. It is most directly relevant to *Usability (RQ2.4)* and to the implications discussed in Section 7.3.

The most frequently expressed desire was for the tool to go beyond reporting how much energy a snippet consumed and to explain why: "*Maybe an overview of the operations that require the most energy to execute would give even more insight*", "*I think the tool is good as is, maybe would be better [...] when selecting many lines, highlighting function calls or lines that significantly hurt energy efficiency*", "*If it would come up with an explanation why the energy value is this much, it would be easier to remember to apply the most energy methods next time I use something similar*", "*Maybe add more reasoning as to why a certain piece of code consumes energy and if possible suggest more efficient alternatives*", and "*It would be nice to get a more fine-grained explanation of where the energy consumption comes from*". This is a meaningful gap, particularly in light of Theme 2 because if the tool's educational value is one of its most appreciated qualities, then adding explanatory context would directly amplify that value. Perhaps the following quote is the most interesting one in this context, "*i [sic] would also like the tool to further attempt to break down the energy usage (energy due to memory or time efficiency?) I also need to trust the tool and can not verify if it's really real*", as it implies that having the said energy measurement breakdown would increase trust in its output. It could therefore help address the trust calibration problem identified in Theme 3 where a participant who does not already know why $O(n^2)$ might be less energy efficient than $O(1)$ would benefit from an explanation attached to the measurement, rather than being left to infer it from the values alone.

A related but distinct request was for the tool to move from diagnosis to prescription, meaning not just identifying which implementation is less efficient, but also suggesting what to do about it: "*maybe would be better if it also provides recommended changes to make the code more energy efficient*" and "*Maybe implement a suggestion provider for efficiency improvement*". This reflects the trajectory from energy awareness to action, since the tool as currently designed seems to be closing the measurement gap but leaves the remediation step to the developer.

Participants also proposed more granular interface improvements: "*only to be very nit-*

picky it would be nice to delete individual responses next to the clear all button", "*When you 'cancel' the tool, it should still output the measurements. Such that you can at least output a vibes based energy consumption.*", and "*Maybe auto-select all lines you have text from (so you can't mess up by missing a letter)?*". These requests collectively point toward a more polished experience. Several of them directly address the friction points identified in Theme 3, suggesting the possibility that participants who experienced confusion or instability were not deterred from the tool's concept but were motivated to articulate how to fix the specific issues they encountered.

6.4 Summary

RQ2 Summary: The thematic analysis yields four themes that together provide a qualitative account of how participants experienced the energy-analysis plugin. Theme 1 explains the primary mechanism by which the tool influenced task performance. It replaced reasoning based on assumptions with real measurement, either by providing a definitive answer where none was available or by confirming or correcting existing intuitions. This theme is the qualitative counterpart of the accuracy gains observed in the quantitative results and addresses *Interpretation (RQ2.1)*, *Identification (RQ2.2)*, *Integration (RQ2.3)*, and *Confidence (RQ2.5)*. Theme 2 reveals a dimension of the tool's value that the quantitative measures did not fully capture. This represents the tool's capacity to build energy intuition, shift participants' self-perception, and create an engaging experience that participants described as "*fun*" and "*cool*". This theme is most relevant to *RQ2* and to *Usability (RQ2.4)*, insofar as perceived educational value was associated with higher reported adoption intent. Theme 3 triangulates the quantitative usability finding that ease of interpretation was the strongest-rated dimension while adoption intent showed the most variability. The qualitative data suggests that this variability could be explained not by a rejection of the tool's purpose but by specific, addressable concerns such as trust calibration rooted in insufficient explanatory context and interface-level friction. This theme addresses *Interpretation (RQ2.1)* and *Usability (RQ2.4)*. Theme 4 captures participants' forward-looking feedback and points toward next development directions, such as adding explanatory context to measurements, providing actionable efficiency recommendations, and refining the interface. This theme speaks primarily to *Usability (RQ2.4)* and informs the practical implications discussed in Section 7.3. Across all four themes, the qualitative findings are consistent with and elaborative of the quantitative results, supporting the triangulation approach adopted. Where the quantitative data showed what happened, meaning accuracy improved (from 56.8% without the tool to 97.7% with the tool), confidence increased (median rating shifted from "Somewhat confident" to "Extremely confident"), usability was positively rated ("Agree" as median), the qualitative data gives potential reasons as to how and why it happened, and identifies the specific mechanisms, limitations, and directions for improvement that descriptive statistics alone cannot reveal.

Chapter 7

Discussion

7.1 Revisiting the RQs

> **MRQ:** How technically and practically feasible is it to implement a lightweight, software-based energy measurement tool for Java code snippets that provides immediate, reliable energy measurements to support developer reasoning about Java code energy efficiency?

Each dimension of this question is addressed by a distinct part of the thesis. Technical feasibility is grounded in the related work (Chapter 2), which establishes the necessary technology. Although the individual components exist, as far as we know, no tool previously combined them into a lightweight, in-IDE measurement workflow. Practical feasibility, in the sense of whether such a tool can actually be built and deployed, is shown through the implementation (Chapter 3). We proposed an architecture consisting of a standalone REPL backend interfacing directly with `powercap`, combined with an IntelliJ plugin that invokes it via a subprocess. We showed this approach to be sufficient to handle a wide range of real Java code scenarios, including cross-file dependencies, runtime exceptions, and classpath resolution. Whether the proposed tool delivers on its two core goals (i.e., the reliable measurements and the support for developers' reasoning on energy efficiency) is then answered empirically. Immediate and reliable measurements are addressed by the verification phase (Section 4.1 and Chapter 5), while support for developer reasoning is addressed by the validation phase (Section 4.2 and Chapter 6), both revisited below. Taken together, these chapters jointly suggest a positive answer to the *MRQ*, but the nature of that answer deserves some reflection. The tool is technically grounded and practically buildable, but it rests on a measurement foundation that is inherently probabilistic. However, this characteristic is not unique to our proposed tool, but reflects a fundamental property of system-level energy measurement. As related work acknowledges, this need not be disqualifying since understanding trends matters more than knowing raw values [18]. The fact that developers can and do reason in this way when given the data is suggested by the user study results. Ultimately, the main boundary conditions when addressing our *MRQ* are the Linux and RAPL-compatible hardware requirements, and the single-machine hardware dependency of the measurements,

but we consider these constraints scope the feasibility claim without invalidating it.

> **RQ1:** Does the proposed energy measurement tool provide sufficiently reliable measurements such that statistically significant energy consumption differences between semantically equivalent but syntactically distinct Java code snippets can be detected on the tested hardware configuration?

For our verification phase (Chapter 5), across 30 LeetCode algorithm pairs, the tool detected statistically significant energy differences in 25 of 30 cases (83.3%), with both time and energy following the expected direction jointly in 24 of 30 pairs. The *Baseline* (*RQ1.1*) was satisfied in 28 of 30 cases (93.3%), showing that the chosen inputs reliably reproduced the performance discrepancies reported by LeetCode. *Sensitivity* (*RQ1.2*) suggests that when the *better* implementation runs faster under the chosen input, the tool tends to reflect that in energy as well. For *Magnitude* (*RQ1.3*), across the 30 pairs, energy effect sizes were distributed as 19 *large*, 3 *medium*, 3 *small*, and 5 *negligible*, compared to 25 *large* for time. Following the effect size results, we can think that if execution time and energy consumption were perfectly identical, meaning, for example, that a 50% reduction in time always resulted in exactly a 50% reduction in energy, a dedicated energy measurement tool would be redundant. If this were the case, developers could simply rely on measuring time to estimate energy efficiency. The findings from the effect size analysis shape the argument for the existence of energy measurement tools in the sense that a massive reduction in execution time does not guarantee a proportional, or even meaningful, reduction in energy consumption. For example, a developer relying only on cloud-platform time metrics like the ones provided by LeetCode would falsely assume substantial energy savings in the case of the *SlidingWindowMaximum* problem. Finally, for *Correlation (RQ1.4)*, median execution time and median energy consumption were strongly positively correlated across the 60 implementations, showing that energy broadly scales with time. However, the scatter around the fitted relation suggests that time alone is an imperfect proxy for energy, reinforcing the value of dedicated energy measurement rather than relying on time.

> **RQ2:** How does the proposed energy measurement tool affect developers' ability to understand, reason about, and make decisions regarding the energy efficiency of Java code snippets?

The validation phase (Chapter 6) provides convergent evidence across quantitative and qualitative data that the tool improved developer performance on energy-related reasoning tasks, as shown through a mixed-methods, within-subject user study with 22 participants applying an AB/BA crossover design.

Interpretation (RQ2.1) was perhaps the most nuanced sub-question to evaluate, as it concerns not just whether participants read off the correct number, but whether they understood what it meant. The quantitative data alone cannot answer this, but the qualitative evidence suggests interpretation was broadly successful as participants consistently described recognising when the tool's output aligned with or contradicted their theoretical hypotheses, and

those whose expectations were confirmed articulated why the result made sense. That said, a recurring source of friction was uncertainty about the absolute meaning of the values, particularly whether results would differ across machines or how the attribution was computed. This suggests the tool's output format is legible enough for relative comparisons, which is its primary intended use, but that the absence of explanatory context can create a trust gap for users who cannot independently verify the measurements.

Identification (RQ2.2) showed the clearest quantitative effect, as accuracy rose from 56.8% to 97.7% overall, with 16 of 22 participants individually performing better in the *with tool* condition. It is worth reflecting on what the 56.8% *without the tool* baseline means. For a binary choice task, this is only marginally above chance, which is an interesting finding in itself, as even technically literate participants with Java experience could not reliably identify more energy-efficient implementations from source code alone. This strengthens the argument for the tool's existence beyond just improving accuracy rates. The one remaining error in the *with tool* condition, occurring on Task 4, is also instructive since the participant explicitly reported choosing to trust their own intuition over the measurement data, which raises a broader question about when and why developers might discount tool feedback, something worth investigating in future work.

Integration (RQ2.3) revealed that participants did not converge on a single way of using the tool, which is arguably a positive signal. Three distinct patterns emerged: measuring first without attempting prior reasoning, using the tool to validate an existing hypothesis, and revising incorrect intuitions after seeing contradictory output. The third pattern is particularly significant for the tool's educational value. When a participant who was confidently wrong on Task 1 subsequently applied more scepticism to their Task 2 reasoning, it suggests that even a single corrective measurement experience can shift reasoning behaviour within the same session. Whether this effect persists beyond the study context remains an open question, but it points to a potential long-term benefit of the tool.

Usability (RQ2.4) was rated positively overall, but the gap between the ease of interpretation item, which received unanimous agreement, and the personal adoption item, which had four neutral and one strongly disagree response, is worth unpacking. The qualitative data suggests this gap is not explained by doubts about the tool's concept, but by specific friction points such as the subtlety of the progress indicator, occasional interface instability, and the lack of typographic hierarchy in the output. These are implementation-level concerns rather than fundamental limitations of the approach, which is an encouraging finding in that they are addressable. It does, however, reinforce a broader principle from the related work that even useful tools will not be adopted if the friction of use is perceived as too high relative to the benefit, particularly for tasks like energy measurement where previous research has found that developers do not yet consider it part of their standard workflow [18, 42, 56].

