LLMs to Automatically Refactor Green Code Anti-Patterns

Saif Rashed
saif.rashed@student.uva.nl

Feb 3, 2025, 42 pages

Academic & Daily supervisor: Dr. A.M. (Ana) Oprescu , a.m.oprescu@uva.nl
Host organisation/Research group: CCI, https://cci-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

The World Bank Group estimates that the IT sector is responsible for 1.7% of global CO_2 emissions. The European Union aims to achieve climate neutrality by 2050. But, more fundamentally, the observation known as Moore's Law, which states that computing power doubles every two years, has come to an end. These factors are driving innovation in software energy performance as the demand for compute increases. This work explores automated software refactoring for energy efficiency by using large language models. First, we introduce a modular multi-stage pipeline called Refacturbo that can identify and refactor green anti-patterns. Green anti-patterns are recurring code constructs that lead to excessive energy use. Static analysis can be used to detect certain segments in source code containing these efficiency problems. By creating targeted prompts from these segments, and passing them to a large language model, we can refactor the code segments by replacing the original code with the generated solution. In the experiments on three selected anti-patterns, if-else chains, string concatenation in loops and excessive boxing, the generated optimizations consistently lower the energy use significantly, achieving an average energy cost reduction of 82%, ranging from 64% up to 90%. Although there is not one prompting strategy that dominates over the others, this study demonstrates that there is a feasible path forward in automated refactoring of energy intensive code segments.

Contents

Chapter 1

Introduction

1.1 Context

If we look from a global perspective, we see that the World Bank Group reports in 2022 that smartphone usage accounts for 23 terawatts of electricity use and PCs consume 392 terawatts. This totals to an amount of 415 terawatts [1]. And the report also states that the IT sector is accountable for approximately 1.7% of global CO2 emissions.

If we look from a European perspective, we can see that there is a target set by the European Union to achieve carbon net neutrality by 2050 [2]. And even more fundamental is the observed end of Moore's law [3, 4], an observation that states that compute doubles every two years.

These factors push the industry to create performance efficiency solutions. And in the case of software, this means to produce code that is more energy-efficient.

1.2 Problem statement

Software is becoming a crucial element of modern systems and what makes software unique is the scale. In a paper by Balanza-Martinez *et al.* [5] the authors show an example with Google, stating that if a small change is made at the application level exponential savings can be made if this change is deployed to all the devices. One practical way to make code energy efficient is by refactoring the software code, restructuring it while maintaining its external behavior [6].

Energy reductions can be achieved by targeting specific recurring problems in code. These anti-patterns can be detected with static analysis tools [7]. However, refactoring these patterns into optimized versions is a non-trivial task. It requires the developer to know and understand what impact a certain change has on the overall software system [3, 8]. And above all, it is a delicate process whereby the developer has to apply a change while it maintains the same functional behavior externally, requiring time and effort.

Large Language Models as a technology offers new opportunities in quickly refactoring code and have shown the ability to do so [9–11]. However, with its benefits for quick refactoring and compatibility with diverse programming languages, prior studies have shown that it is inconsistent in providing energy efficient code after being given a simple instructive prompt [12, 13]. However, studies also show that if you provide this model with a more descriptive description of the task it should solve, it is in fact able to improve its performance [9, 14, 15]. This shows us at least a gap whereby existing static analysis efforts can be combined with these models to make more targeted prompts in order to improve the effectiveness of refactoring in terms of software efficiency.

1.3 Research questions

The main goal of this research is to automate energy-related refactoring by using large language models. In order to solve this problem and fill in the research gap stated earlier, two research questions have been created.:

- **RQ1:** How can a system be designed and implemented to use LLMs for automatically refactoring code anti-patterns?
- **RQ2:** How do different prompting strategies affect the energy consumption of code refactored from unoptimized versions?

The first research question is focused on the problem of automating refactoring and how a system can be created that is able to do this. The second question is about the evaluation of the LLMs ability to create energy efficient code. To see what prompts are able to reliably create energy efficient code.

1.4 Research method

For **RQ1**, a method called action research will be used to investigate an issue while it is being solved. Here, a system will be developed and validated using a set of automated tests. For **RQ2**, a quantitative experimental design will be used. In this method, we use hypotheses and variables to evaluate LLM generated code on energy use. By doing these both separately, this study is able to give a solution for automated refactoring while providing empirical quantitative results for different prompting strategies in energy consumption.

1.5 Contributions

This study makes the following contributions:

1. Automated LLM-driven refactoring system
    - Source code
    - Test suite
    - Established anti-pattern rules
    - Prompts for refactoring
2. Energy evaluation of prompt strategies
    - Energy measurement instrumentation
    - Data on energy consumption for different prompt strategies
    - Replication package

1.6 Scope

The scope of this research encompasses the following topics: static analysis, LLMs and prompt engineering and energy-related code patterns. Any content from these topics that are included in this dissertation have some purpose in solving the main objective of this research, stated in section 1.3.

1.7 Outline

This report is organized in several chapters. Chapter 2 will go into background information for this study. Chapter 3 will explain and show the design and implementation of the automated refactoring system called Refacturbo. Chapter 4 will go into the evaluation methodology used to measure energy consumption for different prompting strategies. Chapter 5 will show and present the results from the energy experiments. The Chapter 6 will provide a discussion on these results and answer both research questions. And Chapter 8 will conclude the research and provide future directions. Any material used can be found in the appendices. The collected results for the experiments can be found in the replication package[1] and the main system used to do the automated refactoring can be found in a public repository [2].

[1]https://github.com/saifrashed/refacturbo-experiment
[2]https://github.com/saifrashed/refacturbo

Chapter 2

Background

2.1 Energy-efficient software

In this chapter we will briefly look at code efficiency standards and prior studies trying to make code efficient using refactoring as the technique.

2.1.1 Code Efficiency Standards

The International Organization for Standardization has published an ISO specification [1] listing code weaknesses. One of the tables included in the specification is performance efficiency. Performance efficiency is defined by the standard as the ability for a software product to perform its functions in a specified time and be efficient in the use of resources. They state that these resources can include CPU, memory, storage and network devices. This standard originally came from the Consortium for IT Software Quality [2]. It was developed there as an automated source code measurement criteria. It was later accepted by the Object Management Group [3] as a standard and later expanded to also show general IT and embedded software weaknesses. This eventually merged into the Automated Source Quality Measure Standard which was submitted to the International Organization for Standardization.

2.1.2 Performance refactoring

Refactoring for performance means to restructure code in a better and optimized version, without changing the external behavior, in order to improve the software in terms of time but also energy. Multiple studies explore different strategies, they look at different tools and do empirical evaluations with the focus on reducing the energy use of software. In the table below we summarize the identified studies in this area. We show a short summary of the key findings based on the abstract.

Table 2.1: Table of Studies on Code Refactoring for Energy Efficiency

Study	**Focus**	**Key Finding(s)**	**Year**
Le Goaer and Hertout [16]	SonarQube plugin (ecoCode) to detect and remove energy smells in Android projects	Identifies energy-related code smells, improving energy efficiency in Android apps	2022
Sehgal *et al.* [17]	Refactoring approaches for green software development	Proposes refactoring techniques to reduce energy consumption in software	2022
Şanlıalp *et al.* [18]	Energy efficiency of refactoring techniques in C# and Java for mobile devices	Certain refactoring combinations significantly improve energy efficiency	2022

[1]ISO: https://www.iso.org/standard/80623.html
[2]CISQ: https://www.it-cisq.org/standards/code-quality-standards/
[3]OMG: https://www.omg.org/

Table 2.1: Table of Studies on Code Refactoring for Energy Efficiency

Study	**Focus**	**Key Finding(s)**	**Year**
Ournani *et al.* [19]	Impact analysis of refactoring on energy consumption	Refactoring can lead to measurable energy savings in software systems	2022
Hamdi *et al.* [20]	A study of refactoring impact in Android apps	Refactoring improves maintainability but energy impact varies	2021
Sanlialp and Ozturk [21]	Impact of refactoring on energy consumption across OOP languages	Refactoring techniques reduce energy use, with language-specific variations	2020
Connolly Bree and Cinnéide [22]	Comparison of inheritance vs. delegation in terms of energy efficiency	Delegation can be more energy-efficient than inheritance in certain contexts	2020
Morales *et al.* [23]	An energy refactoring approach for mobile apps	Proposed approach reduces energy consumption in mobile applications	2018
Kim *et al.* [24]	Code refactoring techniques for embedded computing environments	Specific refactorings lower energy use in embedded systems	2018
Verdecchia *et al.* [25]	Empirical evaluation of refactoring code smells energy impact	Refactoring two different code smells improves energy efficiency substantially	2018
Sahin *et al.* [8]	Empirical study on how refactorings affect energy usage	Some refactorings increase energy consumption, others reduce it	2014
Pérez-Castillo and Piattini [26]	Impact of refactoring a specific type of class on power consumption	Refactoring God Classes can negatively affect power efficiency	2014

These studies show that refactoring can lead to energy savings, but does not necessarily do so. It depends on the domain or programming language, so the impact differs. However, what it does show is that refactoring has the ability to reduce energy costs. And this shows us at least that it is worthwhile to investigate and find the correct problematic patterns that can lead to the most energy savings.

2.2 Large Language Models

A large language model is an artificial intelligence system with the ability to predict the most probable next word for a provided input [27]. These models use a combination of GPUs and CPUs and can be run on a personal computer or at scale be distributed in a cloud architecture. This technology has been developed in order to help users with natural language tasks, providing users the ability to ask questions, receive answers, write text, translate language or even write code.

2.2.1 Architecture

The transformer is a deep neural network architecture design introduced by researchers at Google [28]. It uses attention mechanisms to understand the relationship between words in a sentence.

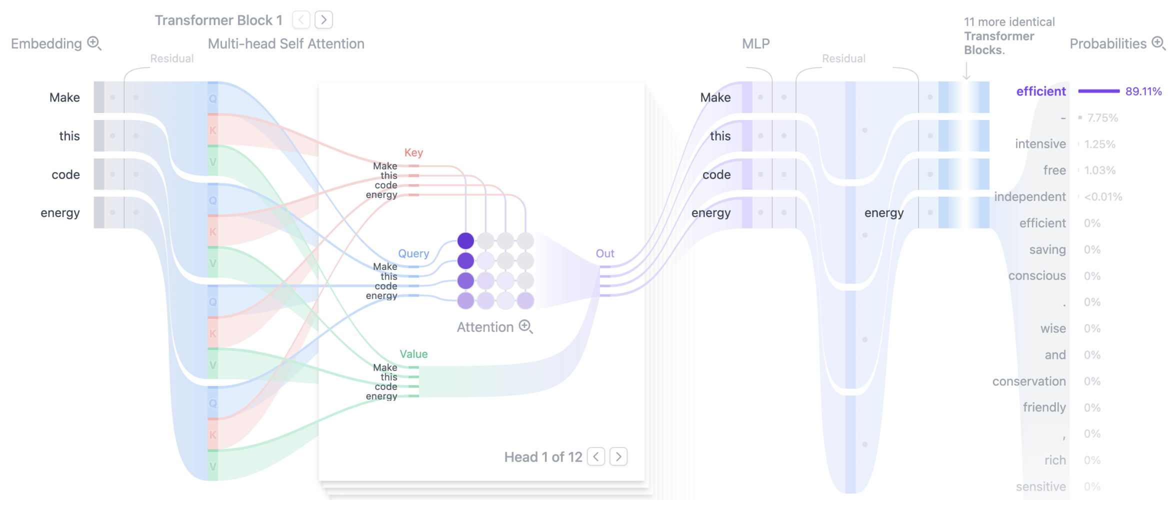

Figure 2.1: A visual explainer of the Transformer architecture [29]

At first, when text is input ingested by the model, it is divided into tokens. The tokens are mapped to embeddings. These embeddings are vectors containing numerical values. These represent the meaning and position of the token. After, the embeddings are pulled through a series of transformer blocks. A transformer block contains a multi-head self-attention. A multi-head self-attention has multiple heads, and these relate tokens with each other. The processed embedding is then passed through a multilayer perceptron that further refines the embedding. After passing through all the transformer blocks, the token embeddings are now influenced by each other and have now encoded the sentence context. Finally, the last contextualized token embedding is used to predict the next most probable word.

