Investigating Mutation-Guided Refactoring Using Large Language Models

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Natanael Djajadi
born in Capelle aan den IJssel, the Netherlands

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

Schuberg Philis
Boeing Avenue 271
Schiphol-Rijk, the Netherlands
www.schubergphilis.com

Investigating Mutation-Guided Refactoring Using Large Language Models

Author: Natanael Djajadi
Student id: 5228719

Abstract

In legacy systems, changing existing software is risky when developers cannot easily understand or test the behavior of the code, which limits evolvability. Tests can reduce this risk, and with mutation testing, surviving mutants can indicate where test oracles can be strengthened and where potential observability issues in production code are situated. This thesis investigates to what extent LLM-guided refactoring, guided by mutation testing results, can increase the observability of production code while not decreasing readability. A case study is performed on two open-source Java projects, JFreeChart and Bukkit, for 12 classes in total, with an LLM that uses an execution-validation workflow. The mutation score increased in all runs. In JFreeChart, the average increase was 5.96%, while in Bukkit it was 47.89%. However, the results show that a higher mutation score does not always mean that production-code observability improved, because some mutants were killed by stronger tests for already observable behavior. The readability impact was limited overall. This suggests that mutation-guided LLM refactoring is most useful when surviving mutants reveal behavior that is genuinely difficult to observe, and when the refactoring exposes it at the intended level of observability. Thus, a surviving mutant should first be interpreted as a decision problem: it may require refactoring, stronger tests, or no change.

Thesis Committee:

Chair:	Prof. Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Ir. I. Heitlager, Schuberg Philis
Committee Member:	Dr. M.A. Migut, Faculty EEMCS, TU Delft

Preface

"*Life is a tree of branching paths. Each choice erases the others. A goal is not a straight line, but a series of decisions that shape both the path and who we become.*"

This was what stuck with me after a conversation with William, the gamification specialist at Labs271. Since high school, one of my goals has been to make a positive impact by using technology. Finishing my Computer Science studies became an important step in this journey, and doing my thesis at a company became part of that path. During this opportunity of doing my thesis at Schuberg Philis, I was able to develop myself academically and gain insight into industry life and working culture. On the side, I even gained some side skills, such as making coffee with latte art for colleagues, and learning to make other drinks.

The completion of this thesis was made possible by the support and guidance of several people. First and foremost, thank you Andy, my main supervisor. I truly appreciate your guidance and feedback throughout these eight months, including learning to have more confidence in the process and decision-making. It is also a great honor for me to have started and ended my Bachelor's studies with you, and now to finish my Master's again with you. Thank you Ilja for the opportunity to do my thesis at Schuberg Philis, and for preparing me for the real world after this thesis, which includes opening myself up more to how things are done in practice. Thank you to all the interns, including Salim, Daniela, Shayan, Tijn, old Roos, new Roos, and Charlise, for the insightful and fun talks. Thank you also to the Lab colleagues and other employees at Schuberg Philis for being open to talk, discuss ideas, and help me learn more about working life. Thank you to the restaurant staff and cleaners, who made the office a pleasant place to work. Finally, thank you to my family and friends for their support throughout this journey. Most importantly, thank you to my parents for always supporting me and encouraging me to do my best in my studies from a young age, which helped me grow and eventually pursue my Master's degree.

Thus, this journey has shaped me in many ways. I hope you enjoy reading this thesis.

Natanael Djajadi
Delft, the Netherlands
June 5, 2026

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Research Questions . . . 2
- 1.2 Schuberg Philis and Lab271 . . . 3
- 1.3 Thesis Outline . . . 4

2 Background 5
- 2.1 Mutation Testing . . . 5
- 2.2 Testability, Observability, and Readability . . . 7
- 2.3 Refactoring . . . 8
- 2.4 LLM Agents and Development Tools . . . 9

3 Methodology 11
- 3.1 Approach . . . 11
- 3.2 Case Study Setup . . . 14

4 Results 23
- 4.1 JFreeChart . . . 23
- 4.2 Bukkit . . . 34
- 4.3 Follow-up Analysis of Selected Surviving Mutants . . . 50
- 4.4 Summary of Findings . . . 64

5 Discussion 67
- 5.1 Answering the Main Research Question . . . 67
- 5.2 Execution Observability versus Behavioral Observability . . . 68
- 5.3 Implications for Mutation-Guided LLM Refactoring . . . 69

5.4 Bug Discovery During LLM-Guided Refactoring 70
5.5 Threats to Validity 72
5.6 Ethical Considerations 74
5.7 Use of Generative AI 74

6 Related Work **77**
6.1 Mutation Testing and Code Observability 77
6.2 Automated Refactoring 78
6.3 LLM-based Refactoring 79
6.4 Readability 81

7 Conclusions and Future Work **83**
7.1 Contributions 83
7.2 Conclusions 84
7.3 Future work 85

Bibliography **87**

A 18 Candidate Projects **95**

B Additional Mutation Results **97**
B.1 Targeted Mutants Selected at the Start of Each Run 97
B.2 Outcome of Targeted Mutants After Refactoring 98

C JFreeChart Experiment Logs **99**

D Bukkit Experiment Logs **115**

List of Figures

2.1 Conceptual view of LLM-guided refactoring and later software evolution. The solid horizontal blue arrows show the refactoring step studied in this thesis, where I_n changes to I_{n+1} and T_n can be extended to T_{n+1}, while the specification remains S_n. The dotted arrows show a possible later evolution step. 9

3.1 Architecture of the proposed mutation-based LLM refactoring system. 12
3.2 Overview of the refactoring run workflow, highlighting the execution phase (analysis and refactoring) and validation phase (testing and mutation evaluation). 15

4.1 Mutation score before and after refactoring for the JFreeChart runs 25
4.2 Targeted mutants by PIT mutation operator for the JFreeChart runs 28
4.3 Killed and surviving targeted mutants for the JFreeChart runs, with surviving mutants grouped by PIT mutation operator 29
4.4 Mutation score before and after refactoring for the Bukkit runs 36
4.5 Targeted mutants by PIT mutation operator for the Bukkit runs 42
4.6 Killed and surviving targeted mutants for the Bukkit runs, with surviving mutants grouped by PIT mutation operator 43

Chapter 1

Introduction

In the industry, the term legacy code is often used to refer to code that is hard to modify because developers do not fully understand it [34]. Legacy code can arise because software systems are changed over time. Lehman argues that when software is evolving, the complexity of code inevitably increases over time unless specific work is performed to maintain or reduce this complexity [52]. This corresponds to what Fowler also states, namely that the design of the program will decay without refactoring [35]. Because the cost of this complexity often appears later, it is usually postponed or ignored [53].

This is most commonly known as technical debt [27, 76]. This reduces evolvability, which Rowe and Leaney define as "a system's ability to withstand changes in its requirements, environment and implementation technologies" [66]. Technical debt increases the risk of breaking the system when making changes to the codebase. Well-implemented changes are very important, because when software changes are slow or unreliable, businesses will lose opportunities [10]. For example, these could be market opportunities, like responding to new customer needs or competitive features, or regulatory and compliance opportunities, like entering or operating in regulated markets.

There are already tools to make the code more analyzable to increase the developer's understanding of the code, which includes Schuberg Philis' Code Insights that uses artificial intelligence to get these insights, and thus promote the evolvability of the system. But next to only understanding the code, it is also important to preserve the current behavior when making changes to make sure the system can evolve safely. When developers introduce bugs to the system, users will stop trusting them [34]. Changes to the code are risky in general because developers cannot predict the effects. Tests reduce this uncertainty by making allowed behavior of the code explicit and checkable [34]. Thus, automatic test generation tools were developed like EvoSuite [36] and Randoop [64] to make it easier for developers to test their code.

However, test generation techniques often aim to maximize coverage, but prior work suggests that this may not necessarily lead to effective test suites [46]. A technique to measure the effectiveness of a test set is mutation testing [49]. In mutation testing, we change small parts of the source code that mimic typical mistakes developers make, which are called mutants. With this, programmers can check if the tests can find the introduced fault. If the test suite is effective, then at least one of the tests will fail, and thus the mutant

is killed.

Zhu et al. [84] have introduced a new way to interpret mutation scores by analyzing them through the lens of testability and observability. They observed a correlation between observability metrics and the mutation score, and simple refactoring changes can improve the mutation score. There are studies done to do refactoring automatically, to reduce the manual effort for developers by automatically identifying design problems and proposing refactoring operations [8]. And because of the significant advancement in Large Language Model (LLM) technology, there has been an increase in studies examining the use of LLMs for refactoring. The strengths of LLMs are that they are trained on a massive amount of publicly available data, which includes open-source code. This allows them to handle large amounts of contextual information and recognize patterns. LLMs can outperform developers in code refactoring in improving code quality [25] and can generate correct and less complex programs while reducing the average cyclomatic complexity [72, 19]. This shows that the use of LLMs for refactoring looks promising.

However, prior work of LLM-generated refactorings mainly focused on structural code quality improvements, like reducing code smells and reducing the average cyclomatic complexity, while refactoring with an LLM from a testing perspective is still mainly unexplored. More precisely, whether LLM-generated refactorings guided by mutation testing can improve code observability, a subset of testability [84]. Testability is one of the subcharacteristics of evolvability [13], thus increasing observability can potentially have a positive impact on the evolvability of a system.

In this thesis, we explore from a testing perspective how observability-targeted refactoring of the source code using LLMs, guided by surviving mutants, affects evolvability. The idea is that when existing behavior becomes easier to observe and verify, tests can better protect this behavior during future changes. This supports evolvability because changes can be made with more confidence that existing behavior is still preserved, whether those changes are made by developers or by an LLM. Since the refactoring is performed by an LLM, the approach could reduce the manual effort needed to apply such improvements across larger legacy systems.