Confidence (RQ2.5) shifted from a median of somewhat confident to extremely confident, with 77.3% of *with tool* responses at the highest scale level. An interesting nuance here is that 13 participants showed equivalent median confidence across both conditions. Rather than interpreting this as evidence that the tool had no effect on their confidence, we can also consider that for participants whose prior knowledge already pointed them in the right direction, the tool's role was not to raise confidence but to validate it. This distinction matters because it means the tool serves two different user needs simultaneously, providing

certainty where it is absent and verification where a prior hypothesis already exists, without undermining either group's reasoning process.

7.2 Implications

This section reflects on how researchers, educators, and practitioners can draw on our findings to advance energy-aware software development.

For researchers, our work responds to a long-standing call in the energy efficiency literature for lightweight measurement tools integrated into development environments. Prior research has consistently shown that software developers need direct feedback on energy consumption [44, 56, 74], yet most tools measure aggregate application energy rather than consumption at the code level. By showing that fine-grained, in-IDE measurement is both technically feasible and empirically beneficial, our research strengthens the conceptual argument that energy awareness should be treated as a primary concern throughout the software lifecycle [18]. Moreover, our user study opens new research directions around measuring whether tool-driven awareness translates into sustained behavioural change and how various developer populations interact with energy feedback differently.

For educators, the gap between developer intuition and actual energy behaviour revealed in this work has direct implications for software engineering education. The finding that participants without tools performed at a level almost equivalent to chance despite many having completed formal coursework on energy or performance optimisation suggests a disconnect between conceptual knowledge and practical reasoning, which educators can use as motivation to redesign how energy topics are taught. Although we have already established that our findings only apply to our specific sample of 22 participants, the diversity in how they used the tool does offer some pedagogical insight, since students could have heterogeneous learning styles regarding energy. It might be that integrating lightweight energy measurement tools into programming courses and development environments can establish early habits of energy-conscious thinking. However, the participant who explicitly discounted measurement data reminds us that tools alone do not guarantee better reasoning, meaning that educators should cultivate critical interpretation skills alongside tool use. Given the concentration of energy knowledge in research and academic contexts, educators bear particular responsibility for democratising energy awareness by incorporating energy-aware development into standard curricula alongside performance, security, and correctness.

For practitioners, our research shows that energy-aware decision-making is achievable without specialised knowledge when appropriate tools are available. However, they must understand the constraints, as energy measurements remain inherently probabilistic and hardware-dependent. Practitioners working with heterogeneous deployments, closed-source systems, or non-Linux platforms should view this work as establishing feasibility for their own context rather than expecting direct applicability.

7.3 Limitations

The current implementation is a research prototype. Several engineering decisions that were acceptable during development, most notably the hardcoded path to the compiled backend JAR, would require generalisation before the tool could be distributed beyond the research context. Similarly, the absence of a persistent installer or a plugin marketplace release means that deployment currently depends on manual setup by the end user.

The PLATYPUS-related restrictions [31], resulting from a vulnerability where attackers exploit unprivileged access to energy counters to infer sensitive data, require the user to manually modify `powercap` file permissions on every system boot before the plugin can function. A permanent `udev` rule would resolve this but was intentionally omitted from this prototype since persistently exposing energy counters reduces friction but also permanently widens the attack surface, which is an acceptable trade-off only in a production-ready release and not during development where the manual per-boot permission step was preferred as a more conservative default. Until a `udev` rule is added, the tool's usability is constrained to sessions where the developer has already performed this manual step.

The tool spawns a fresh backend JAR instance per measurement, introducing JVM initialisation latency on every invocation. For very short-lived snippets, this overhead is non-trivial relative to the measurement window. A persistent backend process with an inter-process communication channel would address this but was considered out of scope.

Each plugin invocation is fully stateless, as variables and class definitions from a previous measurement are not carried over. Although the plugin's dependency resolution automatically handles static context such as necessary imports and project classpaths, it cannot reconstruct prior runtime state. Consequently, users must ensure each selected snippet is entirely self-contained, which may require refactoring the code to include local object instantiations. The interactive backend mode does support persistent state within a session, but this is not exposed through the IDE frontend.

The attribution model tracks the CPU time of the JShell worker PID discovered at startup. Energy consumed by child threads spawned by the snippet is attributed to their own PIDs and excluded from the reported value. This is not a concern for the single-threaded snippets evaluated here, but represents a meaningful boundary for concurrent Java code.

If a selected code snippet calls `System.exit(0)`, the JShell worker process terminates before the `EnergyMonitor` captures the end-state RAPL counter reading. The plugin does not crash, but the user receives no energy metrics for that run. This edge case is currently not surfaced with a dedicated warning message, meaning the user may be unaware of why no measurement was reported. An earlier iteration of the plugin caused the IDE itself to crash in this scenario, meaning the current behaviour is an improvement, but a clear error message and handling remain outstanding.

Code that relies on reflection may behave unexpectedly within the JShell execution environment, as the context imposes constraints on dynamic class loading and access to private members that do not apply in a standard compiled application. This is a boundary of the JShell-based approach rather than of the energy measurement mechanism specifically, but it limits the range of real-world code snippets the tool can reliably evaluate.

7.4 Threats to Validity

Following the framework established in the methodology (Sections 4.1 and 4.2), we organise threats to validity into three categories: internal, external, and construct. For each threat, a mitigation strategy is described where one was applied, but where mitigation was not feasible within the scope of this work, the concern is noted as a direction for future validation.

7.4.1 Internal Validity

The RAPL counter captures package-level energy for the entire chip, meaning background system activity and thermal fluctuations from dynamic voltage and frequency scaling (DVFS) introduce noise into every measurement window. This is reflected empirically in the bimodal energy distributions observed for several problem pairs in Chapter 5, suggesting that some runs occurred under different thermal or frequency states than others. To mitigate this, we tried to keep experimental conditions constant across all 1800 runs (e.g., background programmes, battery percentage, and charging state), measurements were taken in batches of 10 with cooldown periods between batches, and outliers deviating more than three standard deviations from the sample mean were removed following energy measurement recommendations by Cruz [10]. Despite these precautions, 75% of solutions failed the Shapiro-Wilk normality test, indicating that residual environmental variability could not be fully eliminated within the experimental constraints.

The prevalence of non-normal distributions required defaulting to the Mann-Whitney U test for all problem pairs to maintain analytical consistency. A theoretical cost is reduced statistical power for the pairs whose distributions were demonstrably normal. However, as observed by Montgomery and Runger [36], the Mann-Whitney U test "*is approximately 95% as efficient as the t-test in large samples*" when the normality assumption is correct and "*will always be at least 86% as efficient*" regardless of the form of the distributions, making the practical impact of this choice minimal.

All 22 user study sessions were conducted in-person with the researcher present. Participants may have moderated their behaviour due to the awareness of being observed, a form of social desirability effect sometimes referred to as the Hawthorne effect [28]. In particular, it is possible that participants may have engaged more carefully with the tool or may have articulated their reasoning more thoroughly than they would have in a natural setting. The think-aloud protocol may also have amplified this effect by requiring participants to externalise reasoning that would ordinarily be implicit. This threat cannot be fully mitigated in a study of this kind, but future work could, for example, employ diary-based methods to provide a complementary perspective.

The AB/BA counterbalancing distributes learning effects evenly across groups, but reasoning strategies cannot be fully unlearned between blocks. A participant corrected by the tool in their first block may apply scepticism to their intuitions in the second, as was observed qualitatively. This carryover effect is an acknowledged limitation of within-subject designs in software engineering experiments [68] and represents a residual internal validity threat that the counterbalancing mitigates but does not eliminate.

Thematic analysis is inherently interpretative. To improve reliability, a triangulation strategy was applied across written responses, audio and screen recordings, and quantitative results, and an additional analytic auditor reviewed the coding for discrepancies, as it is recommended by Braun and Clarke [3], expanding the initial code set from 14 to 19 codes. Despite this, the possibility that a different researcher would have constructed a different thematic structure from the same data cannot be excluded.

7.4.2 External Validity

All measurements were collected on one machine, meaning that the energy values reported by the tool are hardware-specific and will differ on other machines, even those running the same code. The verification findings, which concern the tool's ability to detect relative differences between implementations, are more likely to generalise than the absolute values, but this has not been validated across configurations. The hardware requirements further restrict the population of developers for whom the tool is accessible, and therefore limits the external scope of both the verification and validation results.

The tool was designed for and evaluated exclusively on Java code, and the verification dataset consists entirely of algorithmic problems sourced from LeetCode. LeetCode problems are self-contained, computationally intensive, and well-suited to the snippet-based measurement model. They are not representative of enterprise software development, where code is typically stateful, distributed, or tightly coupled to frameworks and databases. Although we have tried the tool on a suite of over 40 distinct code scenarios, it is still unclear whether the results from Chapter 5 would hold for more diverse code. Extending the evaluation to other languages and problem domains is a potential direction for future work.

The user study was conducted with 22 participants, recruited from the researchers' personal network, of whom 81.8% were CS, DS, or AI students. They are technically literate, and although many have been exposed to the theory of energy-aware development through coursework, self-reported data confirms they remain largely inexperienced with applied energy optimisation in practice (72.7% having never performed it). The relatively small sample size is not representative of the broader population of professional software developers, meaning that the quantitative results are reported as descriptive statistics and no inferential claims are made about the wider developer population. The extent to which the accuracy gains, confidence shifts, and usability assessments observed in this study would replicate with a more diverse sample, including senior engineers, developers from non-CS backgrounds, or practitioners working in energy-constrained domains, remains an open question.

Participants chose between two pre-written, self-contained implementation pairs under in-person conditions with a researcher present. This scenario is intentionally simplified relative to real-world energy-aware development, which involves writing, iterating, and profiling code over extended periods, often without a clear comparison point already available. The tool's effectiveness in a more natural development context, where the developer must decide when to measure, what to select, and how to act on the results without external guidance, was not evaluated. A longitudinal study in which developers use the tool freely during their own projects would provide stronger evidence for the practical value suggested by the current findings.

7.4.3 Construct Validity

The attribution model estimates snippet energy as the total package energy delta multiplied by the snippet's proportional CPU time share, following Noureddine et al. [40]. This assumes energy scales linearly with CPU time, which is not always the case, since memory-bandwidth-bound workloads consume energy through DRAM without necessarily increasing CPU time [47], and DVFS means a process at a higher clock frequency consumes disproportionately more energy per CPU-time unit [29, 48]. However, this is a fundamental boundary of software-based attribution models rather than a flaw specific to our implementation.

A primary risk in establishing the experimental ground truth comes from sourcing solutions based on the runtimes reported by LeetCode. Because the execution environment of the platform is influenced by its own server's availability and concurrent user load, the performance metrics it reports are non-deterministic. Moreover, in 2024, LeetCode changed how they measure and display code performance to allow for more accurate comparisons, causing some solutions to show misleading metrics as they are compared to older solutions from the time before the changes had taken effect [1]. However, LeetCode stated that these comparisons would eventually stabilise as more users submit code under the new system. This means that submitting the exact same code twice can yield drastically different performance rankings. To prevent these non-deterministic platform metrics from compromising the experiment, the methodology does not rely solely on the rankings given by LeetCode. Instead, we manually assess both the time and space complexity of the chosen solutions to ensure that a genuine difference exists between the better and worse implementations.