2.2.2 Training

Before a transformer becomes a large language model it has to be trained. The training is divided into two phases [27, 30, 31]. The first phase is called **pretraining**, where an algorithm and a dataset are used to train the transformer. The dataset is a corpus of text extracted from the internet. It is fed into the model and the model tries to predict the next word at each step. By doing this it is able to correct the weights according to the sequence of the text. This type of learning is called self-supervision because there is no necessity for a human to label data. The text sequence itself, the structure, serves as the supervision. Anything that happens after pre-training can be considered **post-training**. It has processes such as fine-tuning, where a pre-trained model is further trained on a smaller and more task-specific dataset to adapt it to a particular application. However, this phase is often supervised, meaning that a human has to label the data beforehand.

2.2.3 Energy evaluations

Evaluating a large language model on energy efficiency has become much more important now. This is due to the fact that higher energy expenses make for less usable and scalable models [32]. We can evaluate energy use of the large language model on three different areas. The first is training, second is inference, and third is generated code. For this study we will look at inference and generated code.

Inference

Inference is the process where a large language model predicts a response based on unseen data. This is in part controlled by a temperature parameter. The temperature parameter can be adjusted to control the randomness of the output. The main focus of energy evaluations have been mostly been about model training. However, the operational energy cost during inference also plays an important part. This is difficult to measure because the hardware infrastructure, the software optimizations and API level traffic all play a co-founding role in the energy use of the model [32]. This makes it hard to do a fair comparison between the models. It is even more challenging when we consider that privately held companies keep these internal engineering details secret. However, a study from Samsi *et al.* [33] runs Llama models on their own architecture and demonstrate that different hardware configurations can shift energy use. They show that prompt design and length has some effect on the cost of energy for inference. For example, one study by Rubei *et al.* [34] looked at the impact of several different prompt variations for three different prompt methods. It finds that reorganizing prompts into tags can lower the energy cost.

Generated code

LLMs are increasingly used to generate code, triggering new studies to evaluate efficiency of generated code [12, 13, 35–37]. Note that efficiency is considered broadly; some studies, such as Shypula *et al.* [38], evaluate code efficiency based solely on time rather than energy.

Table 2.2: Table of Studies on LLM Code Efficiency

Study	**Focus**	**Key Finding(s)**	**Year**
Qiu *et al.* [37]	Efficiency evaluator with expert created algorithms and test cases for multiple LLMs	LLMs struggle to produce expert-level efficient code, with eff@k metric showing deficiencies	2025
Cappendijk *et al.* [12]	Prompt modification for energy-efficient code generation	No single prompt consistently reduces energy consumption across problems and LLMs	2024
Peng *et al.* [39]	LLMs as code optimizers for energy efficiency	LLMs improve energy efficiency for multiple programs	2024
Huang *et al.* [36]	A benchmark with 1,000 efficiency-critical coding problems	LLM-generated code is less efficient than human-written solutions, with GPT-4 up to 13.89x slower	2024
Liu *et al.* [35]	Benchmark for efficient code generation	LLMs benefit from instruction tuning for efficiency, but scaling law does not apply	2024
Cursaru *et al.* [13]	Controlled experiment on energy efficiency of Code Llama-generated code	Human-written code is generally more energy-efficient; explicit energy prompts may worsen efficiency	2024
Vartziotis *et al.* [40]	Empirical study on sustainability of LLM-generated code	LLMs show limited sustainability awareness, with varying green capacity across models	2024
Shypula *et al.* [38]	Framework for adapting LLMs to high-level program optimization using performance-improving edits	LLMs achieve a mean speedup of 6.86, close to the human upper limit of 9.56	2023

These studies show that large language models consistently generate code that is less efficient than human written solutions across different benchmarks and metrics, despite advancements in frameworks for evaluating it. However while specific prompt adjustments or model configurations can sometimes improve efficiency for certain specific problems, there is not a single consistent way of prompting the large language model that has shown to reliably produce energy-optimized code.

2.3 In-Context Learning (ICL)

In-Context Learning is a method to train a large language model on new tasks without the requirement to change the internal structure of the model with post-training methods like fine-tuning [41]. It involves the construction of a prompt by having a query concatenated with context. This context contains demonstrations for a specific task in a natural language template. The model then uses this to infer the task and make predictions. A study by Raventós *et al.* [42] suggests that this is an emergent property of large language models. It arises when pre-training task diversity exceeds a certain amount, allowing the model to generalize to unseen tasks. There are several types of prompting methods. First of all, there is zero-shot prompting. It instructs the model to execute a task with simply a descriptive instruction, without examples. One-shot prompting improves on this by including a single example. Few-shot prompting provides multiple examples and can be used by the model to establish a pattern [14]. Last but not least, there is chain-of-thought prompting. This instructs the model to address a task by having it reason through explicit steps [15]. However, despite these benefits, ICL has some limitations [41]. The first is the computational cost increase with more demonstrations, particularly for long-context models. It relies on high-quality examples, which can be challenging to find in situations where these are not available. And adding more demonstrations can have diminishing or negative results, especially if the context becomes overwhelmingly large or if the examples are not well selected.

Chapter 3

Development of an automated LLM refactoring system

In this chapter we introduce Refacturbo [1], a system that was created to automate refactoring of code anti-patterns by using large language models. The conclusion will answer the first research question.

3.1 Introduction

Sustainable software engineering research shows us that there are recurring code patterns that lead to higher energy consumption [16–19, 21, 23, 24, 43]. One way to get rid of these anti-patterns is through the method of refactoring. This is a technique where code is restructured, rewritten into an optimized version without changing its external behavior. Refactoring green anti-patterns starts with detecting them in the code. In a thesis, Sommerhalter [7] shows that these green anti-patterns can be found using a static analysis engine called Semgrep. The system it proposes, named Calabash, has the ability to detect multiple types of anti-patterns in medium to large open source projects, showing that it is able to work at scale.

After detecting these code patterns, large language models can predict optimized versions. These language models have shown promising results in the ability to refactor code [9, 11]. One example, in Liu *et al.* [9], demonstrates that by having a more detailed description of the desired refactoring increases the success rate substantially. In contrast, studies such as those by Cursaru *et al.* [13] and Cappendijk *et al.* [12] show that simple prompts, only containing an instruction with no examples, are unable to consistently reduce the energy cost for code. Likely due to the broad search space for the language model to infer a solution. This tells us that the large language model does not, in fact, have a notion of energy efficiency in terms of code. It is only able to project that what it statistically calculates to be most probable, based on the provided input. This suggests that the notion of energy efficiency is something that needs to be provided in the examples in order to train the model in context.

A challenge with large language models is verifying that the generated changes still behaves functionally the same, and does not introduce new syntax errors or fail in unique cases. It cannot be guaranteed that the results of a large language model is always the same [9, 11]. This inherent variability shows the need for a human-in-the-loop step, in order to mitigate the risks of error. All around, to address these points, we propose Refacturbo, a system that is designed to connect the static analysis efforts with large language model refactoring. By combining these tools, we are able to create a process that is able to detect code anti-patterns, generate prompts based on the detected patterns, and automatically refactor and replace the code with an optimized version.

3.2 Requirements

Refacturbo has 6 functional and 5 non-functional requirements. The requirements are explained with a short rationale.

FR1 The system must perform static code analysis using a predefined set of rules to detect code patterns.

[1]Refacturbo: https://github.com/saifrashed/refacturbo

This requirement ensures the tool automatically identifies code segments given a specification of the pattern structure, such as those patterns documented in previously named studies from Sommerhalter [7] and Feitosa *et al.* [43].

FR2 Static code analysis findings should include metadata including line numbers and file paths.
Including this metadata enables an accurate mapping between the identified anti-patterns and their exact location in the code base. This is important for doing code replacement at **FR6**.

FR3 The system must convert findings into remediation prompts.
Each finding is turned into a prompt that describes how to refactor a specific code snippet. Studies have shown that detailed prompts with relevant code context produce more accurate and targeted refactorings [9, 14, 15], so the prompt file will include a query and corresponding code snippet.

FR4 The system must send prompts to a configured large language model and receive refactored code snippets.
By using large language models, the system can automate modifications that are otherwise labor-intensive [9, 11]. This requirement is the main functionality of Refacturbo, generating improved code versions.

FR5 The system must include a human-in-the-loop step to validate generated refactorings before applying them.
Generated code transformations can introduce unintended changes or errors [9, 11]. Having a human review before finalizing ensures the system remains reliable and preserves functionality.

FR6 The system must replace the detected code segments in the original files with refactored code.
Once the large language model provides changed code, the system must integrate it into the source files using the metadata from **FR2**. Automating this process saves developer time and prevents manual errors [8].

In addition to the functional specifications, Refacturbo will have non-functional requirements that align with the maintainability standards recommended by ISO/IEC 25010, specifically focusing on **modularity**, **reusability**, **analyzability**, **modifiability**, and **testability**. Including these ensures longevity for the system.

NFR1 Modularity
The architecture should be separated into components that are responsible for specific functionalities. This makes development and maintenance easier by keeping changes isolated.

NFR2 Reusability
While the system is specifically built for detecting green anti-patterns, it should have the ability to be used for other tasks requiring detection and transformation of code. This makes the system applicable to a wider variety of problems.

NFR3 Analysability
During every step of the pipeline, output should be stored and saved for facilitating analysis and data collection. Additionally, this helps in tracing and identifying technical errors.

NFR4 Modifiability
Changing the system for new requirements should be flexible and not introduce new bugs. This helps prevent the need for future system overhauls.

NFR5 Testability
The design must enable the testing of individual components and their interactions. This includes unit testing of isolated programs and integration tests for the complete workflow.

3.3 Design

This section will go into the design of Refacturbo, basing it off the requirements listed before. We first have a high-level overview of the system, followed with the architecture.