1.1 Research Questions

Our main research question is: **To what extent can refactoring using LLMs, guided by mutation testing results, increase the observability of production code, while not decreasing the readability of the code?** We also take the readability into account, because increasing the observability of production code should not come at the cost of making production code harder to understand. A refactoring that would make behavior easier to test, but make the code less readable, would shift the problem of trying to improve the code.
Our main research question can be broken down into two sub-questions:

RQ1 To what extent can refactoring using LLMs, guided by mutation testing results, increase the observability of production code?

RQ2 To what extent does refactoring using LLMs, guided by mutation testing results, impact the readability of production code?

To answer these research questions, we perform a case study on two open-source Java projects, JFreeChart and Bukkit. For each project, six classes are selected based on the number of surviving mutants in the baseline mutation testing report. This results in twelve runs in total. In each run, an LLM agent uses the mutation testing results to target surviving mutants, refactor the production code, validate the changes by running the existing tests, add tests for the targeted behavior, and finally run mutation testing again. We evaluate the effect on observability using the mutation score, refactoring strategies, targeted mutant outcomes, and a qualitative assessment of the generated tests. For the effect on readability, we use readability-related metrics and a qualitative analysis of the applied refactorings.

1.2 Schuberg Philis and Lab271

Schuberg Philis is a mission-critical IT company located in Schiphol-Rijk, with additional key offices in Rotterdam and, more recently, Eindhoven. They ranked first in the Whitelane Research 2025 IT Sourcing Study across multiple key categories such as General Satisfaction, Cloud & Infrastructure Services, and Security Services [79]. Their vision is to accelerate business progress while making a positive impact by prioritizing innovation and quality [70], with 100% customer satisfaction.

Mission-critical refers to work that has a direct impact on the customer's strategic business goals. If these systems that are developed fail or underperform, the core activities of the organization will be severely disrupted. This can lead to major operational, financial, or reputational consequences. Thus, they focus on the activities that are essential to the core business of their customers. They design and architect, implement, operate, and secure the systems, and work on the strategic questions behind the IT request. These systems must be available, reliable, and secure all the time. This is because, for example, when a bank's payment processing fails, the impact is catastrophic. Their customers include Nederlandse Spoorwegen (NS), PostNL, the Port of Rotterdam, ING, and Koninklijke Luchtvaart Maatschappij (KLM).

Innovation is vital for business. Specifically, innovation means the tangible growth or process that builds up one's competitive advantage [68]. Thus, within Schuberg Philis, Lab271 functions as the innovation and experimentation environment, which is connected to this mission-critical focus. While the customer teams are responsible for the "plan, build, run" in production environments, Lab271 focuses on innovation, where they explore and test new technologies, architectures, and ways of working that could strengthen or transform customers' mission-critical IT landscapes. Lab271 consists of different labs [69]. This includes:

- Frontier Tech Lab: Focusing on stakeholders gaining perspective on new technologies, such as quantum computing or robotics
- Adaptable Architecture Lab: Focusing on bridging the gap between concepts and implementation

- Strategic Design Lab: Focusing on closing the gap between business ambition and technical reality
- Artificial Intelligence Lab: Exploring how AI reshapes how we build IT, and not only what we are building. They dissect AI agents, multi-agent architectures, spec-driven development, and patterns of human-AI collaboration. Their focus is on making AI operational, trustworthy, and effective in real environments.

Our thesis falls into this last lab, being the Artificial Intelligence Lab. This is because, with the rise of AI, the way in which we use AI changes. New technologies are slowly rising that support the AI models, such as the Model Context Protocol, which was released by Anthropic on the 25th of November 2024 [5], and the rise of parallel multi-agent orchestration which Anthropic used to build a C compiler on the 5th of February 2026 [17]. Our thesis can support Lab271 by giving more insight into the potentials and limitations of using LLMs and to what extent we can trust them, specifically for legacy code, since some of their customers' code bases are also legacy code. To make future changes to legacy code safer, LLMs can be used to refactor the code so that existing behavior can be tested and preserved. However, not all old behavior is testable. Therefore, we want to look for observability gaps and fix them to promote evolvability in the future.

1.3 Thesis Outline

The structure of the thesis is as follows. First, we discuss the background needed to understand the concepts used in this thesis in Chapter 2. Following this, we discuss our methodology in Chapter 3, which includes the setup of our experiments and the choices made for this approach. In Chapter 4, we present the results based on this methodology. We answer the research question, connect the findings to practice, and discuss threats to validity in Chapter 5. In Chapter 6, we discuss the related work. Finally, we finish with the conclusions and future work in Chapter 7.

Chapter 2

Background

In this chapter, we will discuss the relevant concepts for this thesis. We first explain mutation testing, followed by testability, observability, and readability. We then explain the concept of refactoring, and finally explain LLM agents and the relevant development tools used in this study.

2.1 Mutation Testing

The focus of test generation techniques is often to maximize coverage, but prior work suggests that this may not necessarily lead to effective test suites [46]. A technique to measure the effectiveness of a test suite is mutation testing, which was proposed in the 1970s and has been widely and progressively studied since then [49]. In mutation testing, small parts of the source code are changed to mimic typical mistakes developers make. These changed versions are called mutants. This could be, for example, a `Math` mutation operator changing a `+` into a `-`, or a `True Returns` mutation operator that always returns `true` for a method that returns a boolean. An example of a `Math` mutation operator can be seen in Listing 2.1. Since this study uses PIT [24], a tool to apply mutation testing on a Java project, Table 2.1 gives an overview of the default mutation operators used by the tool. By applying these mutations, programmers can check if the tests can find the introduced fault. If the test suite is effective, then at least one of the tests will fail, and the mutant is killed. The test in Listing 2.1 survives, since it cannot differentiate the output of the `add` function with this introduced mutant. In this example, we can change the assertion to `add(3,2)`, which should return 5. With the mutated behavior, the output will be 1, and the test will fail. Thus, the mutant is killed. We can see that a surviving mutant indicates that the test suite did not distinguish the original program from the mutated version. This can indicate that the tests are too weak or that the effect of the mutation is difficult to observe. For this thesis, surviving mutants are especially relevant because they can indicate behavior that was executed but not sufficiently observable by the test suite [84].

There are also no-coverage mutants. These are mutants that were not killed because no test exercised the mutated line of code. This is related to code coverage, since a test suite can only potentially detect a mutant if it executes the line that was mutated [46]. Next to

```java
// Original
int add(int a, int b) {
    return a + b; // 0 + 0 = 0
}

// Mutant
int add(int a, int b) {
    return a - b; // 0 - 0 = 0
}

// Test
@Test
public void add() {
    assertEquals(0, add(0, 0));
}
```

Listing 2.1: Simple mutation-testing example where replacing addition with subtraction is not detected because the test input of 0 for both values produces the same result for both versions.

```java
// Original
int i = 2;
if (i >= 1) {
    return "foo";
}

// Equivalent mutant
int i = 2;
if (i > 1) {
    return "foo";
}
```

Listing 2.2: Example of an equivalent mutant, adapted from the PIT basic concepts documentation [43].

this, there also exist mutants that behave in the exact same way as the original code, which are called equivalent mutants [3, 49]. An example from the site of `PIT/PITest` [24] is shown in Listing 2.2. Here, a `Conditionals Boundary` mutation operator was applied, and there is no difference in output when >= is changed into >.

The mutation score is usually used to measure test-suite effectiveness. This is calculated as the ratio of the number of killed mutants over the total number of mutants. The formula can be seen in Equation 2.1, where u denotes the analysed unit, such as a method, class, or project. In this thesis, the total number of mutants in this calculation refers to the mutants that were either killed or survived.

Table 2.1: PIT default mutation operators used in this study [44]

Mutation Operator	Description	Example
Conditionals Boundary	Replaces relational operators with their boundary counterpart.	`i <= 10` → `i < 10`
Increments	Replaces increments with decrements, and vice versa, for local variables.	`i++` → `i--`
Invert Negatives	Inverts negation of integer and floating-point variables.	`return -x;` → `return x;`
Math	Replaces arithmetic operators with another arithmetic operator.	`a + b` → `a - b`
Negate Conditionals	Negates conditional expressions.	`x == null` → `x != null`
Void Method Calls	Removes calls to void methods.	`list.clear();` → removed
Empty Returns	Replaces return values with an empty value for the return type.	`return list;` → `return Collections.emptyList();`
False Returns	Replaces primitive or boxed boolean return values with `false`.	`return isValid;` → `return false;`
True Returns	Replaces primitive or boxed boolean return values with `true`.	`return isValid;` → `return true;`
Null Returns	Replaces object return values with `null`.	`return object;` → `return null;`
Primitive Returns	Replaces primitive numeric return values with `0`.	`return count;` → `return 0;`
Remove Conditionals (EQUAL_ELSE)	Forces the else branch for equality-based conditionals.	`if (x == y) { A } else { B }` → execute `B`
Experimental Switch	Replaces the default switch label with another label, and replaces other labels with the default.	`default` label ↔ another switch label

$$\text{mutation score}(u) = \frac{\#\text{killed mutants in } u}{\#\text{total mutants in } u} \tag{2.1}$$

2.2 Testability, Observability, and Readability

Testability describes how easy it is to verify through tests that software still behaves correctly after changes [47], and is a subcharacteristic of evolvability [13]. In this thesis, following the perspective of Zhu et al. [84], observability is an important part of testability. Observability describes whether an internal change in the program becomes visible from the outside, usually through the program's output [73] [3, p. 14]. Whalen et al. [78] describe an expression as observable in a test case if changing the value of that expression also changes the output that the test can check. If the test cannot observe any output change, then the expression is not observable in that test case. This is important for mutation testing because a

mutant not only has to be executed by a test, but its effect also has to propagate to something the test checks.