The `powercap` counter updates once per millisecond. For snippets executing faster than this interval, the counter may register no increment, yielding a near-zero measurement. However, considering the verification results, if the limitation affects both implementations equally, the findings remain valid even when absolute values are imprecise, consistent with the principle that energy trends matter more than raw values [18].

All measurements were collected on a single AMD Ryzen 9 5900HX. Although AMD processors from Zen onwards expose a RAPL-compatible interface, energy unit granularity and counter update frequency may differ between AMD and Intel implementations [50], meaning specific energy values are not directly comparable across architectures. Future work should repeat the verification on an Intel processor to establish whether the sensitivity findings generalise across RAPL implementations.

7.5 Ethical Considerations

Given that this research involves both human research subjects and data collected from a commercial platform, ethical considerations were incorporated at each stage of the research design. Formal approval for the user study was granted by the TU Delft HREC.

Recruiting subjects from a personal network introduced the risk that participants might feel socially obligated to participate, or that the presence of a supervisor who could also serve

[1] https://leetcode.com/discuss/post/5947534/inconsistent-execution-time-reporting-0m-rra1/

as a future assessor might create perceived academic pressure. All recruitment materials made it explicit that participation was purely voluntary and that declining or withdrawing carried no consequences. Participants were informed of the study purpose, data collection procedures, and their withdrawal rights verbally before being asked to sign a consent form.

Contact details on consent forms, audio and screen recordings were stored separately from raw research data on TU Delft-approved storage, accessible only to the research team. Because the study group was small, unique phrasing in published quotes could potentially identify individuals, so all data presented in the thesis and any future publications is therefore anonymised and attributed to pseudonyms.

Raw audio and screen recordings are not included in the replication archive, as they represent a data source that could re-identify participants. Anonymised written responses and aggregated quantitative results will be shared, but recordings will be restricted and destroyed after the retention period specified in the submitted TU Delft Data Management Plan (DMP).

Sourcing experimental data from a commercial platform introduces the risk of violating platform policies, which could raise ethical or legal concerns about the research data. All problem selection, solution examination, and interactions with the LeetCode platform were conducted in strict adherence to the LeetCode Terms of Service.[2] In particular, no automated scraping or programmatic access was used and all solutions were manually selected and recorded to comply with the platform's restrictions on data extraction.

A further consideration concerns the energy overhead introduced by the tool itself. Performing energy measurements requires executing code under instrumentation, which consumes additional resources beyond what the program would use in normal operation. This creates a tension where the tool uses energy in order to raise awareness of energy consumption. This is considered justifiable on the grounds that the insight gained from a measurement session can inform implementation decisions that reduce energy consumption over the full lifetime of a software system, where the cumulative savings are likely to outweigh the cost of profiling. Nevertheless, this trade-off should be acknowledged, and future work could investigate the tool's own energy footprint to make this overhead more transparent to users.

7.6 Use of Generative AI

In accordance with TU Delft guidelines on the use of generative AI (GenAI) in academic work, I hereby declare the following uses of AI tools during the preparation of this thesis:

- support for programming tasks, including the generation of boilerplate code and debugging hints, but all generated code was reviewed, tested, and validated by the author before incorporation into the project;
- generation of example scripts for data processing, analysis, and visualisation, but all results and their interpretations were independently verified and are the sole responsibility of the author;
- paraphrasing and rewording of notes for clarity;

[2] See: `https://leetcode.com/terms/`

- assistance with formatting tasks such as producing tables and converting text between styles, but all substantive content, arguments, and conclusions in this thesis are the author's own;
- assistance in retrieving content from relevant literature through Retrieval Augmented Generation (RAG) tools, but all retrieved references were independently located, read, and verified by the author before citation.

Chapter 8

Conclusion

As the software engineering community increasingly seeks to minimise the environmental footprint of digital infrastructures [70, 60], developers require accessible mechanisms to evaluate the energy efficiency of their code [56, 44, 74]. We have carried out an exploratory study to investigate the technical and practical feasibility of a lightweight, software-based energy measurement tool for Java code snippets. By engineering an IntelliJ IDEA plugin backed by a JShell execution environment and the Linux `powercap` framework, we provided developers with immediate, in-situ energy feedback directly within their workflow.

Through a verification phase involving 30 algorithmic problem pairs, we showed that the tool reliably detects statistically significant energy consumption differences between semantically equivalent but syntactically distinct implementations. Furthermore, our mixed-methods validation study involving 22 participants illustrated the impact of this immediate feedback on developer behaviour and reasoning. When relying solely on source code inspection, participants correctly identified the more energy-efficient implementation in 56.8% of cases, but with the measurement plugin, accuracy increased to 97.7%. Alongside this improvement, participants reported significantly higher confidence in their decisions. Qualitative insights further revealed that the tool assisted in correcting flawed intuitive assumptions and provided educational value, helping participants internalise the relationship between code structure and energy cost.

Several limitations constrain the scope of these findings. The tool currently requires Linux and RAPL-compatible hardware, manual permission configuration on each system boot, and a hardcoded path to the backend JAR, all of which must be resolved before broader distribution. Measurements are hardware-specific and were collected on a single machine, meaning that absolute values are not directly transferable across configurations, though relative comparisons between implementations on the same machine should remain valid. The verification dataset consists exclusively of algorithmic LeetCode problems, which are computationally self-contained and may not represent the complexity of real enterprise code. The user study sample of 22 participants, predominantly CS students, does not allow inferential claims about the wider developer population.

Future work should address these limitations along several directions. First, packaging the plugin for distribution via the JetBrains Marketplace, with a bundled backend and a persistent `udev` rule for permission management, would remove the primary barriers to adop-

tion. Second, extending the verification to a broader range of code domains, including I/O-bound, concurrent, and framework-heavy snippets, would establish whether the sensitivity findings generalise beyond algorithmic problems. Third, incorporating explanatory context into the tool's output, such as attributing energy to specific operations or suggesting more efficient alternatives, would directly respond to the most frequently expressed participant request and would amplify the educational value that emerged as one of the tool's strongest qualities. Fourth, a longitudinal study in which developers use the tool freely during their own projects would provide stronger evidence than the constrained task-based setting used here, and would shed light on whether the corrective and awareness-building effects observed within a single session persist in practice. Although the self-contained nature of the snippets evaluated in this thesis was a deliberate choice, such a study would also reveal the practical boundaries of the tool's applicability, since real-world code is often stateful and framework-dependent.

Ultimately, this work suggests that fine-grained, in-IDE energy measurement is both technically achievable and empirically beneficial. As software energy consumption continues to grow as a sustainability concern [70, 27], equipping developers with tools that bring energy feedback directly into the edit-compile-run cycle [18] is a concrete step toward making energy-aware development a routine part of software engineering practice [44, 56].

Bibliography

[1] Richard E Boyatzis. *Transforming qualitative information: Thematic analysis and code development*. Sage, 1998.

[2] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[3] Virginia Braun and Victoria Clarke. Successful qualitative research: A practical guide for beginners. 2013.

[4] Auguste Bravais. *Analyse mathématique sur les probabilités des erreurs de situation d'un point*. Impr. Royale, 1844.

[5] Nancy Carter. The use of triangulation in qualitative research. *Number 5/September 2014*, 41(5):545–547, 2014.

[6] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[7] Robert Coe. It's the effect size, stupid. In *British educational research association annual conference*, volume 12, page 14, 2002.

[8] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 2nd edition, 1988.

[9] Benjamin F. Crabtree and William L. Miller. *Doing Qualitative Research*. SAGE Publications, 2nd edition, 1999.

[10] Luís Cruz. Green software engineering done right: a scientific guide to set up energy efficiency experiments. `http://luiscruz.github.io/2021/10/10/scientific-guide.html`, 2021. Blog post.

[11] Luis Cruz and Rui Abreu. On the energy footprint of mobile testing frameworks. *IEEE Transactions on Software Engineering*, 47(10):2260–2271, 2019.

[12] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 189–194, 2010.

[13] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1449–1461. IEEE, 2025.

[14] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9(5):256–268, 2003.

[15] R Dudley. The shapiro–wilk test for normality, 2015.

[16] Michael P Fay and Michael A Proschan. Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys*, 4:1, 2010.

[17] GH Fischer. On a distribution yielding the error function of several well-known statistics. *Proc. inter. cong. math.*, 2:805–813, 1924.

[18] Alcides Fonseca, Rick Kazman, and Patricia Lago. A manifesto for energy-aware software. *IEEE software*, 36(6):79–82, 2019.

[19] Gene V Glass. Primary, secondary, and meta-analysis of research. *Educational researcher*, 5(10):3–8, 1976.

[20] James E Grizzle. The two-period change-over design and its use in clinical trials. *Biometrics*, pages 467–480, 1965.

[21] Greg Guest, Kathleen M MacQueen, and Emily E Namey. *Applied thematic analysis*. sage publications, 2011.

[22] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. Energat: Fine-grained energy attribution for multi-tenancy. *ACM SIGENERGY Energy Informatics Review*, 4 (3):18–25, 2024.

[23] Abram Hindle. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2):374–409, 2015.

[24] Mathilde Jay, Vladimir Ostapenco, Laurent Lefèvre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. An experimental comparison of software-based power meters: focus on cpu and gpu. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 106–118. IEEE, 2023.

[25] Beth Karlin, Rebecca Ford, and Cassandra Squiers. Energy feedback technology: a review and taxonomy of products and platforms. *Energy Efficiency*, 7(3):377–399, 2014.

[26] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3 (2):1–26, 2018.

[27] Mohit Kumar, Youhuizi Li, and Weisong Shi. Energy consumption in java: An early experience. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8. IEEE, 2017.

[28] Henry A Landsberger. Hawthorne revisited: management and the worker, its critics, and developments in human relations in industry. 1958.

[29] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[30] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th working conference on mining software repositories*, pages 2–11, 2014.

[31] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371. IEEE, 2021.

[32] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *International Conference on Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2015.

[33] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[34] Patricia Yancey Martin and Barry A Turner. Grounded theory and organizational research. *The journal of applied behavioral science*, 22(2):141–157, 1986.

[35] Elena Mihalache, Xutong Liu, and Andy Zaidman. Replication package for energy-aware java development, June 2026. URL `https://doi.org/10.5281/zenodo.20766102`.

[36] Douglas C Montgomery and George C Runger. *Applied statistics and probability for engineers*. John wiley & sons, 2018.

[37] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[38] Geoff Norman. Likert scales, levels of measurement and the "laws" of statistics. *Advances in health sciences education*, 15(5):625–632, 2010.

[39] Adel Noureddine. Powerjoular and joularjx: Multi-platform software power monitoring tools. In *2022 18th International Conference on Intelligent Environments (IE)*, pages 1–4. IEEE, 2022.

[40] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 160–169, 2012.

[41] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software: Energy profiling of software code. *Automated Software Engineering*, 22 (3):291–332, 2015.