3.3.1 High-level overview

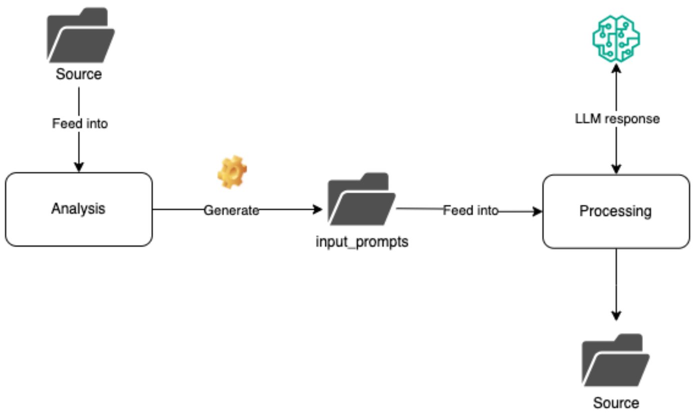

Figure 3.1: High-level overview

Refacturbo is organized as a two-stage modular pipeline (Figure 3.1). In the **Analysis** stage, the system begins by scanning source code with a static analysis tool, to identify potential anti-patterns based on predefined rule sets. In the **Processing** stage, these prompts are fed into a configured LLM. Once the LLM returns the suggested transformations, the snippets are used to replace the original code. This separation of *analysis* and *refactoring* aligns with **NFR1 (Modularity)** by having minimal coupling in between components.

3.3.2 Architecture

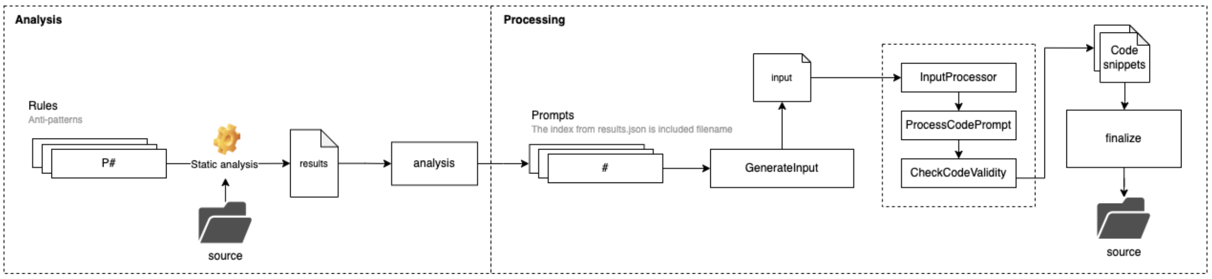

Figure 3.2: Conceptual architecture diagram

The architecture shown at Figure 3.2 is designed to fulfill the non-functional requirements:

1. **Modularity**: Achieved via two-stage design, but also by being divided in separate componentss.
2. **Reusability**: The rule sets are customizable in a way that they can be easily modified or extended for non-green anti-patterns.
3. **Analysability**: Facilitated by storing artifacts in a shared data folder for tracing and inspection. This is not explicitly shown in the diagram.
4. **Modifiability**: By having customizable rules with a prompt field, the system can be changed to accommodate future requirements.
5. **Testability**: Simplified through distinct scripts that allow for unit and integration testing.

As stated in Analysability, this architecture requires a shared data folder to ensure **NF3** is met. This data folder should contain static analysis findings, generated prompts, model input, and the model generated output. This encompasses every type of generated data within the pipeline.

3.4 Implementation

This section covers the implementation of Refacturbo. First, we list the tools used during development and the steps involved in implementing the system's main components.

3.4.1 Tools

The Refacturbo system is built on a variety of inbuilt Python tools and open source projects in order to address the functional requirements. First of all, for doing static analysis, we use Semgrep, a tool that is able to detect code patterns based on a series of specifications defined by the user. In these rules, we are able to add metadata such as a prompt for each pattern. It is also able to produce findings with metadata needed for the subsequent steps, like line numbers, file paths, for each finding, allowing us to fulfill the second requirement. Finally, it supports detecting Java code, which will be the subject language of the experiments. Furthermore, we use Python, specifically via `argparse`, which is able to orchestrate the pipeline's programs through a command line interface. The components are called sequentially, one by one, but they can be called separately from each other. For transforming the code, or refactoring the code, we use the OpenAI Python client to interface with large language models. Testing will be done with the `unittest` framework to validate the system's functional correctness.

3.4.2 Analysis

```
rules:
  - id: avoid-string-concat-in-loop
    message: Avoid string concatenation in loop
    languages: [ java ]
    severity: WARNING
    patterns:
      - pattern-either:
          - pattern: |
              $F(...) {
                ...
                $STR = ...;
                ...
                for (...) {
                  ...
                  $STR += ...;
                  ...
                }
                ...
              }
    metadata:
      prompt: "Avoid string concatenation in loop"
```

Figure 3.3: Example of a Semgrep Rule File

Rule dictionary: Rules are constructed according to the specifications of Semgrep [2]. A Semgrep rule file begins with a top-level key, which defines a list of rule objects that describe the code patterns that should be detected. Each rule object contains fields such as "id" for unique identification, "message" to communicate the rule's purpose, "languages" to specify targeted programming languages, and "severity" to classify the importance of any matched pattern. The "patterns" section governs the actual detection logic, using directives like "pattern-either" to encode disjunctive matching conditions and "pattern-not" to exclude certain contexts. Metavariables, such as "$STR" and "$F(...)," enable matching of variable names and function bodies. The "metadata" key provides the user with the ability to fill in a prompt that will be used at the LLM refactoring step.

[2]Semgrep rules: https://semgrep.dev/docs/writing-rules/rule-syntax

```
{
    ...
    "results": [
        {
            "check_id": "src.....rules.avoid-string-concat-in-loop",
            "path": "...",
            "start": {
                "line": 15,
                "col": 5,
                "offset": 438
            },
            "end": {
                "line": 25,
                "col": 6,
                "offset": 783
            },
            "extra": {
                "message": "Avoid string concatenation in loop",
                "metadata": {
                    "prompt": "Avoid string concatenation in loop"
                },
                "severity": "WARNING",
                "fingerprint": "requires login",
                "lines": "requires login",
                "validation_state": "NO_VALIDATOR",
                "engine_kind": "OSS"
            }
        }
    ],
}
```

Figure 3.4: Findings generated by Semgrep

Static analysis: Semgrep is invoked to perform a static analysis of the provided project directory, relying on the set of scanning rules. This process depends on a properly configured environment where Semgrep is installed, as well as the presence of the rules directory and the appropriate output path for collecting the analysis results. If analysis is finished, a JSON is produced with all the findings.

```
// 0.txt (filename contains index to track)

Avoid string concatenation in loop

    public static String generateFibonacciString(int numRepetitions) {
        String result = "";
        int a = 0, b = 1;
        for (int x = 0; x < numRepetitions; x++) {
            result += a + (x < numRepetitions - 1 ? ", " : "");
            int next = a + b;
            a = b;
            b = next;
        }
        return result;
    }
```

Figure 3.5: Generated prompt

Prompt generation: The analysis.py script parses the JSON output produced by Semgrep and transforms each finding into a standalone textual prompt. The script loads the results data, identifies the prompt text from within the metadata field of each result, and creates a file containing that content in a dedicated output directory. It then correlates the specified file paths and line ranges to the relevant source code and appends those snippets to the prompt. This procedure yields self-contained prompt files that pair the prompt with extracted segments of code. The finding is linked to the prompt by using the index as filename for the prompt text file.

3.4.3 Processing

```
{
    "prompts": [
        "Avoid␣string␣concatenation␣in␣loop\n\n␣␣␣␣public␣static␣String␣
            generateFibonacciString(int␣numRepetitions)␣{\n␣␣␣␣␣␣␣␣String␣result␣=␣
            \\\"\\\";\n␣␣␣␣␣␣␣␣int␣a␣=␣0,␣b␣=␣1;\n␣␣␣␣␣␣␣␣for␣(int␣x␣=␣0;␣x␣<␣
            numRepetitions;␣x++)␣{\n␣␣␣␣␣␣␣␣␣␣␣␣result␣+=␣a␣+␣(x␣<␣numRepetitions␣-␣1␣?␣
            \\\",␣\\\"␣:␣\\\"\\\");\n␣␣␣␣␣␣␣␣␣␣␣␣int␣next␣=␣a␣+␣b;\n␣␣␣␣␣␣␣␣␣␣␣␣a␣=␣b;\n
            ␣␣␣␣␣␣␣␣␣␣␣␣b␣=␣next;\n␣␣␣␣␣␣␣␣}\n␣␣␣␣␣␣␣␣return␣result;\n␣␣␣␣}\n"
    ]
}
```

Figure 3.6: Input used in the processing phase

Prompt parsing: This step orchestrates the generate_input_json.py script to add all textual prompts into a consolidated JSON file, input.json. The script gathers each prompt file from the prompts directory, replaces any embedded quotes to preserve JSON validity, and constructs an in-memory dictionary keyed by "prompts." Afterward, it writes this structure to an export directory, yielding a single authoritative file containing every prompt for use in subsequent steps.

```java
public static String generateFibonacciString(int numRepetitions) {
    StringBuilder result = new StringBuilder();
    int a = 0, b = 1;
    for (int x = 0; x < numRepetitions; x++) {
        result.append(a).append(x < numRepetitions - 1 ? ",␣" : "");
        int next = a + b;
        a = b;
        b = next;
    }
    return result.toString();
}
```

Figure 3.7: Final output

Generating output: The processing.py script coordinates LLM code generation and subsequent result analysis, beginning with the InputProcessor's loading of the input JSON. The prompts then pass to the ProcessCodePrompt class, which delegates them to a LLM class. The class calls the server and generates code that is temporarily stored, after which a human review step is initiated.

```
PROCEDURE ProcessJsonFile(analysisOutputFile, projectDir, modelOutputDir)
    TRY
        Open analysisOutputFile and read JSON data
        FOR each result in data['results']
            Get index, path, startLine, endLine from result
            // Fetch original code
            Construct originalPath using projectDir and path
            Read originalCode from originalPath
            // Fetch refactored code
            Construct refactoredPath using modelOutputDir and index
            Read refactoredCode from refactoredPath
            // Indent and replace code
            Get indentation from originalCode[startLine-1]
            Apply indentation to each line of refactoredCode
            Replace lines in originalCode from startLine to endLine with indented
                refactoredCode
            Write modified code back to originalPath
        END FOR
    CATCH Errors
        Output error
    END TRY
END PROCEDURE
```

Figure 3.8: Pseudo code for replacing code.

Finalize: The last step initiates finalize.py to complete the refactoring process by parsing the JSON output from the analysis step, extracting each finding and collecting the meta data, and subsequently replacing it with the LLMs refactored version. Specifically, the script locates and reads the relevant lines in the original file, applies indentation to the newly generated code, and then rewrites the source file in place. This ensures that the final updated code includes the LLM-refactored insertions exactly where needed.