This is connected to weakly killed and strongly killed mutants. Before explaining weakly killed and strongly killed mutants, it is important to know the RIP model, which stands for Reachability, Infection, and Propagation [3, p. 13] [60]. Reachability means that execution reaches the faulty or mutated program location. Infection means that executing this location causes an incorrect program state compared to the original program. Propagation means that this incorrect state affects the program output. Mirian-Hosseinabadi [60] formally defines strongly killing mutants and weakly killing mutants as:

Strongly killing mutants. Given a mutant $m \in M$ for a program P, and a test t, t is said to strongly kill m if and only if the output of t on P is different from the output of t on m.

Weakly killing mutants. Given a mutant $m \in M$ that modifies a location L in a program P, and a test t, t is said to weakly kill m if and only if the state of the execution of P on t is different from the state of the execution of m on t immediately after L.

In other words, weakly killing mutants satisfy the reachability and infection conditions, but not the propagation condition. Strongly killing mutants satisfy all three conditions.

Lastly, readability is defined by Buse and Weimer as human judgement of how easy a text is to understand, and is related to software quality [16].

2.3 Refactoring

Refactoring is a technique to reorganize the code to improve its design, while keeping its original behavior intact. In general, it is important to refactor the code, because as Fowler describes it: "Without refactoring, the design of software will decay. Regular refactoring helps code retain its shape. As people change code to achieve short-term goals, often without a full comprehension of the architecture, the code loses its structure." [35] Decay is the opposite of evolution [61], which means instead of progressive improvement, the changes will lead to a system that is harder to understand, and thus also harder to maintain or extend. Although refactoring is usually discussed as a technique to improve the design of the code, it can also affect testability. For example, extracting logic into a separate method or exposing state through a controlled access can make behavior easier to verify in tests.

In Figure 2.1, refactoring is shown as the blue transition from implementation I_n to implementation I_{n+1}. Both implementations should still satisfy the same specification S_n. Therefore, the existing tests T_n should still pass after the refactoring, which is shown as an arrow from T_n to I_{n+1}.

In this thesis, refactoring is relevant because the structure of the code can affect how easy it is to test the behavior of that code. Refactoring can, for example, extract logic into a separate method or expose the relevant state through controlled access, such as getters. In that case, the intended behavior of the program does not change, but the behavior can become easier to observe in tests. This also means that the test suite can be extended from

T_n to T_{n+1}, not because the specification has changed, but because the refactored implementation makes more behavior checkable. This arrow is also highlighted blue.

The dotted arrows in Figure 2.1 show a possible later evolution step. After refactoring, the program may be easier to change because more behavior is covered by tests and because the implementation is easier to understand. Thus, when continuing developments on the system, the specification may change from S_n to S_{n+1}. The implementation and tests then also have to evolve, from I_{n+1} to I_{n+2} and from T_{n+1} to T_{n+2}. This later step is not the refactoring step studied in this thesis, but it illustrates why observability and readability are relevant for evolvability. It bridges the gap between I_n to I_{n+2}.

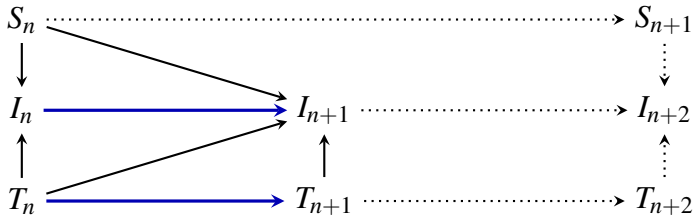

Figure 2.1: Conceptual view of LLM-guided refactoring and later software evolution. The solid horizontal blue arrows show the refactoring step studied in this thesis, where I_n changes to I_{n+1} and T_n can be extended to T_{n+1}, while the specification remains S_n. The dotted arrows show a possible later evolution step.

2.4 LLM Agents and Development Tools

A Large Language Model (LLM) is a model trained on large amounts of text, which can include source code. Because of this, LLMs can be used for software engineering tasks such as explaining code, generating tests, and proposing code changes. In this thesis, the LLM is used through an agentic development tool, which is a tool that follows the idea of LLM agents as systems that can perceive their environment, reason about goals, and execute actions [81, 57]. An LLM agent not only generates text, but can interact with a development environment by reading files, proposing changes, applying edits, and using external tools. Cline [23] is such a development tool and provides modes for planning and applying code changes. The Model Context Protocol (MCP) provides a way to expose external tools and data sources to an LLM-based agent. In this thesis, we created a custom MCP server that makes PIT mutation-testing information available to the agent.

Chapter 3

Methodology

In this chapter, the methodology of our research will be described. The goal of the case study is to answer the main research question: **To what extent can refactoring using LLMs, guided by mutation testing results, increase the observability of production code, while not decreasing the readability of the code?** We will investigate the following sub-questions:

RQ1 To what extent can refactoring using LLMs, guided by mutation testing results, increase the observability of production code?

RQ2 To what extent does refactoring using LLMs, guided by mutation testing results, impact the readability of production code?

To answer these research questions, we first describe the main components of our proposed system. We then describe the case study setup, including the selected projects, the workflow followed during each refactoring run, and how the outcomes are evaluated for observability and readability.

3.1 Approach

In this section, we describe the architecture of our proposed approach. The architecture consists of the LLM agent with an input and output, and it uses several tools. These include PIT/PITest [24], a custom MCP server, and a test runner. An overview is shown in Figure 3.1. The following subsections describe the main components, namely the LLM agent, PIT/PITest, and the MCP server.

3.1.1 LLM Agent

The LLM `Claude-4.5-Sonnet` was used during the study between March and April 2026. This model was selected because it is a recent model, released on September 29 of 2025, and because of its widespread adoption in industry environments, including at Schuberg

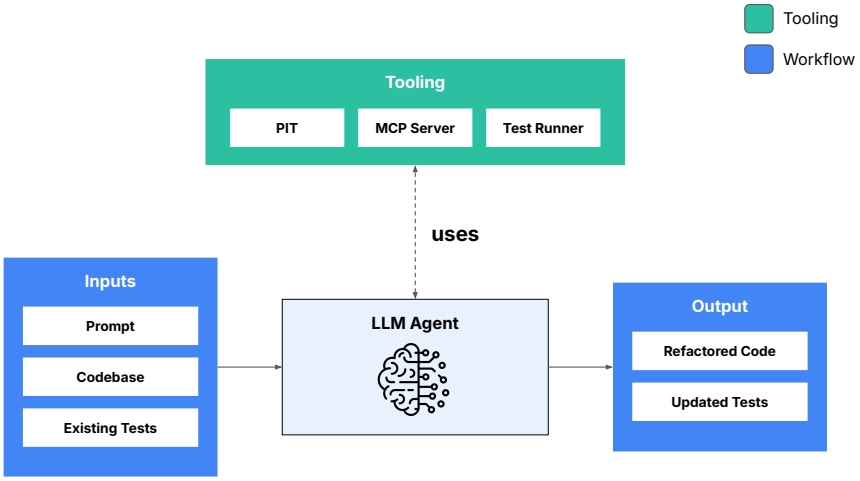

Figure 3.1: Architecture of the proposed mutation-based LLM refactoring system.

Philis. To ensure consistency between the twelve refactoring runs, a single model was used throughout the study. This allows us to fairly compare the results.

Cline [23] was used to allow the LLM to execute actions on the codebase, in the terminal, and to make use of the MCP server. Through Cline, the LLM can read and modify source code, execute commands to run the tests and PIT, and query the MCP server for the mutation testing results. This enables a sequential process in which the LLM analyses mutation results, performs refactorings, and validates the changes. Cline was installed through the Visual Studio Code Extension Marketplace[1]. Cline supports multiple authorization options. In this study, we used a company-provided API key to access and select `Claude-4.5-Sonnet`. However, access to this model does not depend on having a key provided by a company. Cline also allows users to access supported models through the Cline Provider sign-in flow or by using personal provider credentials through Bring Your Own Key [21].

The LLM was used to analyse the mutation testing results from PIT [24], decide where to refactor in a class by targeting specific mutants, provide a refactoring of the code, and generate new tests to make use of the new observable behavior. A guiding prompt was used to define the execution workflow, which will be discussed in Section 3.2.2.

3.1.2 PIT/PITest

PIT/PITest [24] version 1.21.1 was used to apply mutation testing in the study. The default set of mutation operators was used because these are designed to be stable. This means that the mutants are not too easy to detect, and minimize the number of equivalent mutants gen-

[1] https://marketplace.visualstudio.com/items?itemName=saoudrizwan.claude-dev

erated [44]. The default set consists of `Conditionals Boundary`, `Increments`, `Invert Negatives`, `Math`, `Negate Conditionals`, `Void Method Calls`, `Empty Returns`, `False Returns`, `True Returns`, `Null Returns`, `Primitive Returns`, `Remove Conditionals` for the `EQUAL_ELSE` case, and `Experimental Switch` mutation operators [44].