[42] Adel Noureddine, Martín Diéguez Lodeiro, Noëlle Bru, and Richard Chbeir. The impact of green feedback on users' software usage. *IEEE Transactions on Sustainable Computing*, 8(2):280–292, 2022.

[43] Cliodhna O'Connor and Helene Joffe. Intercoder reliability in qualitative research: Debates and practical guidelines. *International journal of qualitative methods*, 19: 1609406919899220, 2020.

[44] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2015.

[45] Michael Quinn Patton. Enhancing the quality and credibility of qualitative analysis. *Health services research*, 34(5 Pt 2):1189, 1999.

[46] Karl Pearson. Notes on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

[47] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, pages 256–267, 2017.

[48] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, 2001.

[49] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 251–259, 2001.

[50] Guillaume Raffin and Denis Trystram. Dissecting the software-based measurement of cpu energy consumption: a comparative analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[51] Suzanne Rivoire, Mehul A Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376, 2007.

[52] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*, volume 14. Citeseer, 2006.

[53] Johnny Saldaña. The coding manual for qualitative researchers. 2021.

[54] June Sallou, Luís Cruz, and Thomas Durieux. Energibridge: empowering software sustainability through cross-platform energy measurement. *arXiv preprint arXiv:2312.13897*, 2023.

[55] Nicolae Scarlat, Matteo Prussi, and Monica Padella. Quantification of the carbon intensity of electricity produced and used in europe. *Applied Energy*, 305:117901, 2022.

[56] Simon Schubert, Dejan Kostic, Willy Zwaenepoel, and Kang G Shin. Profiling software for energy consumption. In *2012 IEEE International Conference on Green Computing and Communications*, pages 515–522. IEEE, 2012.

[57] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[58] George Waddel Snedecor. Calculation and interpretation of analysis of variance and co-variance. 1934.

[59] Charles Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[60] Diomidis Spinellis. The social responsibility of software development. *IEEE Software*, 34(2):4–6, 2017.

[61] Stephen M Stigler. Francis galton's account of the invention of correlation. *Statistical Science*, pages 73–79, 1989.

[62] Margaret-Anne Storey, Rashina Hoda, Alessandra Maciel Paz Milani, and Maria Teresa Baldassarre. Guiding principles for mixed methods research in software engineering. *Empirical Software Engineering*, 30(5):138, 2025.

[63] Anselm Strauss, Juliet Corbin, et al. *Basics of qualitative research*, volume 15. sage Newbury Park, CA, 1990.

[64] Anselm L Strauss. *Qualitative analysis for social scientists*. Cambridge university press, 1987.

[65] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[66] Gail M Sullivan and Anthony R Artino Jr. Analyzing and interpreting data from likert-type scales. *Journal of graduate medical education*, 5(4):541–542, 2013.

[67] L Thomas Van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs Van Der Storm, Benoit Combemale, and Olivier Barais. A principled approach to repl interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 84–100, 2020.

[68] Sira Vegas, Cecilia Apa, and Natalia Juristo. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2): 120–135, 2015.

[69] Roberto Verdecchia, Fabio Ricchiuti, Albert Hankel, Patricia Lago, and Giuseppe Procaccianti. Green ict research and challenges. In *Advances and New Trends in Environmental Informatics*, pages 37–48. Springer, 2017.

[70] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol De Vries. Green it and green software. *IEEE software*, 38(6):7–15, 2021.

[71] Ronald E Walpole, Raymond H Myers, Sharon L Myers, and Keying Ye. *Probability and Statistics for Engineers and Scientists*. Pearson, 9th edition, 2011.

[72] Bernard L Welch. The generalization of 'student's'problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[73] Amber Wutich, Melissa Beresford, and H Russell Bernard. Sample sizes for 10 types of qualitative data analysis: an integrative review, empirical guidance, and next steps. *International Journal of Qualitative Methods*, 23:16094069241296206, 2024.

[74] Andy Zaidman. An inconvenient truth in software engineering? the environmental impact of testing open source java projects. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, pages 214–218, 2024.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AMD APM: AMD Application Power Management

CMOS: Complementary Metal-Oxide-Semiconductor

CPU: Central Processing Unit

DRAM: Dynamic Random Access Memory

GUI: Graphical User Interface

ICT: Information and Communications Technology

(Intel) RAPL: (Intel) Running Average Power Limit

IT: Information Technology

µJ**:** (micro)Joule

ms**:** (milli)second

MSR: Model Specific Registers

NVML: Nvidia Management Library

OS: Operating System

REPL: Read Eval Print Loop

SoC: System on Chip

PID: Process Identifier

PDU: Power Distribution Unit

PMC: Performance Counters

TDP: Thermal Design Power

UI: User Interface

VM: Virtual Machine

Appendix B

Histograms with Gaussian Curve Overlay

This appendix contains the detailed distribution plots with normal curve overlay for both energy and time measurements across all 60 implementations used in the verification phase. Each plot includes a normal curve overlay and the corresponding Shapiro-Wilk p-value to visualise the results discussed in Chapter 5. The plots are grouped into six batches of five algorithm pairs each to maintain legibility and allow for direct side-by-side comparisons between implementations.

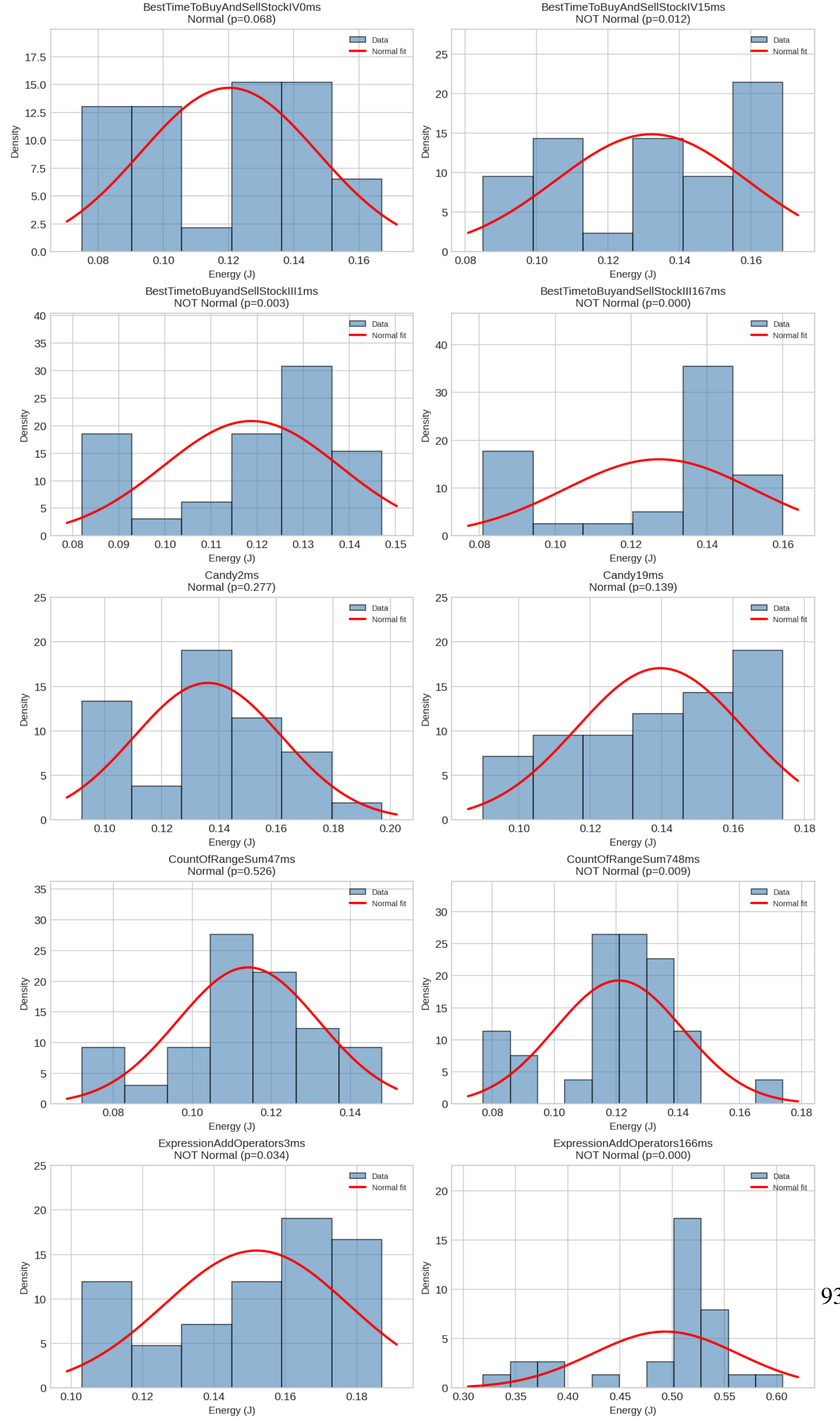


Figure B.1: Energy consumption distributions with normal curve overlay (Part 1).

Energy Histograms With Gaussian Curve Overlay (Part 2)

FirstMissingPositive1ms NOT Normal (p=0.024)

FirstMissingPositive28ms Normal (p=0.285)

IntegerToEnglishWords1ms NOT Normal (p=0.003)

IntegerToEnglishWords9ms NOT Normal (p=0.002)

LongestBalancedSubarrayI9ms NOT Normal (p=0.000)

LongestBalancedSubarrayI445ms NOT Normal (p=0.000)

LongestBalancedSubarrayII145ms NOT Normal (p=0.000)

LongestBalancedSubarrayII456ms NOT Normal (p=0.000)

LongestIncreasingPathInAMatrix6ms NOT Normal (p=0.000)

LongestIncreasingPathInAMatrix90ms NOT Normal (p=0.000)

Figure B.2: Energy consumption distributions with normal curve overlay (Part 2).