3.5 Validation

In order to assess whether or not the functional requirements are correctly met, a series of automated tests will check if the system is correctly performing the refactoring. Not necessarily whether the large language model is producing accurate code. In order to verify that refactoring is being done correctly, we have it change simple code to maintain focus on the pipeline. In the testing scenarios, a combination of **unit** and **integration** tests are used to verify correct program flow and the accurate representation of the findings in the output reports. In specific, a test checks whether a rule was correctly found, confirming that Semgrep has done its part, and after, whether the generated corrections are to be found in the augmented source code as expected.

3.6 Limitations

The system falls short on the reliability and flexibility characteristics, specified in ISO/IEC 25010. The first is the inability to handle multi-file anti-patterns. The current pipeline is only able to detect patterns within a single file and not able to link these file findings across files, therefore making it harder to detect architecture level anti-patterns. The inherent variability in large language models can occasionally lead to incorrect syntax or incomplete updates. The system does not have a formal rollback mechanism in order to restore old versions. From a performance perspective, large codebases have not been tested for this system and they might experience long analysis and transformation times due to the sheer amount of scanning and generative processing work. While Refactor does demonstrate maintainability for all sub-characteristics, there are major limitations in other characteristics, thus showing the need for continued improvement.

3.7 Conclusion

Refacturbo answers **RQ1** by utilizing a static analysis engine that identifies anti-patterns, used in the creation of targeted prompts. These prompts are then processed by an LLM to produce refactored code that replaces the original code. A human-in-the-loop validation step ensures errors are caught before automatically applying changes. This design demonstrates how LLMs can be systematically used to detect and remediate code anti-patterns in a modular, two-stage pipeline.

Chapter 4

Methodology for evaluating prompting strategies on energy consumption

This study evaluates the energy efficiency of code generated with several distinct prompting strategies, following an academic and structured methodology for energy experiments informed by the study 'A Process for Analysing the Energy Efficiency of Software' authored by Mancebo *et al.* [44]. We use the seven phase energy measurement process as a guide for this study.

4.1 Experiment tasks

The experiment evaluates 12 configurations, derived from 1 LLM, 3 anti-patterns, and 4 prompting strategies. The activities are ordered chronologically to facilitate reproducibility. The results of the conducted experiments are included in the replication package[1].

1. **Construct programs and prompts:** We wrote a Java program for each anti-pattern. This is the unoptimized version. After that, we implemented the prompting strategies.
2. **Code Generation:** We used the prompts to generate code variants with the selected large language model.
3. **Testbed Preparation:** We prepared the Device Under Test (DUT) by disabling non-essential processes, turning the internet and Bluetooth off, ensuring the battery was at 100%, and unplugging external devices.
4. **Baseline Measurement:** We measured the DUT idle power consumption by running the 'sleep' command to establish a baseline. The mean energy consumption in Joules is calculated and stored for the test case experiments.
5. **Test Execution:** We set the internal repetition rate to a value that increases program execution time beyond 3 seconds in order to counter the limited sampling frequency of 2Hz. For each configuration, we execute the test case 50 times. Then we integrate the sample power with the composite trapezoidal rule to calculate joules. The value is corrected for idle energy use.
6. **Data Compilation and Cleanup:** After each configuration run we compile the collected data into CSV format. We clean the testbed by moving the generated files to their respective folder in the replication package.
7. **Data Analysis:** We run statistical analysis on the collected compiled data, using the Shapiro-Wilk test for normality and the Mann-Whitney U test to compare energy consumption between unoptimized and optimized code variants.

[1]Replication package: https://github.com/saifrashed/refacturbo-experiment

4.2 Experiment design

4.2.1 Variables

We examine five independent variables: unoptimized code (baseline) and four optimized code variants. The primary dependent variable is the mean energy consumption of each variant, computed to joules and corrected for baseline idle energy consumption.

4.2.2 Hypothesis

The hypothesis for the experiments are shown below, where $\mu_{\text{unoptimized}}$ is the mean energy consumption of the unoptimized code, and $\mu_{\text{optimized}}$ is the mean energy consumption of the optimized code.

1. **Null Hypothesis (H_0):** There is no statistically significant difference in mean energy consumption between the unoptimized code and the optimized code.

$$H_0 : \mu_{\text{unoptimized}} = \mu_{\text{optimized}} \tag{4.1}$$

2. **Alternative Hypothesis (H_1):** The optimized code shows a statistically significant reduction in mean energy consumption compared to the unoptimized code.

$$H_1 : \mu_{\text{unoptimized}} > \mu_{\text{optimized}} \tag{4.2}$$

If we observe a statistically significant difference in a one-tailed test, the null hypothesis can be rejected. We support the alternative hypothesis if the mean energy consumption of the optimized code is significantly lower than that of the unoptimized code.

4.2.3 LLM selection

We use OpenAI's GPT-4o model with the temperature parameter set on 1.0. The reason for this is primarily ease-of-use, public access via their official API, and high rankings in energy performance benchmarks [2] [35].

4.2.4 Prompt template

We present a series of prompt templates in Appendix A in Figures 1 through 4. First, a zero-shot template (Figure 1) provides only a descriptive task instruction where no examples are included. The one-shot template (Figure 2) allows for a single example. The few-shot template (Figure 3) contains a series of examples. Finally, the chain-of-thought template (Figure 4) includes reasoning steps with answers.

4.2.5 Rule selection

In order to build a collection of substantiated anti-patterns we looked at a combination of sources. First and foremost is the "Quality Measure Elements for Automated Source Code Performance Efficiency Measure" in the ISO/IEC 5055:2021 standard for software quality measurement.[3] The standard itself only contains specifications but it refers to the Common Weakness Enumeration (CWE) for more detailed explanations. Additionally, we look at online sources such as EcoCode[4] and Sonar [5] who both publish a list of rule databases. In most cases these listed rules align with one of the specifications in the ISO standard but this is not always explicitly stated in the descriptions. In order to identify valid and applicable rules we make use of the conditions shared by EcoCode[6]. We did minor modifications to the original rules to suit this study:

- It must be statically detectable.
- It should return single boolean response (True/False).
- It detects code that degrades performance efficiency (defined in ISO/IEC 5055:2021[7]).

[2]https://evalplus.github.io/evalperf.html
[3]`https://www.iso.org/standard/80623.html`
[4]`https://github.com/green-code-initiative/creedengo-rules-specifications/blob/main/RULES.md`
[5]`https://rules.sonarsource.com/java/RSPEC-3631/?search=performance`
[6]https://github.com/green-code-initiative/creedengo-rules-specifications/tree/main/docs/rules
[7]https://www.iso.org/obp/ui/en/#iso:std:iso-iec:5055:ed-1:v1:en

The sources consist of explanations and in some cases mitigation and problem examples. These will serve as source material for creating the prompts. Appendix B lists all selected rules.

P1: The use of an if-else chain instead of switch case

EcoCode and Sonar rule lists primarily highlight this anti-pattern. EcoCode explains that a chain of if-else statements triggers more comparisons for the JVM, which degrades performance. The Sonar rule list recommends using a switch statement to address this issue and potentially boost performance. Sonar offers four non-compliant and four compliant examples for this rule.

P2: Creation of text using string concatenation

This pattern involves the '+=' operator within a LOOP construct to build a string, aligning with the CWE-1046[8] specification. The Common Weakness Enumeration warns that this string-building method creates a new object after each iteration, which slows program performance. To mitigate this, it advocates using a text buffer data element. In Java, this corresponds to 'StringBuilder'[9] and 'StringBuffer.' However, Sonar advises[10] against synchronized classes like StringBuffer, as they significantly hinder performance, a point reinforced by the Java documentation for JDK 5 enhancements [11]. Sonar provides one compliant and one non-compliant example.

P3: Excessive boxing of a primitive

This pattern appears in programs where developers box primitives with an unnecessary extra step, aligning with the CWE-1235[12] specification. The specification notes that auto-boxing incurs a performance cost and should only be used in specific cases. Thus, eliminating redundant boxing boosts performance. CWE offers one compliant and one non-compliant example.

4.3 Experiment operation

4.3.1 DUT specification

The Device Under Test (DUT) specifications are provided in Table 4.1.

Parameter	Value
Software	
Operating System	macOS Sequoia 15.4 (24E248)
Unix Kernel	Darwin Kernel Version 24.4.0
Java Version	JDK 21
Hardware	
Computer	MacBook Pro 13-inch, 2018 (C02XHQ2WJHC8)
CPU	Intel Core i5 2,3 GHz Quad-Core
RAM Type	2133 MHz LPDDR3
RAM Capacity	8 GB
GPU	Intel Iris Plus Graphics 655 (1536 MB)

Table 4.1: Specification of the Device Under Test (DUT).

We rely on the Intel-based architecture of the device, using an in-built interface for power monitoring. However, energy measurements on newer Macbook models with the Apple M-series chips are not as clearly defined, as these chips lack an equivalent hardware-level energy monitoring sensor.

[8]https://cwe.mitre.org/data/definitions/1046.html
[9]https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html
[10]https://rules.sonarsource.com/java/RSPEC-3631/?search=performance
[11]https://docs.oracle.com/javase/8/docs/technotes/guides/performance/speed.html
[12]https://cwe.mitre.org/data/definitions/1235.html

4.3.2 Instrumentation

Approaches for software energy use involve measurement or modeling. Measurement, which can be hardware-based or software-based, directly captures consumption through power monitors or processor features at a specified sampling rate; hardware-based measurements, though less invasive and more accurate, rely on external tools that can be costly [44, 45], whereas software-based approaches, are more accessible but prone to estimation errors. We make use of Intel's Running Average Power Limit interface, which uses on-chip sensors and model-specific registers for energy monitoring [45].

According to Stoico *et al.* [46], the principal physical quantities for measuring energy in software execution are electrical energy and power, with energy defined by the International System of Units (SI) and power representing energy consumed per unit time (in joules per second). The fundamental relationship of energy use is given in the equation below, where the energy (joules) equals the product of power (watt) and time (seconds):

$$Energy(J) = Power(W) \times Time(s) \tag{4.3}$$

To estimate the energy consumed, the instrument must be capable of measuring both power and time during a test. To achieve this, two components have been developed to automate measurements.

Sampler

In order to capture real-time power consumption metrics on an Intel-based macOS system a script was created, using the powermetrics utility at 2Hz (500 ms) sampling intervals [13]. Before running a target command, powermetrics is started in the background to measure combined CPU and GPU power usage. Each reading is timestamped and appended in comma-separated format to an output file, resulting in a record of power consumption. After the target command is finished, the powermetrics process is terminated.

Orchestrator

The Orchestrator is a Python CLI script that automates the measurement workflow by coordinating repeated runs of a target command, collecting power readings, and computing the final energy metrics. First, it invokes the measurement script multiple times with cooldown intervals, logging power consumption samples over time. These samples are integrated using the composite trapezoidal rule to obtain energy consumption of the measurement [47].

$$E_{\text{measurement}} = \int_{t_1}^{t_n} P(t)\,dt \approx \sum_{i=1}^{n-1} \frac{P(t_i) + P(t_{i+1})}{2} \cdot (t_{i+1} - t_i) \tag{4.4}$$

Where:

- $E_{\text{measurement}}$: Approximate total energy measured over the time interval from t_1 to t_n.
- $P(t)$: Power function (a continuous function of time t) to be integrated.
- t_1: Initial time point of the integration interval.
- t_n: Final time point of the integration interval.
- t_i: i-th time point in the sequence of n subintervals.
- $P(t_i)$: Value of the power function at time t_i.
- $P(t_{i+1})$: Value of the power function at time t_{i+1}.
- $t_{i+1} - t_i$: Width of the i-th subinterval (time step).
- n: Number of subintervals used in the approximation.