3.1.3 Model Context Protocol Server

An MCP server was created to ensure efficient retrieval of the mutation testing results. Alternatives include attaching the PIT output as text to a prompt. While this is easy to do and can be optimized by using a script to extract the relevant survived mutants for a specific part of the code, it is very manual and token-heavy. Another alternative is a script generating a .json file, with the user prompting the LLM agent to read it. This method is very straightforward to implement, and the files can be saved for reference. However, we need to ensure that when we run mutation testing, we regenerate the file, and that the agent reads the correct file. Another downside is that extending functionality is less flexible, because adding new queries requires modifying the script and regenerating the data.

With an MCP server, the LLM agent retrieves the mutation testing results on demand for the latest available PIT report, can be extended easily with new tools, and retrieves scoped results. Scoped means that it is able to get mutants for class X for method Y, for example. However, the downside is that there is more engineering needed to create the server.

We developed the MCP server in Python. We use the MCP Python SDK [6], which uses `anyio` [42] for asynchronous communication and I/O. This is because an MCP server must read incoming messages, process tool calls, write responses, and do I/O with files. We published the repository of our MCP server on Zenodo [31] with instructions on how to make it work in Cline.

The MCP server consists of five tools:

- `ping` to test the connectivity
- `pit_find_latest_xml` to find the latest .xml file under the `<workspace>/target/pit-reports` folder, which always points to the most up-to-date PIT report
- `pit_classes` to return the mutation scores and the number of survived, killed, and no coverage mutants per class
- `pit_methods` to return the mutation scores and the number of survived, killed, and no coverage mutants per method of a targeted class
- `pit_survivors_for_method` to return the survived mutants with their mutator (e.g., MathMutator), description (e.g., Replaced integer addition with subtraction), and line number of a targeted method

The server exposes these tools through the MCP, which allows the LLM agent to invoke them with specific arguments and receive the scoped and structured results. This reduces unnecessary context and supports more focused refactoring decisions.

3.2 Case Study Setup

This section describes how the approach is evaluated through a case study. First, we describe the selected open-source projects and how the target classes were chosen. After that, we describe the workflow followed during each refactoring run, including the pre-execution setup, prompt design, execution phase, and validation phase. Finally, we describe how the outcomes of the runs are evaluated for observability and readability.

3.2.1 Open-Source Projects

Two open-source Java projects were selected for this study, namely JFreeChart [48] and Bukkit [15]. The projects with their specific release versions and their main characteristics are summarized in Table 3.1. This includes the lines of code (LOC) of Java source files measured using `cloc` [28], the number of tests (#Tests) counted using `@Test`, the total number of mutants (#Mutants), the `NO_COVERAGE` mutants (#No Coverage), the killed mutants (#Killed), and the mutation score as a percentage (MS (%)). Both are hosted on GitHub and were chosen based on three criteria. The first criterion is their prior use in mutation testing research [46, 84]. The second criterion is their compatibility with the case study setup, specifically with PIT [24], and thus use the Maven build system. The last criterion is the presence of a comprehensive test suite. This is required because observability issues can only be identified when tests are able to detect behavioral differences.

The project selection started with a list of open-source projects, which are used in four mutation testing studies [46, 84, 75, 82]. From these papers, there is a total of 18 candidate projects collected and checked in order. The full 18 candidate projects can be found in Appendix A. In the first step, projects that did not use Maven were removed because PIT requires Maven to run in our environment. In the second step, the projects were checked by trying to compile them, run their tests, and run PIT. Based on this process, JFreeChart and Bukkit were selected for this case study. There are several other projects that are also compatible with the setup, but the study was limited to two projects, because this allows us a more detailed analysis of refactoring runs.

For both projects, the `pom.xml` files were modified to make the projects compatible with PIT. A `tools` folder was also added at the root of each project and contains the scripts used for running tests and mutation testing. These modified `pom.xml` files and scripts are included in our replication package [32]. This replication package also includes the prompts used and links to the modified project repositories and the MCP server.

Table 3.1: Overview of selected open-source projects

PID	Project	LOC	#Tests	#Mutants			MS (%)
				Total	#No Coverage	#Killed	
1	JFreeChart-1.5.6	139 679	2 361	36 676	16 246	12 291	60.2%
2	Bukkit-1.7.9-R0.2	32 378	280	7 432	6 227	950	78.8%
	Total	172 057	2 641	44 108	22 473	13 241	61.2%

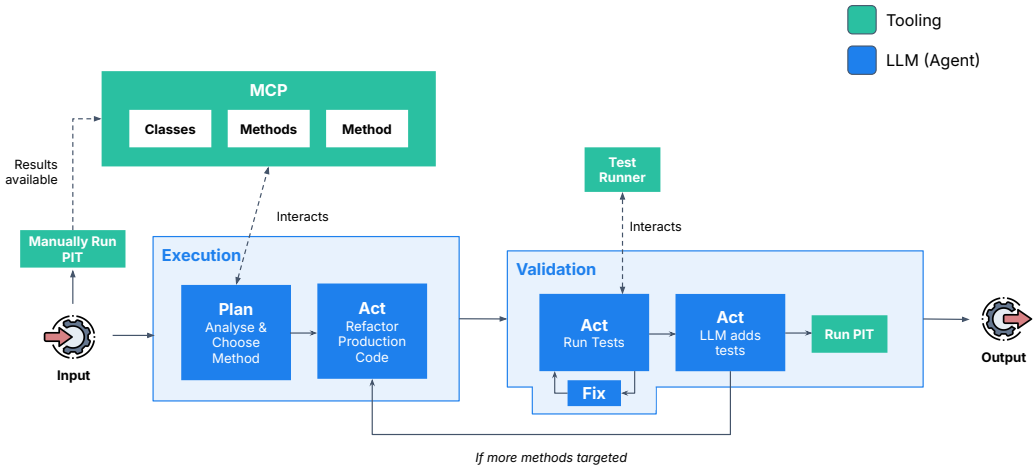

Figure 3.2: Overview of the refactoring run workflow, highlighting the execution phase (analysis and refactoring) and validation phase (testing and mutation evaluation).

3.2.2 Workflow for Each Refactoring Run

Figure 3.2 provides an overview of the refactoring run workflow. The two main phases are the execution and the validation. Before these phases, the process starts with three inputs: the prompt, the codebase, and the existing test suite. The breakdown of the inputs can be seen in the architecture in Figure 3.1.

For each project, six refactoring runs were performed, resulting in twelve runs in total. The target classes were selected based on the number of surviving mutants in the baseline PIT report. The first run targeted the class with the highest number of surviving mutants, the second run targeted the class with the second-highest number, and this continued until six classes were selected. Each run focused on one selected class. This scope allowed us to do a detailed analysis per run, including refactoring targets, generated tests, and readability impact. Furthermore, each run was started with the same initial prompt, after which the agent carried out the execution and validation phases within that same run automatically. The full workflow consists of the pre-execution setup, prompt design, execution phase, and validation phase, which we will describe in that order.

Pre-Execution

Mutation testing using PIT [24] is run once before the execution phase to obtain a baseline mutation report. This report can then be reused throughout the runs, instead of having to run PIT again before each run. For the MCP server to retrieve this report, the baseline report must be stored under the project's `target/pit-reports` directory, where the MCP tool

searches for the latest PIT report. During execution, the LLM agent can access the mutation testing results through the MCP server, allowing the LLM to analyse and guide refactoring decisions. The codebase and the test suite are both used as input for the workflow and for mutation testing. A single prompt is used to guide the behavior of the LLM across both the execution and validation phases.

Prompt Design

The prompt defines the objective, guides decision-making, controls the agent's behavior, and specifies the required actions. This single prompt is used to initiate the full workflow, after which the LLM agent carries out the analysis, refactoring, validation, and test generation steps automatically. The prompt shown in Listing 3.1 was used for the first run, which targeted the class with the most surviving mutants. For the following runs, the same prompt was used, but the target rank was adjusted so that the second run targeted the class with the second-most surviving mutants, the third run targeted the class with the third-most surviving mutants, and so on. In this study, distinguishing correct from incorrect behavior means that a test passes for the original program behavior, but fails when the targeted mutant changes that behavior. The prompt instructs the LLM agent to follow a structured approach: 1. decide which mutant(s) to target; 2. decide which refactoring technique to apply; 3. justify how the refactoring increases observability and enables new assertions.

In addition, the prompt specifies execution constraints for the agent. The agent is instructed to only use the scripts in the tools folder for running tests and PIT, and to analyse mutation results only through the MCP server. This avoids unnecessary terminal output or whole CSV files of PIT from being included in the model's context. Otherwise, this would increase token usage significantly without contributing relevant information when the tests or mutation testing succeed, or when retrieving all the mutant details of all the classes. The scripts in the tools folder can be found in our replication package [32]. The final constraint is that PIT is executed only once at the end of the run.

Lastly, the prompt instructs the LLM agent to perform the refactoring and add tests that expose previously unobservable behavior.

The prompt was initially inspired by the paper by Alomar et al. [2]. The paper suggests that a structured prompt can reduce the number of interaction turns. In our runs, we used a single-pass execution to maintain a consistent setup across runs. Therefore, we use a structured, task-focused prompt that includes only the components that are most relevant to the refactoring goal and its execution constraints. In addition, the prompt incorporates Chain-of-Thought prompting, where the LLM is instructed to make intermediate reasoning steps explicit before producing a solution. This has been shown to improve performance on complex reasoning tasks [77].

```
Refactor the class with the most survived mutants to increase observability of
     internally mutated behavior indicated by surviving mutants, enabling
    tests to distinguish correct from incorrect behavior.

Surviving mutants may indicate insufficient observability when mutated
    behavior occurs internally but cannot be observed or asserted on by tests,
     leaving the test suite unable to distinguish behavioral differences.
    After refactoring, add new tests that make the previously unobservable
    behavior observable: directly test the new observable behavior introduced
    by the refactoring with focused inputs that distinguish correct behavior
    from the mutated alternatives indicated by the surviving mutants.

Before each refactoring step, briefly state: (a) which surviving mutant(s) (by
     location/description) you are targeting, (b) what refactoring technique
    you will apply, and (c) why this increases observability and what new
    assertion it enables.

To run PIT and the tests, please use the scripts provided in the tools folder.
     To analyse the mutants, you can use only the tools provided in the MCP
    server. PIT was already run beforehand once, so it is not needed to run it
     again to be able to retrieve the baseline results. Do not run PIT after
    each change. Only run PIT once at the end of the iteration to evaluate the
     effect of the refactoring + new tests. You may run unit tests iteratively
     as needed to check correctness while developing.
```

Listing 3.1: Prompt used to guide the LLM in mutation-based refactoring and test generation.