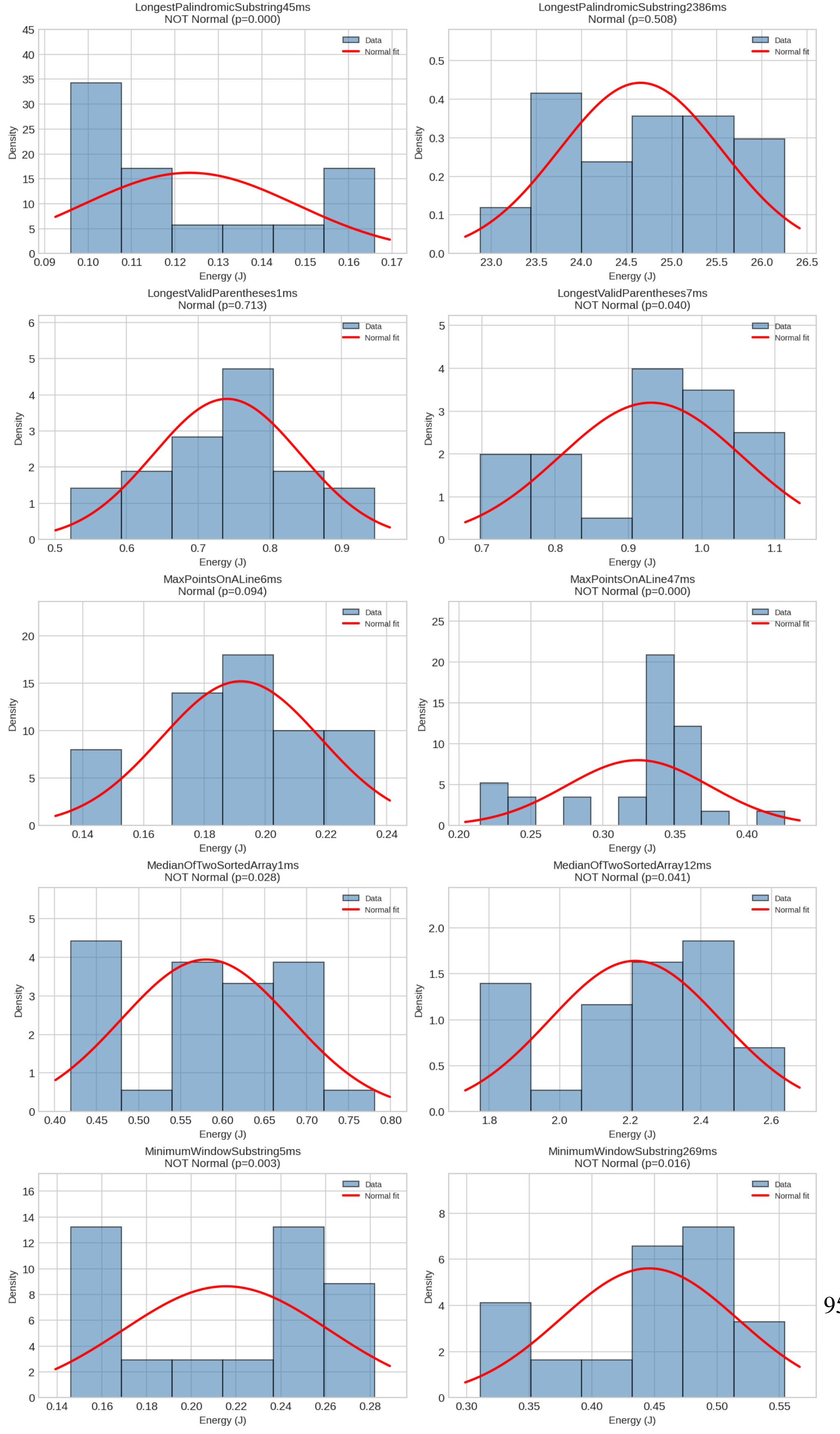


Figure B.3: Energy consumption distributions with normal curve overlays (Part 3).

Energy Histograms With Gaussian Curve Overlay (Part 4)

PalindromePairs104ms
NOT Normal (p=0.002)

PalindromePairs1431ms
NOT Normal (p=0.000)

PerfectRectangle5ms
NOT Normal (p=0.006)

PerfectRectangle79ms
NOT Normal (p=0.001)

PermutationSequence8ms
NOT Normal (p=0.000)

PermutationSequence1279ms
Normal (p=0.734)

RegularExpressionMatching6ms
Normal (p=0.235)

RegularExpressionMatching1036ms
Normal (p=0.808)

RemoveInvalidParentheses1ms
NOT Normal (p=0.001)

RemoveInvalidParentheses423ms
NOT Normal (p=0.002)

Figure B.4: Energy consumption distributions with normal curve overlay (Part 4).

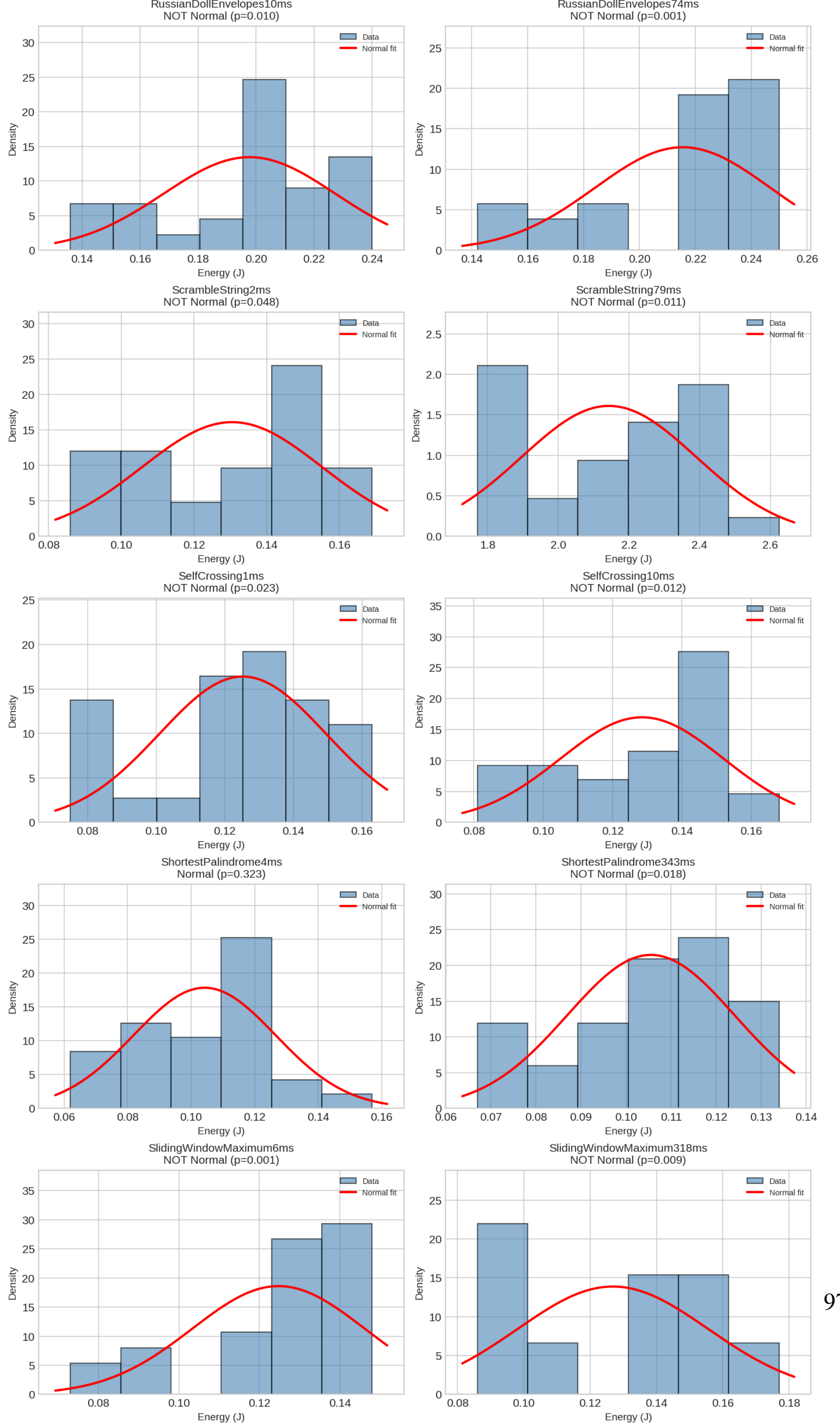


Figure B.5: Energy consumption distributions with normal curve overlay (Part 5).

Energy Histograms With Gaussian Curve Overlay (Part 6)

Figure B.6: Energy consumption distributions with normal curve overlay (Part 6).

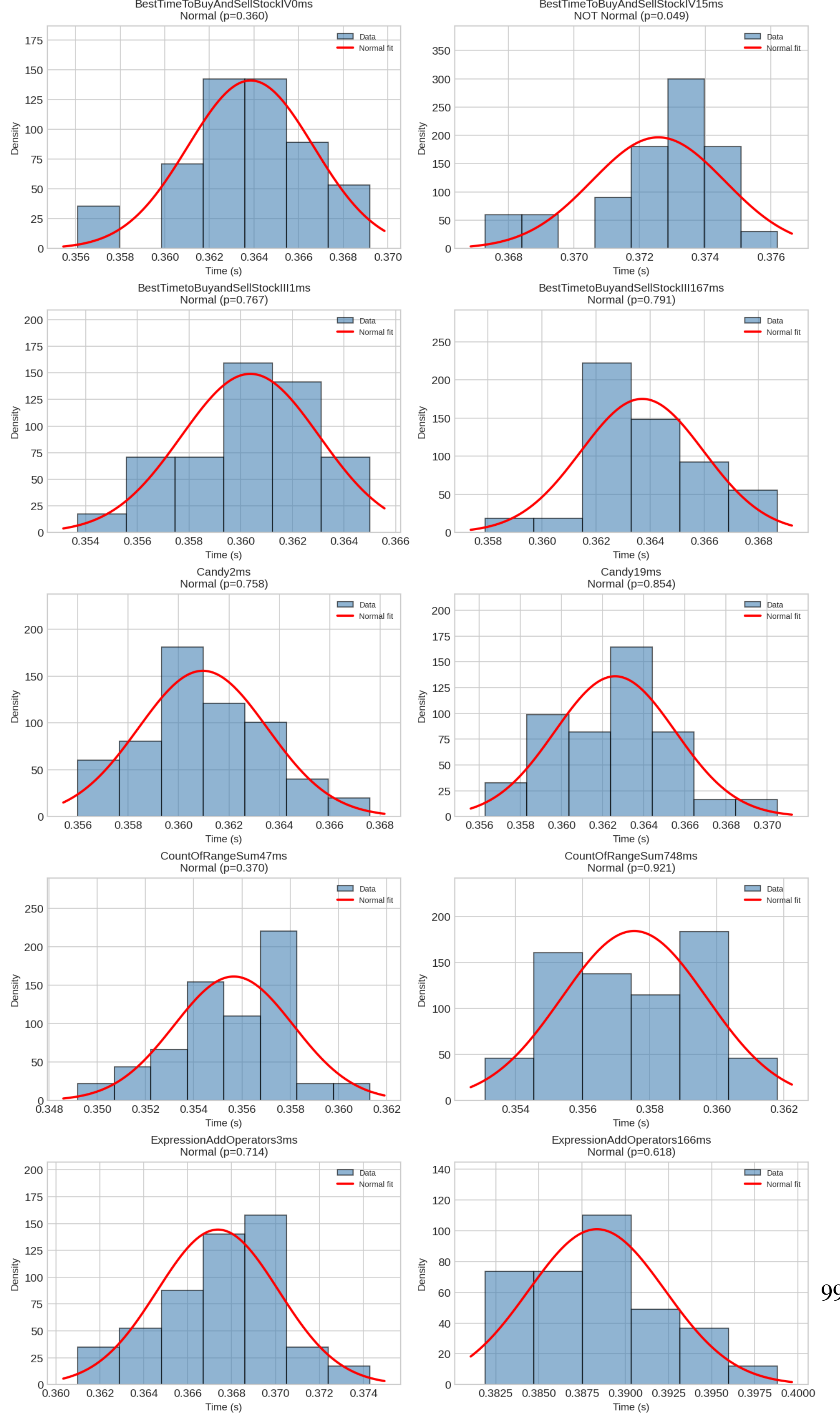


Figure B.7: Execution time distributions with normal curve overlay (Part 1).