Essentially, by averaging the first and second power value in a time step, and multiplying it by the time difference, a trapezoid is approximated under the curve. The sum of these gives total joules. If the baseline time and power constants are set, the script subtracts the baseline consumption term from the measured consumption to calculate the corrected energy consumption. The calculation for a time independent baseline measurement is provided by Mancebo *et al.* [44].

[13]Powermetrics: https://firefox-source-docs.mozilla.org/performance/powermetrics.html

$$E_{\text{baseline}} = \frac{\overline{\text{EC}}_{\text{Baseline}}}{\overline{T}_{\text{Baseline}}} \times \overline{T}_{\text{Measurement}} \tag{4.5}$$

$$E_{\text{corrected}} = E_{\text{measurement}} - E_{\text{baseline}} \tag{4.6}$$

This yields the net energy usage attributable to the target command. However there is a error margin due to a delay in calling the target command. The delay can be considered problematic if it is not stable; however, if stable, it can be treated as a fixed offset, negligible in group comparisons. Finally, the Orchestrator compiles summary statistics and performs normality checks to support subsequent analysis.

4.4 Statistical analysis

To assess data distribution and compare sample groups, several tests are used. First, to decide whether data follows a normal distribution and thus justifies parametric statistical methods, the Shapiro-Wilk test is applied [48]. If the Shapiro-Wilk test suggests non-normality, we use the non-parametric Mann-Whitney U test [49]. However, if data is found to be normally distributed, a two-sample *t*-test can be used to compare the means of two independent groups [50].

Shapiro-Wilk Test

We use the Shapiro-Wilk test to evaluate whether a given sample comes from a normally distributed population [48]. The test generates a p-value that can either be above or below a significance threshold. If it is below the data is non-normal, conversely, if it exceeds the threshold, the data is normally distributed.

t-Test

The t-test [50] compares the means of two independent samples and can only work if the sample is normal. The outcome of the test is a p-value that indicates whether the difference in two samples is statistically significant. If the Shapiro-Wilk test confirms that both samples follow a normal distribution and the t-test p-value is below 0.05 the difference can be considered significant.

Mann-Whitney U Test

The Mann-Whitney U test [49] is a non-parametric test, this implies that it can be applied when the data is not normally distributed, and thus fails the Shapiro-Wilk test. As with the t-test, a p-value is calculated. If this p-value is below 0.05 the difference can be considered significant.

Chapter 5

Results

In this chapter we present the results from the energy experiments.

5.1 Baseline measurement

Table 1.1 shows the baseline energy measurements recorded for the device at idle. The mean idle energy consumption was 18.71 Joules with a median of 18.63 Joules. The standard deviation was 1.59. This indicates a slight variance around the mean. The minimum and maximum energy use was 17.15 Joules and 28.16 Joules. This suggests that while almost every sample clustered around the mean, some values pushed the upper bound higher.

Table 5.1: The energy consumption (J) of idle DUT measurement.

Metric	Value
Mean	18.71
Standard Deviation	1.59
Median	18.63
Min	17.15
Max	28.16
Coefficient of Variation (%)	8.52

Overall, these baseline figures show that at idle the device has some variance in energy use. This could likely be due to changes in system processes, the operating state, or possibly background tasks.

5.2 P1: The use of an if-else chain instead of switch case

Table 5.2: The duration (s) of profiling for P1.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	3.87	0.50	0.50	0.50	0.50
Standard Deviation	0.23	1.50e-3	1.10e-3	1.10e-3	1.30e-3
Median	4.02	0.50	0.50	0.50	0.50
Min	3.52	0.50	0.50	0.50	0.50
Max	4.03	0.51	0.51	0.51	0.51
Coefficient of Variation (%)	6.02	0.30	0.22	0.21	0.26

The unoptimized variant has an average profile duration of 3.87 seconds with a standard deviation of 0.23. In contrast, all the optimized versions show a lower average duration at around 0.50 seconds with small variation. The coefficient of variation is higher for the unoptimized run than for the optimized code. This indicates much more consistent timing for the latter.

Table 5.3: The power (W) during profiling for P1.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	17.60	20.39	20.50	20.58	20.18
Standard Deviation	0.22	1.19	1.07	0.96	1.07
Median	17.62	20.05	20.71	20.76	20.22
Min	17.12	19.16	19.04	19.19	18.91
Max	18.12	25.52	24.00	23.16	23.29
Coefficient of Variation (%)	1.22	5.84	5.22	4.67	5.32

The unoptimized code uses the least power with 17.60 Watt and it has a low standard deviation of 0.22. The optimized code variants use more power, approximately 20 Watt, but it shows much more variability due to the standard deviation range between 0.96 Watt and 1.19 Watt. The coefficient of variation is the least for the unoptimized version 1.22% and generally higher for the optimized code versions. This means that there is more spread relative to the mean for the optimized versions.

Table 5.4: the corrected energy consumption (J) for P1.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	49.22	7.91	7.97	8.01	7.82
Standard Deviation	2.98	0.59	0.54	0.48	0.54
Median	50.69	7.73	8.10	8.11	7.83
Min	43.98	7.28	7.25	7.31	7.19
Max	53.22	10.43	9.74	9.31	9.49
Coefficient of Variation (%)	6.06	7.52	6.78	5.97	6.94

The unoptimized variant consumes 49.22 joules with a standard deviation of 2.98. The optimized code versions consistently have lower and more stable energy consumption. Furthermore, the optimized variants show relatively small spread, with a standard deviation under 0.60 joules.

Table 5.5: Percentage reduction in corrected energy consumption compared to unoptimized code for P1.

Prompt	Zero shot (%)	One shot (%)	Few shot (%)	CoT (%)
Reduction	83.93	83.81	83.73	84.11

Each optimization achieves a reduction of approximately 83% in comparison to the unoptimized code. The lowest being the few shot optimization and the highest being the chain of thought optimization. However, the differences are minor.

5.2.1 Normality Testing

Table 5.6: Shapiro-Wilk P-values for P1

Unoptimized	Zero shot	One shot	Few shot	CoT
0.0000	0.0000	0.0015	0.0648	0.0001

The samples for the unoptimized, zero shot, one shot and chain of thought are all non-normal, due to having a p-value beneath 0.05. However, the few shot optimization does not reject normality and is therefore normal.

5.2.2 Hypothesis Testing

Table 5.7: Mann-Whitney U P-values for P1

	Zero shot	One shot	Few shot	CoT
Unoptimized Code	0.000002438	0.001248	0.03298	0.00008366

Due to the fact that the unoptimized code is non-normal, we employ a one-tailed test, Mann-Whitney U-Test. Since all p-values are below the significance level, we are able to reject the null hypothesis, in favor of the alternative hypothesis due to the direction of the data.

5.3 P2: Creation of text using string concatenation

Table 5.8: The duration (s) of profiling for P2.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	9.25	1.26	2.78	4.02	1.51
Standard Deviation	0.47	0.25	0.38	3.20e-3	1.60e-3
Median	9.05	1.26	2.51	4.02	1.51
Min	9.04	1.00	2.51	4.02	1.50
Max	12.05	1.51	3.52	4.03	1.52
Coefficient of Variation (%)	5.12	20.18	13.79	0.08	0.11

The first thing we can notice is that unoptimized code has the highest average duration of all the code variants. Zero shot achieves the lowest duration and this is followed by the one shot optimization, chain of thought and the highest, few shot. Few shot achieves the lowest variability with its coefficient of variation at 0.08. This is followed by chain of thought at 0.11.

Table 5.9: The power (W) during profiling for P2.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	25.53	20.64	21.67	22.39	21.83
Standard Deviation	1.43	0.56	0.98	0.84	0.63
Median	25.02	20.57	22.06	22.18	21.60
Min	23.40	19.61	20.07	21.13	21.11
Max	31.75	22.41	23.27	24.77	23.62
Coefficient of Variation (%)	5.60	2.70	4.50	3.77	2.86

The unoptimized code variant achieves the highest average power use and it has the least stable sample with a coefficient of variation at 5.60. The optimized code variants show slightly lower mean power use and relatively smaller variations overall. The smallest of these is zero shot at 2.70% and the highest is the one-shot optimization at 4.50%.

Table 5.10: the corrected energy consumption (J) for P2.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	193.27	20.02	47.11	69.34	25.26
Standard Deviation	21.85	4.22	5.76	3.14	0.92
Median	184.67	20.11	44.93	68.57	24.89
Min	170.05	15.20	38.60	64.30	24.26
Max	285.47	26.17	62.82	78.27	27.99
Coefficient of Variation (%)	11.30	21.08	12.23	4.53	3.62

The unoptimized code variant uses substantially more energy on average than its optimized counterparts. Across the optimization variants, we see that zero-shot achieves the lowest average at 20.02 J, but it has the highest variability with 21.08% of coefficient of variation. Few-shot and chain-of-thought are centered and they hang in between at 69 joules and 25.26 joules. And they have, from all the variants, the lowest variance with a coefficient variation at 4.53.

Table 5.11: Percentage reduction in corrected energy consumption compared to unoptimized code for P2.

Prompt	Zero shot (%)	One shot (%)	Few shot (%)	CoT (%)
Reduction	89.64	75.62	64.12	86.93

For P2 we can see that zero-shot achieves the highest percentage reduction from all the optimizations. The lowest among these is the few-shot optimization, and in between we see one-shot and chain-of-thought at approximately 80%.

5.3.1 Normality Testing

Table 5.12: Shapiro-Wilk P-values for P2

Unoptimized	Zero shot	One shot	Few shot	CoT
0.0000	0.0000	0.0003	0.0001	0.0000

All the variants result in p-values below 0.05. This indicates that none of the samples satisfy the normality assumptions, therefore being non-normal.

5.3.2 Hypothesis Testing

Table 5.13: Mann-Whitney U P-values for P2

	Zero shot	One shot	Few shot	CoT
Unoptimized Code	9.224e-8	0.0002062	0.0000555	0.0000016

Since every sample is non-normal, we use a one-tailed Mann-Whitney U-test. Each p-value is below the significance level and we can therefore reject the null hypothesis, in favor of the alternative hypothesis due to the direction of the data. This tells us that the optimized code uses significantly less energy if we compare it to the unoptimized code.