Execution

The execution phase consists of two sub-phases. In the first sub-phase, the LLM agent analyses mutation testing results using the MCP server. Cline is set to Plan mode, since this mode in Cline is used by the agent to understand and plan, without focusing yet on the implementation details [22]. The agent requests the MCP server in this order:

1. `pit_classes` to find the class with the most surviving mutants (or the second-highest, third-highest, etc.)
2. `pit_methods` to get an overview of the methods with their mutants for a specific class
3. `pit_survivors_for_method` (can be called multiple times) to analyse the mutants in detail for a specific method

After constructing a plan, the agent is switched to Act mode, where it implements the planned refactorings on the production code.

Validation

After the agent has applied the refactorings to the production code, the validation phase starts automatically. The validation phase consists of three sub-phases. As described before in Section 3.2.2, the LLM uses scripts to avoid unnecessary terminal output for running the test runner and PIT. After the refactoring, the agent runs the existing test suite using the provided test runner script. Since the test runner invokes Maven, this step checks whether the project still compiles and whether the existing tests pass, which helps ensure that the original behavior is preserved.

If the test command fails, either due to compilation errors or failing test cases, the agent continues under the same initial prompt by using the command output to identify the cause, modifying the code or tests, and rerunning the test command until the test suite completes successfully, all without additional manual intervention needed. After the test suite passes, the LLM adds new tests with the goal of making use of the new observable behavior. If the agent selected multiple refactoring targets, it can return to the production-code refactoring step after validation and repeat the validation step for the next target. Finally, PIT is executed again by the agent to obtain the updated mutation testing results, which are used for data analysis.

3.2.3 How We Evaluate Observability

We evaluate the outcomes of the runs for observability based on the mutation score and mutant outcomes, the refactoring technique used and its effectiveness, and the generated tests. Each of these is discussed separately below.

Mutation Score and Mutant Outcomes

To assess observability, the mutation score is used as an indicator of whether the refactoring increased the observability. A low mutation score can indicate low behavioral observability, because surviving mutants can suggest that behavioral differences cannot be detected by tests [84]. Thus, an increase in mutation score would be expected when the observability is increased. However, an increase can also be caused by the generated tests that test behavior that was already observable before the refactoring. The mutation score is calculated by Equation 3.1, where u denotes the analysed unit, such as a method, class, or project. Furthermore, equivalent mutants are not manually removed from this calculation. We only use the mutation score as an indicator for observability, rather than a standalone measure, and manually checking each mutant can be time-consuming. Therefore, all generated mutants are treated as non-equivalent. This is a common method when the mutation score is used as a relative comparison [83].

$$\text{mutation score}(u) = \frac{\#\text{killed mutants in } u}{\#\text{total mutants in } u} \tag{3.1}$$

We also analyse the number of killed and surviving mutants before and after each run. This is because the mutation score gives only a relative measure that makes runs easier to

compare, but the number of killed and survived mutants shows the absolute size of change. The number of killed mutants does not always have to be equal to the decrease in surviving mutants, because newly killed mutants can also come from refactorings that introduce mutants or from mutants that previously had no coverage.

Refactoring Technique and Effectiveness

In addition to the mutation score, we analyse what kind of refactoring technique was used and whether it is useful for increasing observability. For this, we analyse the targeted mutants, the reasoning of the LLM, and whether these are successfully killed. These are the initially surviving mutants that the LLM selected during its Plan mode in Cline, but also targeted during its Act mode. This is because sometimes the LLM planned a lot of different targets initially, but during its Act mode only targeted a subset. It is clear what the LLM actually targeted during the Act mode, because at the end of a run it will show a task summary. In here, the LLM states again what mutants it targeted, and how it targeted those. During this analysis, we also manually checked which mutation operator was applied to each targeted mutant. If these targeted mutants are not killed, we also analyse the reasons for their survival.

Lastly, we also assess if there are any potential behavioral changes introduced. This is because a change in behavior can change the mutation score's outcome. This happened in one of our pilot runs. There were fewer killed mutants and one more surviving mutant than expected. Apparently, by changing the behavior of the code in one line that changed the branching, some mutants after that line changed from initially killed or survived to having no coverage, since the tests could not reach certain branches anymore. As seen in Listing 3.2, the condition in the if statement changed from `isEmpty()` to `shouldSkipDrawing()`, thus changing "only skip if area is empty" to "skip if area is empty or too small to draw", and as a consequence, some tests return prematurely when the area is too small but not empty.

```
// Before
dataArea = integerise(dataArea);
if (dataArea.isEmpty()) {
    return;
}

// After
dataArea = integerise(dataArea);
if (shouldSkipDrawing(dataArea)) {
    return;
}
```

Listing 3.2: Example of replacing an empty-area check with a helper method that also checks whether the drawing area is too small.

Generated Tests

The generated tests are measured both quantitatively and qualitatively, since tests are a prerequisite for the mutation score. For each selected class, we record the test lines of code, number of tests, and number of assertions before the run, and the corresponding additions made by the LLM. The line coverage before and after the run is also reported. These measurements provide context for the mutation score changes, because killing mutants depends not only on refactoring production code, but also on whether the generated tests execute and assert the relevant behavior.

We also observe the tests qualitatively to try to make sure that the LLM does not cheat the mutation score. By cheating, we mean when the LLM tries to increase the mutation score without using the newly introduced refactorings to increase observability, or makes the test only to kill one specific mutant, which may not be useful in a broader context. An example would be a test that only excludes a known mutant result, such as `assertNotEquals(-1, add(2, 3))`, rather than asserting the expected behavior directly with `assertEquals(5, add(2, 3))`.

To make the analysis more consistent across runs and guide us through the process, we follow a specific set of questions. This includes whether the tests are using behaviorally meaningful inputs, if the tests would be worth keeping outside the sole purpose of mutation testing, and whether the assertions make use of the newly observable behavior after the refactoring.

The mutation results, refactoring technique with the targeted mutant outcomes, and generated tests are interpreted together to determine whether the improvement came from newly enabled observability or from tests for behaviour that was already observable.

3.2.4 How We Evaluate Readability

To assess readability, we use two perspectives. We use quantitative source-code metrics at both class level and method level to get an indication of changes in complexity and size. However, they do not fully determine whether the code is easier or harder to understand. This is why we use a second perspective to complement the metrics using a qualitative method from the perspective of the researchers. We will discuss the quantitative and qualitative methods below in more detail.

Quantitative

For the metrics, we mainly look at complexity to get an indication of readability, since studies show a negative correlation between complexity and readability [1], and reducing if nesting and loop structures are among the most frequent suggestions for increasing code readability [71]. Furthermore, complexity metrics, including McCabe Cyclomatic Complexity (MCC) and nesting, tend to decrease when improving readability [33]. Lastly, results indicate that minimizing nesting improves readability [50].

Based on this, we chose metrics that are available in CK [4], a tool on GitHub to calculate class-level and method-level code metrics in Java using static analysis. At class level,

we use the weighted methods per class (WMC), maximum nesting depth, lines of code, number of methods, number of variables, number of loops, and coupling between objects (CBO). At method level, we measure WMC, maximum nesting depth, lines of code, number of parameters, number of variables, and number of loops.

After we retrieve the CK results in `csv` format, we use scripts we created ourselves in Python to extract specific metrics and their corresponding results. These scripts are also available in our replication package [32].

Qualitative

The qualitative analysis complements the metrics by evaluating the refactoring from the perspective of the researchers. For each run, the refactoring is evaluated using three criteria: the local clarity of the targeted method, the intent, and additional code structure. Local clarity considers whether the logic of the targeted method became easier to understand. Intent considers whether the class better expresses what it does. Additional code structure considers whether the refactoring added extra methods or a testing-oriented structure that makes the class heavier to understand. We use a readability scale of improved, neutral, and worsened. These criteria are based on common readability principles from clean code and refactoring literature. These emphasize simple and direct code, focused responsibilities, and code that communicates its purpose [58, 35].

The readability metrics are compared before and after each run and interpreted together with the qualitative criteria.

Chapter 4

Results

In this chapter, we present the results of the case study conducted on the selected projects JFreeChart and Bukkit, as described in Chapter 3. The results are organized per project, followed by a summary of the findings that addresses the research questions.

Each project is divided into two parts: observability and readability. The observability results are discussed in five steps. We first focus on the mutation score and the number of survived and killed mutants to show the overall impact of the LLM-guided refactorings and generated tests. Second, to understand why the mutation score increased, the next part analyses the applied refactoring techniques. Third, the refactoring targets and initial observability issues are discussed to explain why these parts of the code were difficult to test before refactoring. Fourth, the targeted mutant results are analysed to determine if the refactorings applied were effective. In other words, if the LLM was able to kill the initial targeted surviving mutants. Lastly, since tests are a prerequisite for killing mutants, we analyse the generated tests.