Time Histograms With Gaussian Curve Overlay (Part 2)

FirstMissingPositive1ms NOT Normal (p=0.003)
FirstMissingPositive28ms Normal (p=0.137)
IntegerToEnglishWords1ms NOT Normal (p=0.005)
IntegerToEnglishWords9ms NOT Normal (p=0.000)
LongestBalancedSubarrayI9ms Normal (p=0.833)
LongestBalancedSubarrayI445ms Normal (p=0.254)
LongestBalancedSubarrayII145ms Normal (p=0.241)
LongestBalancedSubarrayII456ms Normal (p=0.054)
LongestIncreasingPathInAMatrix6ms Normal (p=0.413)
LongestIncreasingPathInAMatrix90ms Normal (p=0.080)

Figure B.8: Execution time distributions with normal curve overlay (Part 2).

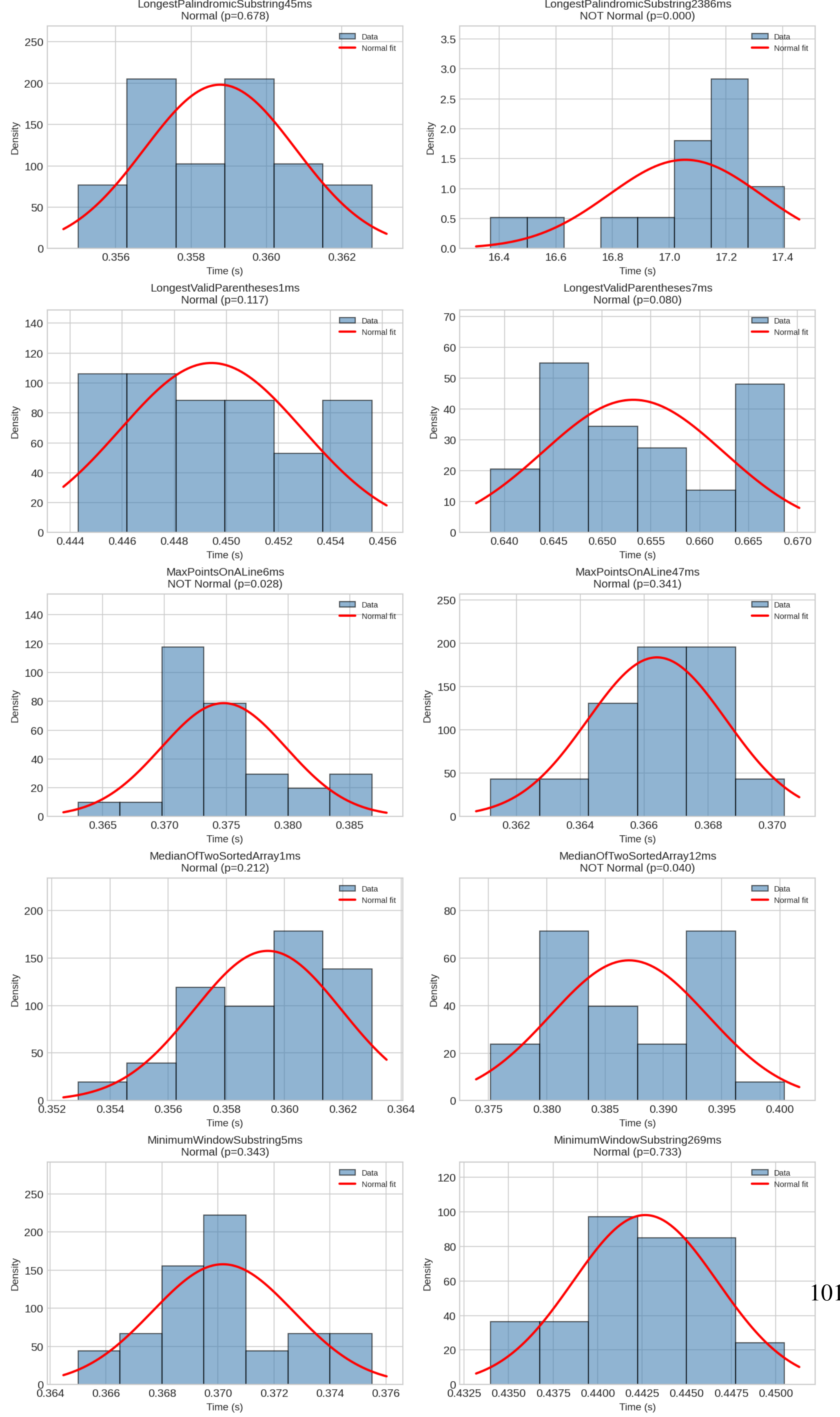


Figure B.9: Execution time distributions with normal curve overlay (Part 3).

Figure B.10: Execution time distributions with normal curve overlay (Part 4).

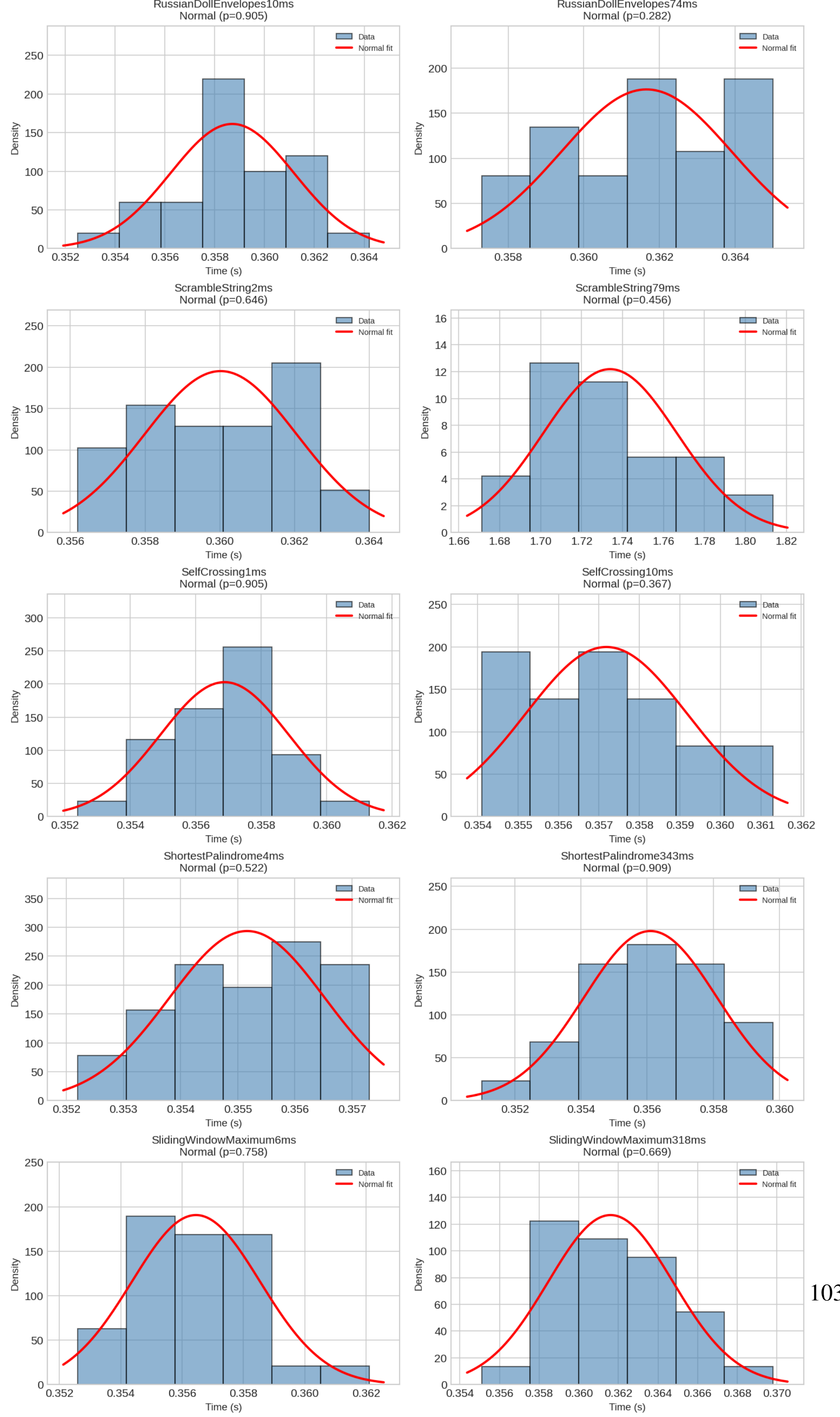


Figure B.11: Execution time distributions with normal curve overlay (Part 5).

Figure B.12: Execution time distributions with normal curve overlay (Part 6).

Appendix C

Kernel Density Estimation (KDE) Plots

This appendix contains Kernel Density Estimation (KDE) plots comparing the energy and time distributions of the `fast` and `slow` implementations for each of the 30 algorithm pairs. These plots visually represent the effect size magnitude (Cliff's delta) discussed in Chapter 5. The plots are grouped into three batches of 10 algorithm pairs each to maintain legibility.