5.4 P3: Excessive boxing of a primitive

Table 5.14: The duration (s) of profiling for P3.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	6.03	1.51	1.51	1.51	1.50
Standard Deviation	3.00e-3	1.40e-3	1.50e-3	3.20e-3	0.03
Median	6.03	1.51	1.51	1.51	1.51
Min	6.02	1.50	1.50	1.49	1.31
Max	6.04	1.51	1.51	1.51	1.51
Coefficient of Variation (%)	0.05	0.09	0.10	0.21	1.83

The unoptimized variant takes the longest time to measure with an average of 6.03 seconds. This is followed with all the optimizations at a stable 1.5 seconds. The standard deviations for every variant is practically zero. This implies a low variability in the data. The highest among these is chain of thought achieving a coefficient of variation at 1.83%.

Table 5.15: The power (W) during profiling for P3.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	23.99	17.63	17.36	17.00	16.99
Standard Deviation	0.76	0.53	0.55	2.10	2.12
Median	24.34	17.77	17.38	16.42	16.49
Min	21.90	16.72	16.43	15.27	15.62
Max	24.96	18.49	18.41	28.33	30.47
Coefficient of Variation (%)	3.15	3.01	3.15	12.36	12.47

The unoptimized variant achieves the highest mean power. The optimizations are slightly lower at around 17 watts. From all the optimizations we see that few shot and chain of thought have a particularly high variability with a coefficient of variation at 12.36% and 12.47%.

Table 5.16: the corrected energy consumption (J) for P3.

Prompt	Unoptimized	Zero shot	One shot	Few shot	CoT
Mean	116.50	18.63	18.21	18.19	18.21
Standard Deviation	4.78	0.68	0.76	3.32	2.83
Median	118.45	18.72	18.25	17.28	17.66
Min	104.40	17.39	16.95	15.69	16.27
Max	123.03	19.93	19.90	36.82	36.37
Coefficient of Variation (%)	4.10	3.64	4.16	18.27	15.52

The unoptimized code variant consumes the most energy at 116.50 joules. The optimizations are substantially lower than that, averaging out at around 18-19 joules. From all the values, few shot and chain of thought have the highest coefficient of variation, indicating that these variables are not stable.

Table 5.17: Percentage reduction in corrected energy consumption compared to unoptimized code for P3.

Prompt	Zero shot (%)	One shot (%)	Few shot (%)	CoT (%)
Reduction	84.01	84.37	84.39	84.37

The optimizations for P3 show a consistent reduction of 84%, having small differences in between, showing that the few shot achieves the highest, while the zero shot achieves the lowest. However, these differences are again minor and don't suggest anything out of the ordinary.

5.4.1 Normality Testing

Table 5.18: Shapiro-Wilk P-values for P3

Unoptimized	Zero shot	One shot	Few shot	CoT
0.0003	0.0759	0.0918	0.0000	0.0000

After applying the Shapiro-Wilk tests on all the code variants, the unoptimized few-shot and chain-of-thought variants have shown not to follow a normal distribution. In contrary, the zero-shot and one-shot code variants do meet the criteria for normality.

5.4.2 Hypothesis Testing

Table 5.19: Mann-Whitney U P-values for P3

	Zero shot	One shot	Few shot	CoT
Unoptimized Code	0.02637	0.03575	3.489e-10	2.816e-12

Since the unoptimized code is not normal, we will employ a one-tailed Mann Whitney U-test. The results show that for all p-values they are each below the significance level and as a consequence we can reject the null hypothesis. The data direction therefore supports the alternative hypothesis.

Chapter 6

Discussion

This chapter discusses the results of the energy measurements (P1, P2, and P3). Subsequently, the second research question will be addressed, followed by an identification of threats to validity.

Finding 1: LLM optimizations consistently lead to significantly reduced energy expenditure when compared to the unoptimized code.

Across all three patterns, the results indicate that unoptimized code consumes noticeably more energy than optimized variants. In each scenario, the Mann-Whitney U tests found statistically significant reductions in energy usage, confirming that refactoring anti-patterns can have a measurable, positive impact on a program's energy efficiency.

Finding 2: LLM optimizations show small differences in energy consumption at P1 and P3.

Although refactored code variants outperformed the baseline in each pattern, results for *P1* and *P3* indicate that no single prompting strategy, whether zero-shot, one-shot, few-shot, or chain-of-thought, consistently and substantially outperforms the other. The corrected energy consumptions among these optimizations remain comparatively close, and report only minor differences.

To address the first research question, we observed that Refacturbo is able to use a static analysis engine to identify anti-patterns and use it to generate targeted prompts. These are then fed into a large language model that produces a refactored version, an energy-optimized version, that in turn replaces the original code. By combining these components in a two-stage pipeline, we are able to automatically refactor single file anti-patterns. With this system in place, we were able to generate code for four different prompt strategies in order to evaluate them for the second research question. The results show that all optimized versions achieve substantial energy savings relative to the unoptimized code. Yet, the specific optimizations show small differences in energy impact for two of the tested patterns. Therefore, while the large language model is able to optimize code, generally reducing energy consumption, no single prompting approach consistently outperforms the other.

6.1 Threats to validity

Construct validity: The energy experiments rely on system-level power metrics via the PowerMetrics application in order to approximate the energy consumption of a program. Although RAPL-based measurements are widely used, they capture overall system power, rather than isolating the Java process itself. This implies that background processes or minor OS activities or programs are able to introduce noise into the measurements. The study also assumes that refactoring anti-patterns into optimized versions yields energy-efficient code. While this is consistent with the measured data, any of these gains can also be a byproduct of compiler optimizations or other system-level factors, rather than the code changes alone.

Internal Validity: Even with attempts to turn off non-essential processes of the device, some background tasks could still run intermittently, such as indexing or OS daemons, affecting the power use.

Variations in the baseline energy consumption across runs due to short-lived processes or temperature throttling could result into slight inaccuracies in the energy measurements. Furthermore, power was sampled at 2 Hz, every 500 ms. Changes outside this sampling interval are smooth and this might potentially not capture small but large power spikes or dips. Additionally, Java's just-in-time[1] compilation and potential caching[2] within the JVM could affect execution performance over repeated runs. While repeating the test multiple times can reduce this randomness, it might never be completely eliminated. Finally, the calls made by the orchestrator and the sampler script also consume resources. Even though this study attempts to subtract a baseline, if the measurement overhead is not stable, residual overhead could have an influence on the results.

External Validity: The experiments are run on a 2018 Intel based MacBook Pro. Any characteristics such as clock speed, CPU architecture, cooling differs by hardware vendor and model. Newer MacBook Pros for example don't have an Intel chip and are not able to interface with Intel RAPL. Therefore the results cannot generalize to other systems. Mac OS behavior, process scheduling and power management differ from other operating systems and the measured impacts may not replicate on Windows or Linux. Furthermore the code is generated by a single model GPT-4o using four prompting strategies. While these represent typical generation methods, other large language models may produce code fragments differently, thus impacting the energy usage.

[1]https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdev/Oracle-JVM-JIT.html
[2]https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/codecache.htm

Chapter 7

Related work

This research looks at automated refactoring of green anti-patterns by using large language models. This topic is related to multiple studies. However, we can see in the table below that these studies use a variety of methods to achieve that goal. This study differentiates itself by using the ISO/IEC 25010 specification to make static analysis rules in order to generate targeted prompts. Prior work by Lin *et al.* [3] does hint at the use of static analysis in a tool called CodePlan. However, this tool does not target energy efficiency problems. Another work by Dearing *et al.* [10] shows that they do use a multistage pipeline to catch such problems, similar to Refacturbo. However, it does not use the ISO specification as a guide for remediating these anti-patterns.

Table 7.1: Table of Related Studies

Author	**Project**	**Description**
Lin *et al.* [3] (2025)	Development of an code optimizer, a system designed to automatically refactor source code for performance at scale.	A code optimizer that automates refactoring of source code to improve the performance at scale. It uses historical commits to identify and address performance anti-patterns in a codebase.
Dearing *et al.* [10] (2025)	Development of an automated refactoring framework.	An automated refactoring framework that uses large language models to generate energy efficient parallel code for target systems.
Cordeiro *et al.* [51] (2024)	Evaluation of an LLM, specialized in code generation.	Evaluates a large language model that is optimized for code generation. In refactoring code from 30 open source Java projects.
Shypula *et al.* [4] (2024)	Framework for improving LLMs for code optimizations	A framework for using LLMs to do high-level program optimizations. It uses prompting strategies such as retrieval-based prompting, but also fine-tuning.
Peng *et al.* [39] (2024)	Developed a tool compatible with various programming languages to improve software sustainability	Introduces an application that uses LLMs to optimize software for energy-efficiency. It shows that code generated by the model can achieve better performance than compiler optimizations for small programs.

Chapter 8

Conclusion

In this research, a system has been developed that can automate refactoring for inefficient code patterns by using large language models. In order to do this, we set out two research questions. The first looked at how such a system can be developed, while the second evaluated the generated code on energy consumption. For answering the first question, we introduced Refacturbo, a two-stage pipeline that can use static analysis to find problematic code patterns, generate targeted prompts, and then use this as an input for an LLM to refactor these and subsequently replace the original code. While the pipeline is able to do these single-file refactorings, it is limited in that it cannot connect findings across files and solve larger architectural-level patterns. These show areas for further improvement. After collecting three different green anti-patterns, we were able to answer the second question by having Refacturbo generate four distinct optimizations for three different unoptimized Java programs containing a performance anti-pattern, which were then evaluated for energy consumption using custom-built instrumentation. Across all three anti-patterns, the optimized code versions achieved an average reduction of 82%, ranging from a 64% up to 90%. Therefore, the optimized versions consistently reduced the energy cost significantly relative to the unoptimized code. However, there is not one optimization that universally outperformed the other. Therefore, this research demonstrates that large language models can be used to target specific green anti-patterns and transform them into more energy-efficient optimizations, but it also shows that there is much to improve in the current system for future work.

8.1 Future directions

Future directions following this research could go into two paths, where the first path is about extending the capability of the current Refacturbo system, giving it the feature of connecting static analysis findings with each other, in order to solve larger architectural patterns. These patterns can be found in the ISO specification used for this study. However, making such a system requires at least to think about how to create a dynamic system, whereby the context of files are exchanged with each other in separate remediation prompts, and how the output from the LLM can be used in replacing the original code. Furthermore, there is room to see how a rollback mechanism can be created, in order to restore all changes after an optimization has not succeeded. One could think about, for example, using git as one of these methods.

Another way to extend the capability of Refacturbo is to make it integratable in CI-CD pipelines. Doing this could be achieved by having some threshold value for detected anti-pattern and if crossed cancel the build and create a report showing all the possible patterns that can be remediated and what kind of refactoring would be needed for that. This makes Refacturbo more useful in the software engineering workflow than it currently is.

The second path is testing the system as it is now, on a larger scale. Currently, we test on small isolated patterns that are repeated internally in order to make for a testable configuration. However, this system has not yet been tested on larger open source projects, and in order to see and evaluate where the system finds its limits, it could be worthwhile to test it on open source systems.