The readability results are discussed in two steps. In the first step, we analyse the impact on readability quantitatively using metrics. In the second step, the refactorings are analysed qualitatively to interpret how they affected the readability of the code. This is because metrics can only give an indication of possible readability changes, but do not show whether the code became easier or harder to understand. Thus, we complemented this analysis with a qualitative assessment.

4.1 JFreeChart

This section reports the results for JFreeChart, divided into observability and readability. As discussed in Chapter 3, we ran the study on the six classes with the highest number of initial surviving mutants. Table 4.1 provides an overview of these classes, including their size in lines of code (LOC) and the number of initially surviving mutants.

4.1.1 Observability

This subsection follows the observability structure described in the results introduction for the JFreeChart runs, starting with the mutation score results, then analysing the refactorings,

Table 4.1: Overview of selected JFreeChart classes and initial mutation characteristics

ID	Class	LOC	Initial Surviving Mutants
1	org.jfree.chart.plot.XYPlot	2703	354
2	org.jfree.chart.plot.CategoryPlot	2420	349
3	org.jfree.chart.plot.PiePlot	1617	321
4	org.jfree.chart.plot.MeterPlot	644	196
5	org.jfree.chart.plot.PiePlot3D	693	177
6	org.jfree.chart.axis.CategoryAxis	754	175

targeted mutants, the effectiveness of the refactorings, and generated tests.

Mutation Score

After applying the LLM-guided refactorings and generated tests, the mutation score increased across all JFreeChart runs, as shown in Figure 4.1. As seen in Table 4.2, the average increase in mutation score across the classes is 5.96%. The mutation score shows the relative improvement and makes the results easier to compare across classes. However, because it is a percentage, it does not show how big the size is of the underlying change. For example, killing 70 out of 100 mutants gives the same mutation score as killing 7 out of 10 mutants, but the number of killed mutants is different. Next to this, the increase in killed mutants does not have to be equal to the decrease in surviving mutants, since newly killed mutants can also come from newly introduced mutants because of the refactoring (e.g., adding new methods) or from mutants that previously had no coverage. Thus, both killed and surviving mutants are reported to give more context of the impact of the refactoring and the generated tests. The table shows that there is an average increase of 26 killed mutants and a decrease of 15 surviving mutants.

Table 4.2: Mutation score (MS) changes and mutant outcome differences for the JFreeChart runs

ID	Class	MS Before (%)	MS After (%)	Δ MS (%)	Δ Killed	Δ Survived
1	XYPlot	49.86%	52.44%	2.58%	24	-13
2	CategoryPlot	54.68%	55.97%	1.30%	24	1
3	PiePlot	36.81%	43.56%	6.75%	43	-23
4	MeterPlot	25.76%	36.60%	10.85%	29	-28
5	PiePlot3D	10.15%	15.58%	5.43%	11	-9
6	CategoryAxis	27.69%	36.55%	8.86%	24	-17
Average		**34.16%**	**40.12%**	**5.96%**	**25.83**	**-14.83**

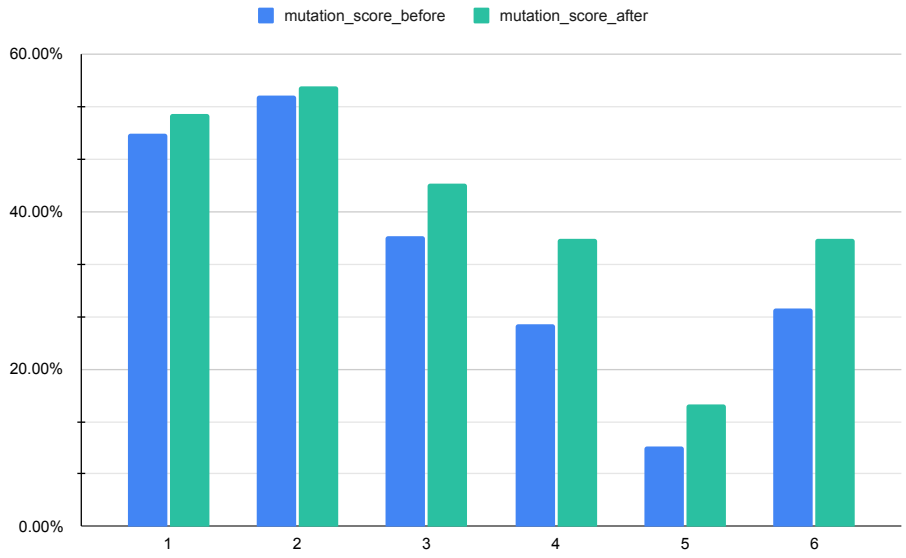

Figure 4.1: Mutation score before and after refactoring for the JFreeChart runs

Refactoring Strategies

To understand why the mutation score increased, we first analyse the applied refactoring techniques. The LLM applied different refactoring techniques, which can be seen in Table 4.3. The table also reports the number of methods targeted (# Methods). The refactoring strategies in the table were determined by looking at both the summary produced by the LLM after the run was completed and the actual code changes. In some cases, this directly corresponds to a common refactoring operation, such as Extract Method or changing the visibility of a method. In other cases, the label describes the larger observability strategy that was applied. For example, the Observable State Pattern consists of multiple changes, including adding a private field, exposing it through a getter, and recording operations or state changes during execution.

In two of the six refactoring runs, the LLM uses an Observable State Pattern, in which a new private variable is introduced with a corresponding getter. This allows the tests to access additional information. For example, in `XYPlot`, specific operations are recorded and later checked in the tests. This can be seen in Listing 4.1 for one of the operations. The important detail in this example is that the recording call is separate from the actual drawing call. Other examples of operations are `setPlotArea()` and `drawBackground()`. In `CategoryPlot`, the existing object `CategoryCrosshairState` is made directly accessible. This is similar to changing the return type from `void` to `crosshairState`, because in both cases we can access the state. However, the main difference is that storing the `lastCrosshairState`, instead of returning the state immediately, keeps the original method signature unchanged. In this context, it is needed because `draw()` overrides a void method.

Because Observable State Pattern is not a standard refactoring technique like Extract Method, we compare it with the closest existing refactoring technique. This helps clarify which part of the change is similar to an existing refactoring and where it differs. The closest refactoring technique to Observable State Pattern is Encapsulate Variable [35]. In Encapsulate Variable, one makes a field private and provides controlled access via getter and setter methods. With this, changes to the field can be handled in one place without affecting the rest of the code. In Observable State Pattern, however, we create a new private field, and its main purpose is to increase observability.

In one run, the LLM did change the return type from void to an object. This corresponds to one of the simple refactorings in the study of Zhu et al. [84] that enable killing mutants that were previously not killed. Another refactoring that was mentioned in their study is visibility change, which means changing `private` to `protected` to enable direct tests. This was used by the LLM in run 5. Lastly, in half of the runs, the LLM used Extract Method [35] to isolate specific parts of the code to make them directly testable.

```
private DrawingOperations lastDrawingOperations;

public DrawingOperations getLastDrawingOperations() {
    return this.lastDrawingOperations;
}

if (domainAxisState != null) {
    // actual drawing call
    drawDomainGridlines(g2, dataArea, domainAxisState.getTicks());
    // recording call
    this.lastDrawingOperations.recordDomainGridlinesDrawn();
    drawZeroDomainBaseline(g2, dataArea);
}
```

Listing 4.1: Example of Observable State Pattern, where the recording call is separate from the actual drawing call (XYPlot).

Refactoring Targets and Observability Issues

After analysing the applied refactoring techniques, we analyse the targets to understand which kinds of code the LLM attempted to make more observable. The refactoring targets vary across the runs as seen in Table 4.3, and the corresponding targeted mutation operators are shown in Figure 4.2. These targets can be grouped into a small number of categories. By grouping these similar targets, it makes it easier to see which JFreeChart runs had similar observability problems, and also makes the later comparison with Bukkit easier. In the first group, the LLM targets state-related logic, which includes void method calls and state-setting operations (e.g., `crosshairState.setRowKey(getDomainCrosshairRowKey())`). This was mainly observed in the first three runs, and this is also reflected in Figure 4.2, where these runs mainly target `Void Method Calls` mutants. In the second group, the

Table 4.3: Overview of refactoring strategies and targets for JFreeChart runs

ID	Class	# Methods	Refactoring Strategy	Refactoring Target
1	XYPlot	1	Observable State Pattern	Void calls and state-setting operations in `draw()`
2	CategoryPlot	1	Observable State Pattern	Surviving mutants in crosshair-related logic
3	PiePlot	1	Change return type	`drawPie` method
4	MeterPlot	1	Extract method	Tick position calculations
5	PiePlot3D	3	Visibility change + Extract Method	Angle detection and paint logic
6	CategoryAxis	1	Extract Method	Coordinate calculations

LLM targets calculation logic, as seen in runs 4 and 6, where tick position and coordinate calculations are extracted into separate methods. These runs mainly target `Math` mutants. Lastly, we have the LLM targeting decision logic, such as angle classification and conditional checks, as seen in run 5 and some mutants in run 2. This corresponds mainly to `Negate Conditionals`, `Conditionals Boundary`, and `True Returns` mutants.