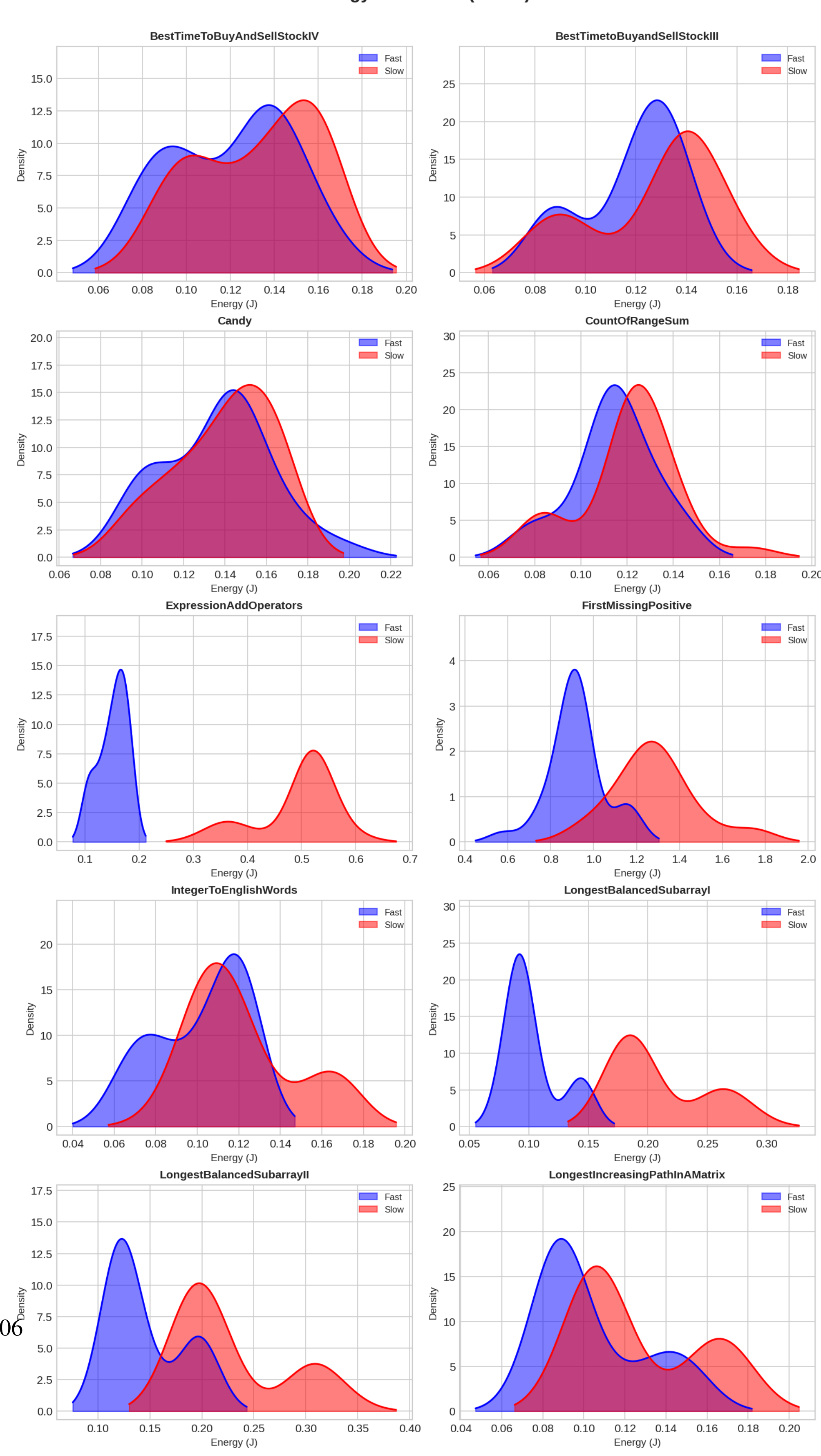


Figure C.1: Energy consumption KDE plots comparing `fast` and `slow` implementations (Part 1).

Energy KDE Plots (Part 2)

Figure C.2: Energy consumption KDE plots comparing `fast` and `slow` implementations (Part 2).

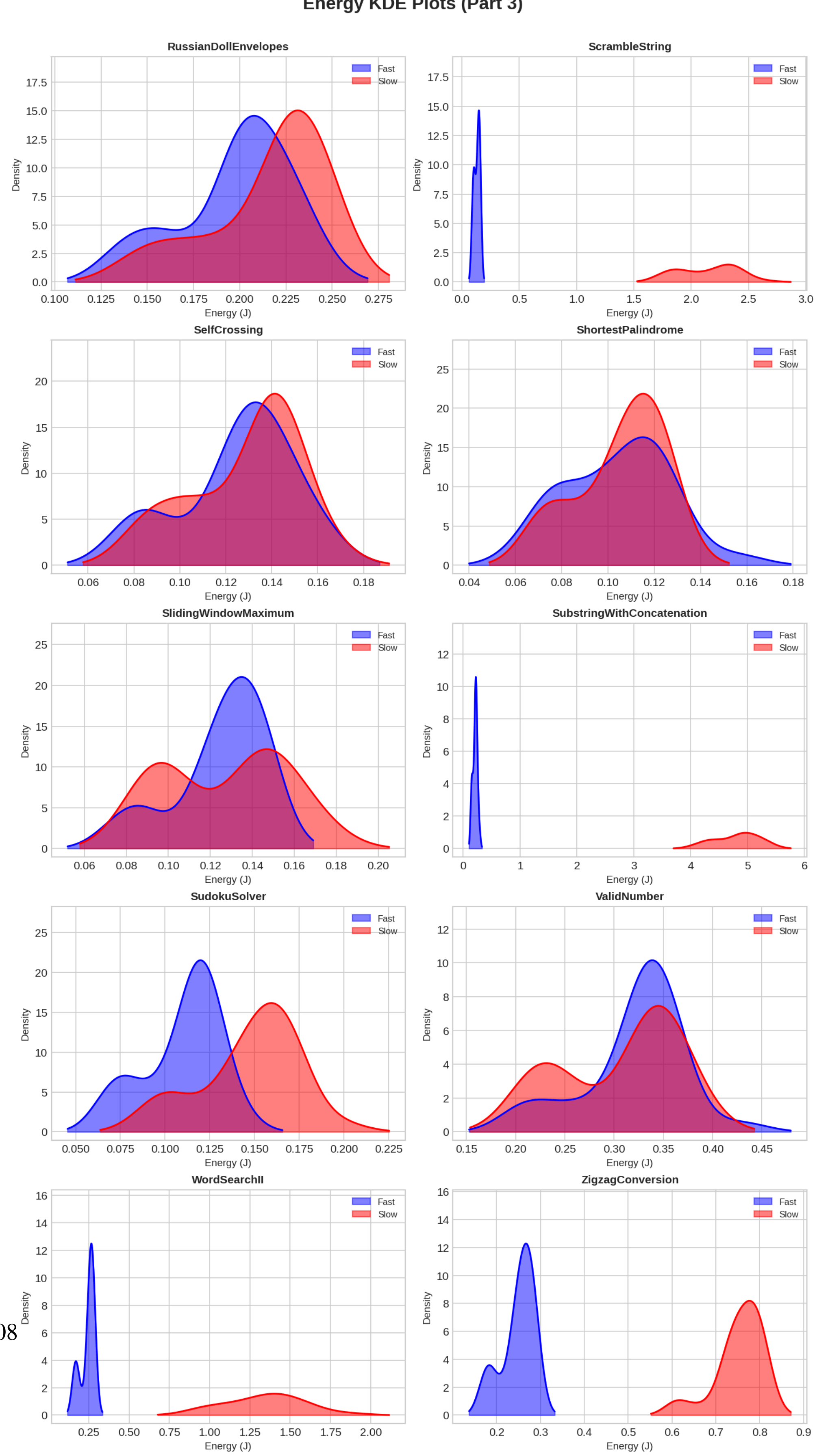


Figure C.3: Energy consumption KDE plots comparing `fast` and `slow` implementations (Part 3).

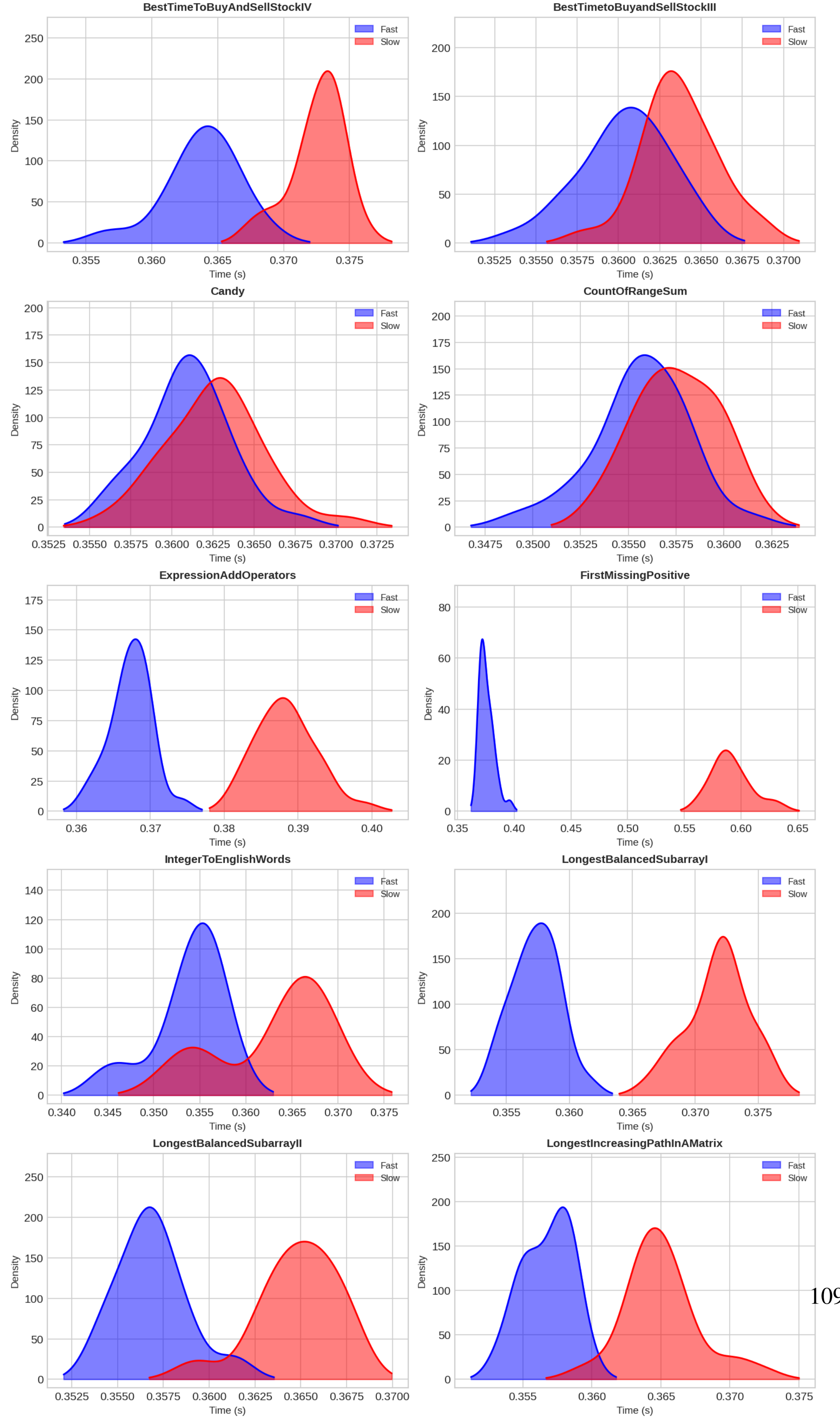


Figure C.4: Execution time KDE plots comparing `fast` and `slow` implementations (Part 1).