Acknowledgements

First of all, I want to thank Ana Oprescu, my supervisor, for allowing me to research this topic that is so relevant in this time. Also, she and her team managed to create a great experience throughout the master program. I would also like to thank various experts and contributors, specifically Pepijn van de Kamp and Gushu Gao from Software Improvement Group, who gave me access to Sigrid, a tool that was instrumental in understanding green code anti-patterns. My thanks go also to my classmates who have been with me since the pre-master and who regularly met with me during this process. Finally, I thank my family and friends who have been reassuring me throughout the process and encouraging me making this achievement possible.

Bibliography

[1] S. Ayers, S. Ballan, V. Gray, and R. McDonald, "Measuring the emissions and energy footprint of the ict sector: Implications for climate action," International Telecommunication Union, World, Publication 186435, Mar. 20, 2024, English. [Online]. Available: `https://documents.worldbank.org/en/publication/documents-reports/documentdetail/099121223165540890/p17859702a98880540a4b70d57876048abb`.

[2] European Commission, "The european green deal," European Commission, Brussels, Communication COM(2019) 640 final, Dec. 11, 2019. [Online]. Available: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=COM%3A2019%3A640%3AFIN`.

[3] H. Lin *et al.*, *ECO: An LLM-Driven Efficient Code Optimizer for Warehouse Scale Computers*, arXiv:2503.15669 [cs], Mar. 2025. DOI: `10.48550/arXiv.2503.15669`. [Online]. Available: `http://arxiv.org/abs/2503.15669` (visited on 07/06/2025).

[4] A. Shypula *et al.*, *Learning Performance-Improving Code Edits*, arXiv:2302.07867 [cs], Apr. 2024. DOI: `10.48550/arXiv.2302.07867`. [Online]. Available: `http://arxiv.org/abs/2302.07867` (visited on 07/06/2025).

[5] J. Balanza-Martinez, P. Lago, and R. Verdecchia, "Tactics for Software Energy Efficiency: A Review," en, in *Advances and New Trends in Environmental Informatics 2023*, V. Wohlgemuth, D. Kranzlmüller, and M. Höb, Eds., Series Title: Progress in IS, Cham: Springer Nature Switzerland, 2024, pp. 115–140, ISBN: 978-3-031-46901-5 978-3-031-46902-2. DOI: `10.1007/978-3-031-46902-2_7`. [Online]. Available: `https://link.springer.com/10.1007/978-3-031-46902-2_7` (visited on 05/26/2025).

[6] H. Al-shakarjy and D. Basheer Taha, "Software Code Refactoring: A Comprehensive Review," en, *Journal of Education and Science*, vol. 32, no. 1, pp. 71–80, Mar. 2023, ISSN: 2664-2530. DOI: `10.33899/edusj.2023.137163.1298`. [Online]. Available: `https://edusj.mosuljournals.com/article_177130.html` (visited on 06/22/2025).

[7] P. I. Sommerhalter, "Calabash: An analysis framework and catalog for green code patterns in software," English, Available until 2031-08-19. No license applied; reading and citing permitted, distribution and reuse require author's permission., Master's thesis, Faculteit der Natuurwetenschappen, Wiskunde en Informatica, Universiteit van Amsterdam, Amsterdam, Netherlands, 2024. [Online]. Available: `https://dspace.uba.uva.nl/bitstreams/1d8b2220-ae17-4944-89fd-34789a3fe6db/download`.

[8] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" en, in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Torino Italy: ACM, Sep. 2014, pp. 1–10, ISBN: 978-1-4503-2774-9. DOI: `10.1145/2652524.2652538`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2652524.2652538` (visited on 05/12/2025).

[9] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, *An Empirical Study on the Potential of LLMs in Automated Software Refactoring*, arXiv:2411.04444 [cs], Nov. 2024. DOI: `10.48550/arXiv.2411.04444`. [Online]. Available: `http://arxiv.org/abs/2411.04444` (visited on 04/29/2025).

[10] M. T. Dearing, Y. Tao, X. Wu, Z. Lan, and V. Taylor, *Leveraging LLMs to Automate Energy-Aware Refactoring of Parallel Scientific Codes*, arXiv:2505.02184 [cs], May 2025. DOI: `10.48550/arXiv.2505.02184`. [Online]. Available: `http://arxiv.org/abs/2505.02184` (visited on 07/06/2025).

[11] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar, “Exploring ChatGPT’s code refactoring capabilities: An empirical study,” en, *Expert Systems with Applications*, vol. 249, p. 123 602, Sep. 2024, Publisher: Elsevier BV, ISSN: 0957-4174. DOI: `10.1016/j.eswa.2024.123602`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0957417424004676` (visited on 05/23/2025).

[12] T. Cappendijk, P. de Reus, and A. Oprescu, *Generating Energy-efficient code with LLMs*, Version Number: 1, 2024. DOI: `10.48550/ARXIV.2411.10599`. [Online]. Available: `https://arxiv.org/abs/2411.10599` (visited on 03/04/2025).

[13] V.-A. Cursaru *et al.*, *A Controlled Experiment on the Energy Efficiency of the Source Code Generated by Code Llama*, Version Number: 1, 2024. DOI: `10.48550/ARXIV.2405.03616`. [Online]. Available: `https://arxiv.org/abs/2405.03616` (visited on 03/04/2025).

[14] T. B. Brown *et al.*, *Language Models are Few-Shot Learners*, Version Number: 4, 2020. DOI: `10.48550/ARXIV.2005.14165`. [Online]. Available: `https://arxiv.org/abs/2005.14165` (visited on 03/04/2025).

[15] J. Wei *et al.*, *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, Version Number: 6, 2022. DOI: `10.48550/ARXIV.2201.11903`. [Online]. Available: `https://arxiv.org/abs/2201.11903` (visited on 03/04/2025).

[16] O. Le Goaer and J. Hertout, “ecoCode: A SonarQube Plugin to Remove Energy Smells from Android Projects,” en, in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Rochester MI USA: ACM, Oct. 2022, pp. 1–4, ISBN: 978-1-4503-9475-8. DOI: `10.1145/3551349.3559518`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3551349.3559518` (visited on 05/12/2025).

[17] R. Sehgal, D. Mehrotra, R. Nagpal, and R. Sharma, “Green software: Refactoring approach,” en, *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 4635–4643, Jul. 2022, ISSN: 13191578. DOI: `10.1016/j.jksuci.2020.10.022`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S1319157820305164` (visited on 05/12/2025).

[18] İ. Şanlıalp, M. M. Öztürk, and T. Yiğit, “Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices,” en, *Electronics*, vol. 11, no. 3, p. 442, Feb. 2022, ISSN: 2079-9292. DOI: `10.3390/electronics11030442`. [Online]. Available: `https://www.mdpi.com/2079-9292/11/3/442` (visited on 05/12/2025).

[19] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, “Tales from the Code #2: A Detailed Assessment of Code Refactoring’s Impact on Energy Consumption,” en, in *Software Technologies*, H.-G. Fill, M. Van Sinderen, and L. A. Maciaszek, Eds., vol. 1622, Series Title: Communications in Computer and Information Science, Cham: Springer International Publishing, 2022, pp. 94–116, ISBN: 978-3-031-11512-7 978-3-031-11513-4. DOI: `10.1007/978-3-031-11513-4_5`. [Online]. Available: `https://link.springer.com/10.1007/978-3-031-11513-4_5` (visited on 05/12/2025).

[20] O. Hamdi, A. Ouni, M. Ó. Cinnéide, and M. W. Mkaouer, “A longitudinal study of the impact of refactoring in android applications,” en, *Information and Software Technology*, vol. 140, p. 106 699, Dec. 2021, ISSN: 09505849. DOI: `10.1016/j.infsof.2021.106699`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0950584921001531` (visited on 05/12/2025).

[21] I. Sanlialp and M. M. Ozturk, “Investigating the Impact of Code Refactoring Techniques on Energy Consumption in Different Object-Oriented Programming Languages,” en, in *Artificial Intelligence and Applied Mathematics in Engineering Problems*, D. J. Hemanth and U. Kose, Eds., vol. 43, Series Title: Lecture Notes on Data Engineering and Communications Technologies, Cham: Springer International Publishing, 2020, pp. 142–152, ISBN: 978-3-030-36177-8 978-3-030-36178-5. DOI: `10.1007/978-3-030-36178-5_12`. [Online]. Available: `http://link.springer.com/10.1007/978-3-030-36178-5_12` (visited on 05/12/2025).

[22] D. Connolly Bree and M. Ó. Cinnéide, “Inheritance versus Delegation: Which is more energy efficient?” en, in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, Seoul Republic of Korea: ACM, Jun. 2020, pp. 323–329, ISBN: 978-1-4503-7963-2. DOI: `10.1145/3387940.3392192`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3387940.3392192` (visited on 05/12/2025).

[23] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, Dec. 2018, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: `10.1109/TSE.2017.2757486`. [Online]. Available: `https://ieeexplore.ieee.org/document/8052533/` (visited on 05/12/2025).

[24] D. Kim, J.-E. Hong, I. Yoon, and S.-H. Lee, "Code refactoring techniques for reducing energy consumption in embedded computing environment," en, *Cluster Computing*, vol. 21, no. 1, pp. 1079–1095, Mar. 2018, ISSN: 1386-7857, 1573-7543. DOI: `10.1007/s10586-016-0691-5`. [Online]. Available: `http://link.springer.com/10.1007/s10586-016-0691-5` (visited on 05/12/2025).

[25] R. Verdecchia, R. Aparicio Saez, G. Procaccianti, and P. Lago, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," pp. 365–345. DOI: `10.29007/dz83`. [Online]. Available: `https://easychair.org/publications/paper/MxpT` (visited on 05/12/2025).

[26] R. Pérez-Castillo and M. Piattini, "Analyzing the Harmful Effect of God Class Refactoring on Power Consumption," *IEEE Software*, vol. 31, no. 3, pp. 48–54, May 2014, ISSN: 0740-7459, 1937-4194. DOI: `10.1109/MS.2014.23`. [Online]. Available: `https://ieeexplore.ieee.org/document/6728938/` (visited on 05/12/2025).

[27] H. Naveed *et al.*, *A Comprehensive Overview of Large Language Models*, arXiv:2307.06435 [cs], Oct. 2024. DOI: `10.48550/arXiv.2307.06435`. [Online]. Available: `http://arxiv.org/abs/2307.06435` (visited on 04/08/2025).

[28] A. Vaswani *et al.*, *Attention Is All You Need*, arXiv:1706.03762 [cs], Aug. 2023. DOI: `10.48550/arXiv.1706.03762`. [Online]. Available: `http://arxiv.org/abs/1706.03762` (visited on 04/08/2025).

[29] A. Cho *et al.*, *Transformer Explainer: Interactive Learning of Text-Generative Models*, arXiv:2408.04619 [cs], Aug. 2024. DOI: `10.48550/arXiv.2408.04619`. [Online]. Available: `http://arxiv.org/abs/2408.04619` (visited on 07/02/2025).

[30] D. Jurafsky and J. H. Martin, *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition* (Prentice Hall series in artificial intelligence), eng. Upper Saddle River, N.J: Prentice Hall, 2000, ISBN: 978-0-13-095069-7.