The categories correspond to different kinds of observability problems. In some cases, removing void method calls cannot be checked directly by tests. In other cases, calculations are harder to test because they are mixed with graphical code. For decision logic, we observed two situations. In some cases, the condition itself is accessible, but its effect is not observable. This is, for example, when it leads to a void drawing call. An example of such a mutant, located in `CategoryPlot`, is highlighted in Listing 4.2. Because `drawDomainCrosshair()` is a void method, which is unobservable, we cannot observe the effect if `isDomainCrosshairVisible()` is negated. A similar surviving mutant existed in the same class, where an update of a state object is not accessible to the tests. In the other cases, the logic is implemented in a private method, which cannot be directly accessed by tests.

```
if (isDomainCrosshairVisible() && columnKey != null) {
    Paint paint = getDomainCrosshairPaint();
    Stroke stroke = getDomainCrosshairStroke();
    drawDomainCrosshair(g2, dataArea, this.orientation,
            datasetIndex, rowKey, columnKey, stroke, paint);
}
```

Listing 4.2: Example where the surviving condition mutant is difficult to detect because its effect is only observable through the void drawing call `drawDomainCrosshair()`.

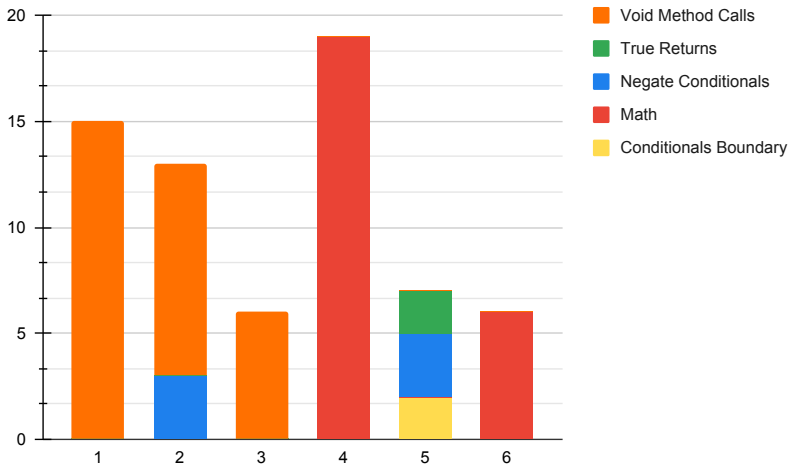

Figure 4.2: Targeted mutants by PIT mutation operator for the JFreeChart runs

Targeted Mutant Effectiveness

The next step is to analyse whether the LLM-guided refactorings and generated tests were effective in killing the initially targeted surviving mutants. As one can see in Figure 4.3 and Table 4.4, combined with the refactoring strategies in Table 4.3, only in the runs where the Observable State Pattern refactoring was used, the LLM did not succeed in killing all the mutants that were targeted. In Figure 4.3, killed mutants are shown in grey, while the remaining surviving mutants are grouped by PIT mutation operator. As one can see, mainly Void Method Calls survived in `XYPlot`, and both Void Method Calls and Negate Conditionals survived in `CategoryPlot`. Earlier, Listing 4.1 was used to show the refactoring strategy applied by the LLM in `XYPlot`. Here, the same listing is used to explain why the targeted mutant still survived. The tests were able to kill the mutants in the conditionals located before the targeted drawing calls. This is because changing the condition prevents the expected operation from being recorded. In this example, if `domainAxisState != null` is changed to `domainAxisState == null`, the branch is not executed, and thus the test fails since the drawing operation is not recorded. However, the mutant that removes `drawDomainGridlines()` is not detected, because `recordDomainGridlinesDrawn()` is still called afterwards. This means the test only checks that something was recorded, not that the actual drawing call was executed. This is why all `Void Method Calls` mutants survived.

In `CategoryPlot`, the LLM was not able to kill all the targeted mutants. As stated before, the refactoring technique is very similar to returning `crosshairState` directly. However, several generated assertions used this now-observable state weakly. An example is in `testRangeCrosshairStateWithLockedOnData`, where a mutant can break the crosshair computation and return a NaN, while the test still passes. Another example is where tests would only check that a state object exists, but not whether the state object is correct, as in `testCrosshairStateObservability`. These examples are shown in Listing 4.3.

```
// In testCrosshairStateObservability()
CategoryCrosshairState state = plot.getLastCrosshairState();
assertNotNull(state);

// In testRangeCrosshairStateWithLockedOnData()
double crosshairY = state.getCrosshairY();
assertTrue(crosshairY == 15.0 || crosshairY == 25.0 ||
           Double.isNaN(crosshairY),
           "Crosshair Y should be a data value or NaN: " + crosshairY);
```

Listing 4.3: Example of a weak assertion on the exposed crosshair state (`CategoryPlot`).

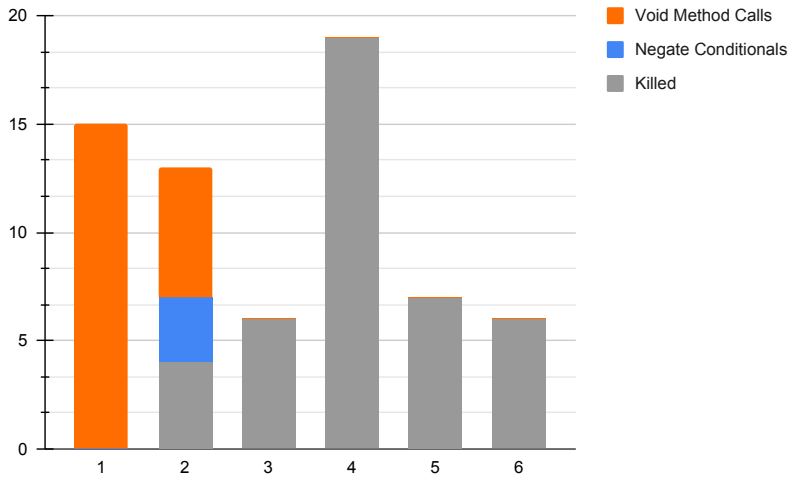

Figure 4.3: Killed and surviving targeted mutants for the JFreeChart runs, with surviving mutants grouped by PIT mutation operator

Table 4.4: Targeted mutant outcomes for the JFreeChart runs

ID	Class	# Targeted	# Still Surviving	# Killed	Killed (%)
1	XYPlot	15	15	0	0.00
2	CategoryPlot	13	9	4	30.77
3	PiePlot	6	0	6	100.00
4	MeterPlot	19	0	19	100.00
5	PiePlot3D	7	0	7	100.00
6	CategoryAxis	6	0	6	100.00

Generated Tests Analysis

Tests are a prerequisite for killing mutants, because a mutant is only killed when at least one test fails after the mutation is applied. This is why we also analyse the generated tests. The analysis first reports the size of the generated test suites and the line coverage changes to give more context. Afterward, we will discuss the main qualitative observations about the generated tests.

The size of the existing test suite before the refactoring and the amount of test code added by the LLM can be seen in Table 4.5. The table reports the test LOC, number of tests, and number of assertions before refactoring, and the corresponding additions after refactoring. This shows whether the generated tests were small or large compared with the existing test suite. The existing tests had an average LOC of 24.03 per test, and 6.29 assertions per test. The generated tests had an average LOC of 19.53 per test, and 4.22 assertions per test.

Table 4.6 shows the line coverage before and after refactoring. Structural coverage is reported to give additional context to the mutation score. It shows whether the generated tests increased the amount of code that was executed, while the mutation score shows whether the tests were able to kill more mutants.

Table 4.5: Overview of test suite size before and after generated tests for JFreeChart

		Test LOC		Tests		Assertions	
ID	Class	Before	Added	Before	Added	Before	Added
1	XYPlot	992	144	45	6	241	18
2	CategoryPlot	903	89	38	6	220	13
3	PiePlot	470	142	16	7	142	25
4	MeterPlot	143	115	4	5	49	35
5	PiePlot3D	50	101	3	6	8	36
6	CategoryAxis	109	112	5	6	38	25
Average		**444.50**	**117.17**	**18.50**	**6.00**	**116.33**	**25.33**

However, test size and line coverage do not show whether the generated tests checked the intended behaviour. This is the reason why a qualitative analysis was also conducted, and four main observations can be identified. Firstly, some tests mainly verify that an exposed state changed or that an operation was recorded. Thus, these tests only check that something happened, but do not necessarily verify whether the behavior is correct. This was mainly notable for `XYPlot`, and an example is shown in Listing 4.4. Here, it only verifies that the background drawing operation was recorded, but not whether it is correctly drawn. We will discuss this deeper in the follow-up analysis in Section 4.3, whether we can also verify its behavior.

Secondly, some tests check returned values or calculations. These tests verify the outcome of a computation and can be considered stronger, since they directly test the expected

Table 4.6: Line coverage before and after generated tests for JFreeChart

ID	Class	Before	After	Δ
1	XYPlot	73.52%	73.70%	0.18
2	CategoryPlot	79.39%	82.25%	2.86
3	PiePlot	73.31%	75.69%	2.38
4	MeterPlot	88.83%	88.94%	0.11
5	PiePlot3D	51.37%	51.76%	0.39
6	CategoryAxis	82.00%	84.05%	2.05
Average		**74.74%**	**76.07%**	**1.33**

behavior. This pattern is mainly observed in several other runs. An example is shown in Listing 4.5, where the test checks whether the computed bounds (i.e., x_0, x_1, y_0, and y_1) match the expected values.

Thirdly, the test suites have a mix of tests that use boundaries as assertions and tests where expected values were derived directly from the implementation. These tests can only be strong when one is fully sure that the original implementation is right and will not change in the future. As we can see in Listing 4.5, the expected values are computed using the same method calls and calculations as in the original implementation, such as `getCategoryStart()`.