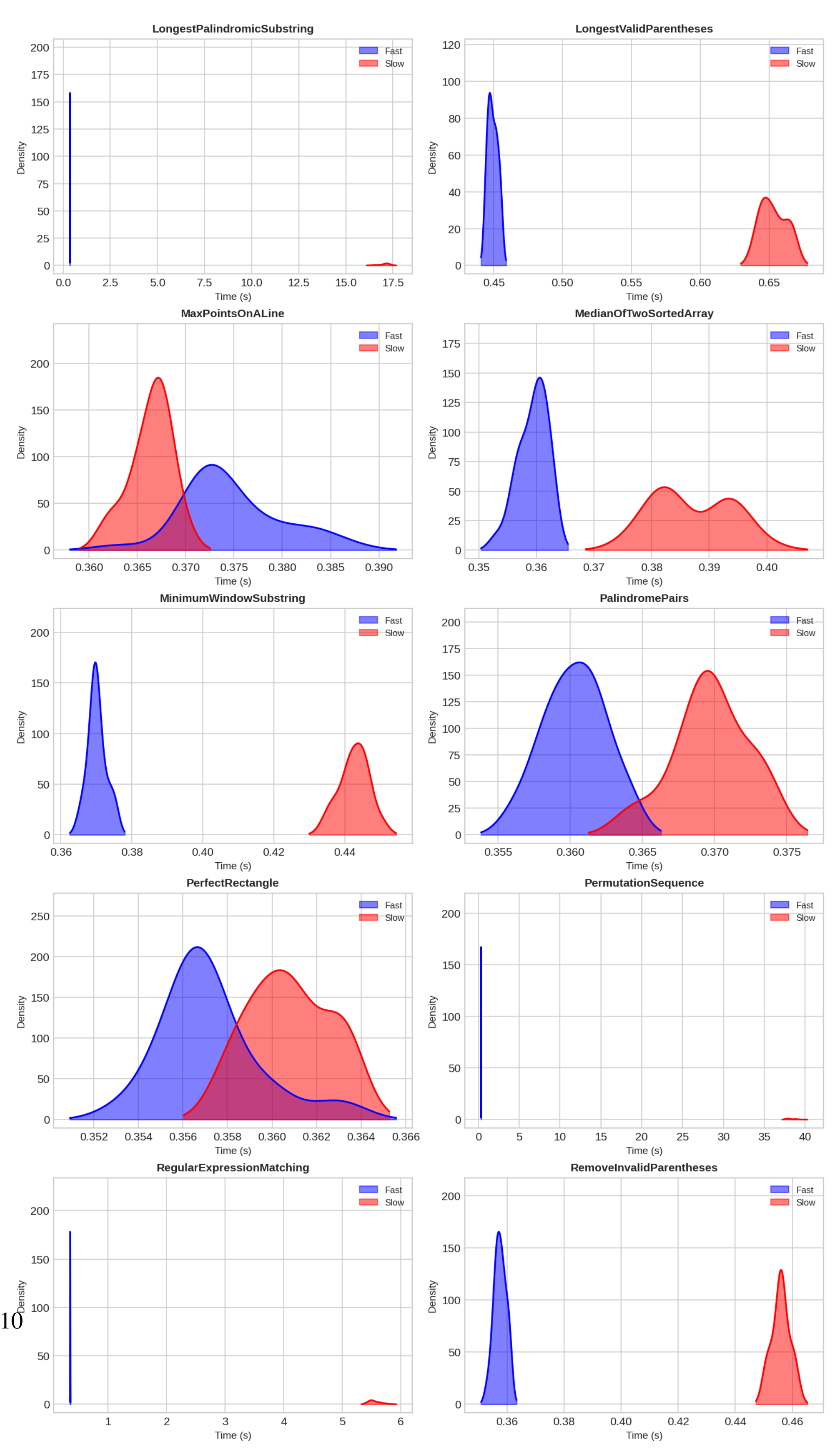


Figure C.5: Execution time KDE plots comparing `fast` and `slow` implementations (Part 2).

Figure C.6: Execution time KDE plots comparing `fast` and `slow` implementations (Part 3).

Appendix D

Microsoft Forms User Study Survey

This appendix contains the Microsoft Forms survey used in the user study.

Energy Conscious Java Development User Study

* Required

Purpose of study

The goal of this research is to evaluate an energy-analysis plugin for IntelliJ and understand if it helps developers reason about the energy efficiency of code.
The entire session will take approximately 40 minutes.
During this time, you will be asked to:

1. Complete a background questionnaire.
2. Go through a short tutorial on how to use the energy-analysis plugin.
3. Complete two code analysis tasks. In each task, you will look at two different implementations of a problem and determine which is more energy efficient. You will complete one task using the plugin and the other without it.
4. Fill out a short post-task questionnaire regarding your experience.

While you are working on the code analysis tasks, we ask that you **think aloud**. This means you should continuously speak your thoughts as you navigate the code and make decisions. Please tell us:

- What you are currently looking at.
- What you are trying to figure out.
- The reasoning behind your conclusions.

To accurately capture your feedback and reasoning, this session will include audio and screen recording.

1. Name *

2. Have you read and signed the consent form? *

○ Yes

○ No

3. What is your programming experience? *

- less than 1 year
- 1-3 years
- 3-5 years
- 5+ years

4. What is your Java experience? *

- less than 1 year
- 1-3 years
- 3-5 years
- 5+ years

5. What kind of Java experience do you have? *

- none
- have used Java in my education
- have used Java only outside of my education
- have used Java in and outside of my education
- Java is one the my main programming languages of choice

6. What is your IntelliJ experience? *

- never used IntelliJ
- use IntelliJ sometimes
- IntelliJ is my IDE of choice for Java development

7. Have you taken courses or done research related to energy and/or performance optimisation? *

- ◯ Yes
- ◯ No

8. How frequently have you performed the following types of optimisation? *

	Never	Once or twice a year	Monthly	W
Energy optimisation (e.g., power draw)	◯	◯	◯	
Performance optimisation (e.g., runtime)	◯	◯	◯	

9. What is your occupation? *

- ☐ CS/DS/AI Student
- ☐ Other Student
- ☐ Software Engineer/Developer
- ☐ Frontend Engineer/Developer
- ☐ Backend Engineer/Developer
- ☐ Full-stack Engineer/Developer
- ☐ Systems Engineer/Developer
- ☐ Data/AI Engineer/Developer
- ☐ QA/Testing
- ☐ Cybersecurity
- ☐ Other

Tutorial

Do you have any questions? Is everything clear?

Task 1

Description: build a result string by concatenating 100,000 words.

10. Which solution do you think is more energy efficient?

- ◯ A
- ◯ B

11. How confident are you in your choice?

- ◯ Extremely confident
- ◯ Somewhat confident
- ◯ Neutral
- ◯ Somewhat not confident
- ◯ Extremely not confident

12. Explain your reasoning briefly.

Task 2

Description: check whether a list of 100,000 integers contains any duplicates

13. Which solution do you think is more energy efficient?

- ◯ A
- ◯ B

14. How confident are you in your choice?

- ◯ Extremely confident
- ◯ Somewhat confident
- ◯ Neutral
- ◯ Somewhat not confident
- ◯ Extremely not confident

15. Explain your reasoning briefly.

Task 3

Description: calculate the Fibonacci number at index 45.
Note: a Fibonacci number is part of a specific integer sequence where each number is the sum of the two preceding ones, starting from 0 and 1.

16. Which solution do you think is more energy efficient? *

- ◯ A
- ◯ B

17. How confident are you in your choice? *

- ◯ Extremely confident
- ◯ Somewhat confident
- ◯ Neutral
- ◯ Somewhat not confident
- ◯ Extremely not confident

18. Explain your reasoning briefly. *

Task 4

Description: multiply two 1500×1500 matrices.

19. Which solution do you think is more energy efficient? *

- ◯ A
- ◯ B

20. How confident are you in your choice? *

- ◯ Extremely confident
- ◯ Somewhat confident
- ◯ Neutral
- ◯ Somewhat not confident
- ◯ Extremely not confident

21. Explain your reasoning briefly. *

Post-task questionnaire

22. How much do you agree with the following statements?

	Strongly disagree	Disagree	Neutral	A
The tool helped me understand energy consumption.	◯	◯	◯	
The output was easy to interpret.	◯	◯	◯	
The tool would be useful in real development.	◯	◯	◯	
I would use this tool in practice.	◯	◯	◯	

23. Did you find anything confusing about the tool? If so, what? *

24. Did you find the tool to be useful? If so, when or how? *

25. What would you change or improve about the tool? Is there any feature or functionality you are missing? *

26. Anything else you would like to share with us?

Appendix E

Participant Consent Form

This appendix contains the consent form provided to all participants prior to their involvement in the study.

Invitation and purpose: You are being invited to participate in a research study conducted by a Master's student from TU Delft. The purpose of this research is to evaluate a new IntelliJ IDEA plugin designed to provide immediate energy feedback for Java code snippets. The study aims to investigate whether this tool can increase energy awareness among developers and how well it integrates into the developer workflow.

Procedure: If you agree to participate, the session will take approximately 45-60 minutes. During this session, you will be asked to:

- Use the provided IntelliJ plugin to solve or evaluate specific Java programming tasks (e.g., LeetCode problems).
- Think aloud while performing the tasks, allowing the researcher to observe your workflow and decision-making process.
- Answer questions regarding your experience, the usability of the tool, and the clarity of the energy feedback.

Data collection and recording: To accurately analyse user interaction and feedback, this study requires:

- Audio recording: to capture your verbal feedback and thinking aloud comments.
- Screen recording: to capture your interaction with the IntelliJ IDE and the plugin interface.
- Observation notes: taken by the researcher during the session.

Confidentiality and data storage: Your data will be stored securely and accessible only to the research team. Any research data will be de-identified before being used in any thesis reports or publications. Data will be retained for 10 years in accordance with TU Delft data management policies.

Risks and mitigation: The risks associated with this study are minimal. The primary risk involves the processing of personal data (your voice and screen activity). To mitigate this no sensitive personal data (e.g., health, political views) will be collected and all data will be pseudonymised (your name will be replaced with a code) during analysis.

Voluntary participation and withdrawal: Participation in this study is voluntary. You have the right to withdraw at any time without penalty or loss of benefits to which you are otherwise entitled.

Consent: By participating in this study, you acknowledge that you have read and understood the information provided. You agree to participate voluntarily and understand that you may withdraw at any time without penalty or loss of benefits to which you are otherwise entitled.

Contact information: If you have any questions or complaints about this study, please contact Elena Mihalache at e.mihalache@student.tudelft.nl.

PLEASE TICK THE APPROPRIATE BOXES	Yes	No
A: GENERAL AGREEMENT – RESEARCH GOALS, PARTICiPANT TASKS AND VOLUNTARY PARTICIPATION		
1. I have read and understood the study information dated [*DD/MM/YYYY*], or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.	☐	☐
2. I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.	☐	☐
3. I understand that taking part in the study involves: [*see points below*] Audio recording of my verbal responses and feedback. Screen recording of my interaction with the software tool. Observation by the researcher while I complete programming tasks.	☐	☐
4. I understand that I will NOT be compensated for my participation.	☐	☐
B: POTENTIAL RISKS OF PARTICIPATING (INCLUDING DATA PROTECTION)		
5. I understand that taking part in the study collecting personally identifiable information (my voice and screen activity) with the potential risk of identification. I understand that these will be mitigated by pseudonymisation and secure storage.	☐	☐
6. I understand that the following steps will be taken to minimise the threat of a data breach, and protect my identity in the event of such a breach: pseudonymisation, secure data storage, limited access to data, deletion after 10 years.	☐	☐
7. I understand that personal information collected about me that can identify me, such as my name, will not be shared beyond the study team.	☐	☐
C: RESEARCH PUBLICATION, DISSEMINATION AND APPLICATION		
8. I understand that after the research study the de-identified information I provide will be used for a Master's thesis and potential academic publications.	☐	☐

9. I agree that my responses, views or other input can be quoted anonymously in research outputs	☐	☐
D: (LONGTERM) DATA STORAGE, ACCESS AND REUSE		
10. I give permission for the de-identified data that I provide to be archived in the TU Delft repository so it can be used for future research and learning.	☐	☐

Signatures

____________________________ ___________________________ _________

Name of participant Signature Date

I, as researcher, have provided the information sheet to the potential participant and, to the best of my ability, ensured that the participant understands to what they are freely consenting.

__________________________ ____________________ _________

Researcher name Signature Date

Study contact details for further information:

Elena Mihalache

+40766806222

e.mihalache@student.tudelft.nl