[31] Y. Liu *et al.*, *Understanding LLMs: A Comprehensive Overview from Training to Inference*, arXiv:2401.02038 [cs], Jan. 2024. DOI: `10.48550/arXiv.2401.02038`. [Online]. Available: `http://arxiv.org/abs/2401.02038` (visited on 04/09/2025).

[32] P. Liang *et al.*, *Holistic Evaluation of Language Models*, arXiv:2211.09110 [cs], Oct. 2023. DOI: `10.48550/arXiv.2211.09110`. [Online]. Available: `http://arxiv.org/abs/2211.09110` (visited on 04/09/2025).

[33] S. Samsi *et al.*, *From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference*, Version Number: 1, 2023. DOI: `10.48550/ARXIV.2310.03003`. [Online]. Available: `https://arxiv.org/abs/2310.03003` (visited on 03/04/2025).

[34] R. Rubei, A. Moussaid, C. di Sipio, and D. di Ruscio, *Prompt engineering and its implications on the energy consumption of Large Language Models*, Version Number: 1, 2025. DOI: `10.48550/ARXIV.2501.05899`. [Online]. Available: `https://arxiv.org/abs/2501.05899` (visited on 03/04/2025).

[35] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, *Evaluating Language Models for Efficient Code Generation*, arXiv:2408.06450 [cs], Aug. 2024. DOI: `10.48550/arXiv.2408.06450`. [Online]. Available: `http://arxiv.org/abs/2408.06450` (visited on 03/04/2025).

[36] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, *EffiBench: Benchmarking the Efficiency of Automatically Generated Code*, arXiv:2402.02037 [cs], Oct. 2024. DOI: `10.48550/arXiv.2402.02037`. [Online]. Available: `http://arxiv.org/abs/2402.02037` (visited on 03/04/2025).

[37] R. Qiu, W. W. Zeng, J. Ezick, C. Lott, and H. Tong, *How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark*, arXiv:2406.06647 [cs], Feb. 2025. DOI: `10.48550/arXiv.2406.06647`. [Online]. Available: `http://arxiv.org/abs/2406.06647` (visited on 03/04/2025).

[38] A. Shypula *et al.*, *Learning Performance-Improving Code Edits*, Version Number: 5, 2023. DOI: `10.48550/ARXIV.2302.07867`. [Online]. Available: `https://arxiv.org/abs/2302.07867` (visited on 03/04/2025).

[39] H. Peng *et al.*, *Large Language Models for Energy-Efficient Code: Emerging Results and Future Directions*, arXiv:2410.09241 [cs], Oct. 2024. DOI: 10.48550/arXiv.2410.09241. [Online]. Available: http://arxiv.org/abs/2410.09241 (visited on 07/06/2025).

[40] T. Vartziotis *et al.*, *Learn to Code Sustainably: An Empirical Study on LLM-based Green Code Generation*, arXiv:2403.03344 [cs], Mar. 2024. DOI: 10.48550/arXiv.2403.03344. [Online]. Available: http://arxiv.org/abs/2403.03344 (visited on 04/21/2025).

[41] Q. Dong *et al.*, *A Survey on In-context Learning*, Version Number: 6, 2023. DOI: 10.48550/ARXIV.2301.00234. [Online]. Available: https://arxiv.org/abs/2301.00234 (visited on 03/04/2025).

[42] A. Raventós, M. Paul, F. Chen, and S. Ganguli, *Pretraining task diversity and the emergence of non-Bayesian in-context learning for regression*, Version Number: 2, 2023. DOI: 10.48550/ARXIV.2306.15063. [Online]. Available: https://arxiv.org/abs/2306.15063 (visited on 04/11/2025).

[43] D. Feitosa, L. Cruz, R. Abreu, J. P. Fernandes, M. Couto, and J. Saraiva, "Patterns and Energy Consumption: Design, Implementation, Studies, and Stories," en, in *Software Sustainability*, C. Calero, M. Á. Moraga, and M. Piattini, Eds., Cham: Springer International Publishing, 2021, pp. 89–121, ISBN: 978-3-030-69969-7 978-3-030-69970-3. DOI: 10.1007/978-3-030-69970-3_5. [Online]. Available: https://link.springer.com/10.1007/978-3-030-69970-3_5 (visited on 07/03/2025).

[44] J. Mancebo, F. García, and C. Calero, "A process for analysing the energy efficiency of software," en, *Information and Software Technology*, vol. 134, p. 106 560, Jun. 2021, ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106560. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000446 (visited on 03/04/2025).

[45] F. Castor, *Estimating the Energy Footprint of Software Systems: A Primer*, arXiv:2407.11611 [cs], Jul. 2024. DOI: 10.48550/arXiv.2407.11611. [Online]. Available: http://arxiv.org/abs/2407.11611 (visited on 04/11/2025).

[46] V. Stoico, V. Cortellessa, I. Malavolta, D. Di Pompeo, L. Pomante, and P. Lago, "An Approach Using Performance Models for Supporting Energy Analysis of Software Systems," en, in *Computer Performance Engineering and Stochastic Modelling*, M. Iacono, M. Scarpa, E. Barbierato, S. Serrano, D. Cerotti, and F. Longo, Eds., vol. 14231, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2023, pp. 249–263, ISBN: 978-3-031-43184-5 978-3-031-43185-2. DOI: 10.1007/978-3-031-43185-2_17. [Online]. Available: https://link.springer.com/10.1007/978-3-031-43185-2_17 (visited on 04/17/2025).

[47] K. E. Atkinson, *An introduction to numerical analysis*, eng, 2. ed. New York: Wiley, 1989, ISBN: 978-0-471-50023-0.

[48] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," en, *Biometrika*, vol. 52, no. 3-4, pp. 591–611, Dec. 1965, ISSN: 0006-3444, 1464-3510. DOI: 10.1093/biomet/52.3-4.591. [Online]. Available: https://academic.oup.com/biomet/article-lookup/doi/10.1093/biomet/52.3-4.591 (visited on 05/22/2025).

[49] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," en, *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, Mar. 1947, ISSN: 0003-4851. DOI: 10.1214/aoms/1177730491. [Online]. Available: http://projecteuclid.org/euclid.aoms/1177730491 (visited on 05/22/2025).

[50] Student, "The Probable Error of a Mean," *Biometrika*, vol. 6, no. 1, p. 1, Mar. 1908, ISSN: 00063444. DOI: 10.2307/2331554. [Online]. Available: https://www.jstor.org/stable/2331554?origin=crossref (visited on 05/22/2025).

[51] J. Cordeiro, S. Noei, and Y. Zou, *An Empirical Study on the Code Refactoring Capability of Large Language Models*, arXiv:2411.02320 [cs], Nov. 2024. DOI: 10.48550/arXiv.2411.02320. [Online]. Available: http://arxiv.org/abs/2411.02320 (visited on 07/06/2025).

Appendix A

Prompt templates

```
Task: Task description

Code:
...
```

Figure A.1: zero-shot

```
Task: Task description

Example 1: Optimized code example

Code:
```

Figure A.2: one-shot example

```
Task: Task description

Example 1: Optimized code example 1

Example 2: Optimized code example 2

...

Code:
```

Figure A.3: few-shot example

```
Task: Task description

Q:
A:

Q:
A:

...

Code:
```

Figure A.4: a Chain of Thought (CoT) example

Appendix B

Anti-pattern rules

```
rules:
  - id: p1
    message: Avoid the use of a long if-else chain
    languages: [java]
    severity: WARNING
    patterns:
      - pattern: |
          if ($COND1) $STMT1;
          else if ($COND2) $STMT2;
          else if ($COND3) $STMT3;
          else if ($COND4) $STMT4;
          else if ($COND5) $STMT5;
          ...
      - pattern-not-inside: |
          if ($COND0) $STMT0;
          else if ($COND1) $STMT1;
          else if ($COND2) $STMT2;
          else if ($COND3) $STMT3;
          else if ($COND4) $STMT4;
          ...
    metadata:
      prompt: |
        prompt
```

Figure B.1: Semgrep rule file for P1

```
rules:
  - id: p2
    message: Avoid string concatenation in loop
    languages: [ java ]
    severity: WARNING
    patterns:
        - pattern-either:
        - pattern: |
            $F(...) {
                ...
                $STR = ...;
                ...
                for (...) {
                    ...
                    $STR += ...;
                    ...
                }
                ...
            }
    metadata:
      prompt: |
        prompt
```

Figure B.2: Semgrep rule file for P2

```
rules:
  - id: p3
    message: Unnecessary boxing detected. Avoid boxing/unboxing overhead.
    languages: [java]
    severity: WARNING
    patterns:
         - pattern: |
            public $RET $METHOD(...){
              ...
              Integer.valueOf(Integer.parseInt(...));
              ...
            }
    metadata:
      prompt: |
        prompt
```

Figure B.3: Semgrep rule file for P3

Appendix C

Programs

```
public class program {
    public static final int INTERNAL_REPETITION_COUNT = 500000000;

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        for (int x = 0; x < INTERNAL_REPETITION_COUNT; x++) {
            getFibonacci(49);
        }

        long endTime = System
                .currentTimeMillis();
        System.out.println("Execution time: " + (endTime - startTime) / 1000.0 + " 
            seconds");
    }

    static public long getFibonacci(int n) {
        if (n == 0)
            return 0;
        else if (n == 1)
            return 1;
        else if (n == 2)
            return 1;
        else if (n == 3)
            return 2;
        else if (n == 4)
            return 3;
        else if (n == 5)
            return 5;
        else if (n == 6)
            return 8;
        else if (n == 7)
            return 13;
        ...
        else
            return -1;
    }
}
```

Figure C.1: FibonacciP1

```
public class program {
    public static final int INTERNAL_REPETITION_COUNT = 10000;

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        for (int x = 0; x < INTERNAL_REPETITION_COUNT; x++) {
            generateFibonacciString(x);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Execution time: " + (endTime - startTime) / 1000.0);
    }

    public static String generateFibonacciString(int numRepetitions) {
        String result = "";
        int a = 0, b = 1;
        for (int x = 0; x < numRepetitions; x++) {
            result += a + (x < numRepetitions - 1 ? ", " : "");
            int next = a + b;
            a = b;
            b = next;
        }
        return result;
    }
}
```

Figure C.2: FibonacciP2

```
public class program {
    public static final int INTERNAL_REPETITION_COUNT = 500000000;

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        for (int x = 0; x < INTERNAL_REPETITION_COUNT; x++) {
            Integer n = Integer.valueOf(40);
            fibonacci(n);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Execution time: " + (endTime - startTime) / 1000.0 + " 
            seconds");
    }

    public static Integer fibonacci(Integer n) {
        if (n.intValue() <= 1) {
            return Integer.valueOf(n.intValue());
        }
        Integer a = Integer.valueOf(0);
        Integer b = Integer.valueOf(1);
        Integer fib = Integer.valueOf(0);

        for (Integer i = Integer.valueOf(2); i.intValue() <= n.intValue(); i = Integer.
            valueOf(i.intValue() + 1)) {
            fib = Integer.valueOf(a.intValue() + b.intValue());
            b = fib;
        }
        return fib;
    }
}
```

Figure C.3: FibonacciP3