Lastly, some generated tests became weaker after fixing a failing assertion. In one case in `PiePlot3D`, the LLM first used incorrect boundary expectations for angle checks, but then changed the test into an assertion that an angle cannot be both at the front and at the back. This assertion always holds for the implementation, because the two helper methods check opposite strict inequalities. Therefore, the final test passed, but it no longer meaningfully tested the original boundary behaviour.

Test-Based Observability Enhancement (Unexpected Run)

There is one case where the LLM did not fully follow its initial plan. This was in the case of `XYPlot`, and thus the run was repeated. This was considered an unexpected result, as the LLM did not make any changes to the original source code. This run is still useful to discuss, because it shows that a higher mutation score does not always mean that the production code became more observable.

The LLM first planned to create a `DrawingOperations` class to track which drawing operations were performed, as it later did in the following run that is recorded in our results. After that, it planned to add getter methods or make fields accessible to make `PlotRenderingInfo` and `CrosshairState` more observable. However, during Act mode, it found that these fields already had getters, and it immediately wrote tests for them. After this, it ended the iteration.

This means that the initial plan was reasonable, because the LLM correctly identified that these objects needed to be more observable. However, no source-code change was needed, since the observability was already present in the existing code. Therefore, in terms

```
@Test
public void testDrawRecordsBackgroundDrawn() {
    XYSeriesCollection dataset = new XYSeriesCollection();
    XYSeries series = new XYSeries("Series 1");
    series.add(1.0, 2.0);
    dataset.addSeries(series);

    NumberAxis xAxis = new NumberAxis("X");
    NumberAxis yAxis = new NumberAxis("Y");
    XYItemRenderer renderer = new XYLineAndShapeRenderer();
    XYPlot plot = new XYPlot(dataset, xAxis, yAxis, renderer);

    BufferedImage image = new BufferedImage(400, 300, BufferedImage.
    TYPE_INT_RGB);
    Graphics2D g2 = image.createGraphics();
    Rectangle2D area = new Rectangle2D.Double(0, 0, 400, 300);

    plot.draw(g2, area, null, null, null);

    DrawingOperations ops = plot.getLastDrawingOperations();
    assertNotNull(ops);
    assertTrue(ops.wasBackgroundDrawn(),
            "Background should have been drawn");

    g2.dispose();
}
```

Listing 4.4: Example test verifying that a drawing operation was recorded (`XYPlot`).

of production code, the observability stayed the same. The mutation score still increased from 49.86% to 51.88%, with 35 more killed mutants, because the LLM added new tests that used the existing getters to detect mutants that were not killed before.

4.1.2 Readability

This subsection discusses the impact of the JFreeChart refactorings on code readability. We first present the quantitative metric changes at class level, and then at method level. Lastly, we also give a qualitative assessment of how the refactorings affected the code in terms of readability.

Quantitative

The impact on readability at the class level is negligible, which we can see in Table 4.7. The differences in cyclomatic complexity (Δ WMC), maximum nesting depth (Δ Nesting), lines of code (Δ LOC), number of methods (Δ Methods), number of variables (Δ Vars), number of loops (Δ Loops), and coupling between objects (Δ CBO) are minimal. There is a correlation

```
@Test
public void testCalculateCategoryLabelBoundsBottom() {
    CategoryAxis axis = new CategoryAxis("Test");
    axis.setCategoryLabelPositionOffset(10);

    Rectangle2D dataArea = new Rectangle2D.Double(
            100.0, 200.0, 400.0, 300.0);
    AxisState state = new AxisState(250.0);
    state.setMax(50.0);

    CategoryLabelBounds bounds = axis.calculateCategoryLabelBounds(
            0, 3, dataArea, RectangleEdge.BOTTOM, state);

    double expectedX0 = axis.getCategoryStart(0, 3, dataArea,
            RectangleEdge.BOTTOM);
    double expectedX1 = axis.getCategoryEnd(0, 3, dataArea,
            RectangleEdge.BOTTOM);
    double expectedY0 = 250.0 + 10.0;
    double expectedY1 = 260.0 + 50.0;

    assertEquals(expectedX0, bounds.getX0(), 0.0001);
    assertEquals(expectedX1, bounds.getX1(), 0.0001);
    assertEquals(expectedY0, bounds.getY0(), 0.0001);
    assertEquals(expectedY1, bounds.getY1(), 0.0001);
}
```

Listing 4.5: Example test verifying computed output values, with implementation-derived expectations (`CategoryAxis`).

Table 4.7: Changes in readability-related metrics for JFreeChart runs

ID	Class	Δ WMC	Δ Nesting	Δ LOC	Δ Methods	Δ Vars	Δ Loops	Δ CBO
1	XYPlot	1	0	15	1	1	0	1
2	CategoryPlot	1	0	5	1	1	0	0
3	PiePlot	0	0	1	0	0	0	0
4	MeterPlot	1	0	3	1	2	0	1
5	PiePlot3D	1	0	4	1	0	0	0
6	CategoryAxis	1	0	4	1	1	0	1

between cyclomatic complexity and the number of methods, because when a new method is added, the cyclomatic complexity increases by at least one. This follows from the nature of cyclomatic complexity, since every method has a minimum cyclomatic complexity of 1.

At the method level, there are some notable changes, as seen in Table 4.8. This table reports, for each targeted method, the changes in cyclomatic complexity (Δ WMC), maximum nesting depth (Δ Nesting), lines of code (Δ LOC), number of parameters (Δ Params),

number of variables (Δ Vars), and number of loops (Δ Loops). The number after the slash in the method name indicates the number of parameters, which helps distinguish overloaded methods in the results.

In particular, we can see a notable decrease in lines of code and the number of variables for `drawTick/4` and `drawCategoryLabels/6`. If we connect this to the refactoring used in Table 4.3, we can see Extract Method was used. There is also a small decrease for `drawSide/10` on the cyclomatic complexity and lines of code, where Extract Method was also applied.

Table 4.8: Method-level readability-related metric changes for the targeted JFreeChart methods

Class	Target Method	Δ WMC	Δ Nesting	Δ LOC	Δ Params	Δ Vars	Δ Loops
XYPlot	draw/5	0	0	11	0	0	0
CategoryPlot	draw/5	0	0	1	0	0	0
PiePlot	drawPie/3	0	0	1	0	0	0
MeterPlot	drawTick/4	0	0	-9	0	-7	0
PiePlot3D	drawSide/10	-1	0	-2	0	0	0
PiePlot3D	isAngleAtBack/1	0	0	0	0	0	0
PiePlot3D	isAngleAtFront/1	0	0	0	0	0	0
CategoryAxis	drawCategoryLabels/6	-4	0	-27	0	-3	0

Qualitative

In the qualitative case studies, we saw that most changes are small and thus readability is mostly unchanged. For example, in runs 2, 3, and 5 (changing visibility), there is almost no impact.

Extract Method is the only refactoring that we judged as consistently improving readability. This can be seen in runs 4, 5, and 6. This is because a part of the method, such as a calculation, is moved out. Consequently, the remaining method becomes shorter and easier to follow. In addition, the code expresses more clearly what it does, since the LLM chooses a clear name for the extracted part of the method. For example, in `MeterPlot`, `calculateTickEndpoints()` communicates its purpose instead of leaving it implicit in a long sequence of calculations without any comments. The structure in this case remains neutral because, while a helper method is introduced, it is limited and contained.

Only in run 1 is the structure worsened, since the method now mixes two responsibilities, which are drawing and the recording of operations.

4.2 Bukkit

Using the same structure as we used for JFreeChart, this section reports the results for Bukkit, and is divided into observability and readability for the six selected Bukkit classes. Table 4.9 provides an overview of the selected classes.

Table 4.9: Overview of selected Bukkit classes and initial mutation characteristics

ID	Class	LOC	Initial Surviving Mutants
7	org.bukkit.plugin.messaging.StandardMessenger	365	25
8	org.bukkit.plugin.messaging.PluginMessageListenerRegistration	70	21
9	org.bukkit.configuration.MemorySection	617	20
10	org.bukkit.command.SimpleCommandMap	220	15
11	org.bukkit.command.Command	198	12
12	org.bukkit.configuration.file.FileConfiguration	97	11

4.2.1 Observability

This subsection follows the observability structure described in the results section for the Bukkit runs. We first present the mutation score results to show the overall effect of the LLM-guided refactorings and generated tests, followed by an analysis of the refactoring strategies, refactoring targets, targeted mutant outcomes, and generated tests.

Mutation Score

The mutation scores across all runs have increased, as seen in Figure 4.4. Table 4.10 shows that the average increase is 47.89%, with an average increase of 24 killed mutants and a decrease of 9 surviving mutants. However, we can see that runs 10 and 11 started with a mutation score of 0.00%. Thus, this played a big part in the overall average increase. These runs are still part of the top six classes because the ranking is based on the number of surviving mutants, not on the initial mutation score. As in the JFreeChart results, the killed and surviving mutant counts are both included to show the absolute changes behind the mutation score.

Table 4.10: Mutation score (MS) changes and mutant outcome differences for the Bukkit runs

ID	Class	MS Before (%)	MS After (%)	Δ MS (%)	Δ Killed	Δ Survived
7	StandardMessenger	75.00%	84.07%	9.07%	20	-7
8	PluginMsgListenerReg	43.24%	85.19%	41.94%	30	-13
9	MemorySection	90.34%	93.91%	3.57%	29	-6
10	SimpleCommandMap	0.00%	80.65%	80.65%	25	-9
11	Command	0.00%	100.00%	100.00%	22	-12
12	FileConfiguration	31.25%	83.33%	52.08%	15	-7
Average		**39.97%**	**87.86%**	**47.89%**	**23.50**	**-9.00**

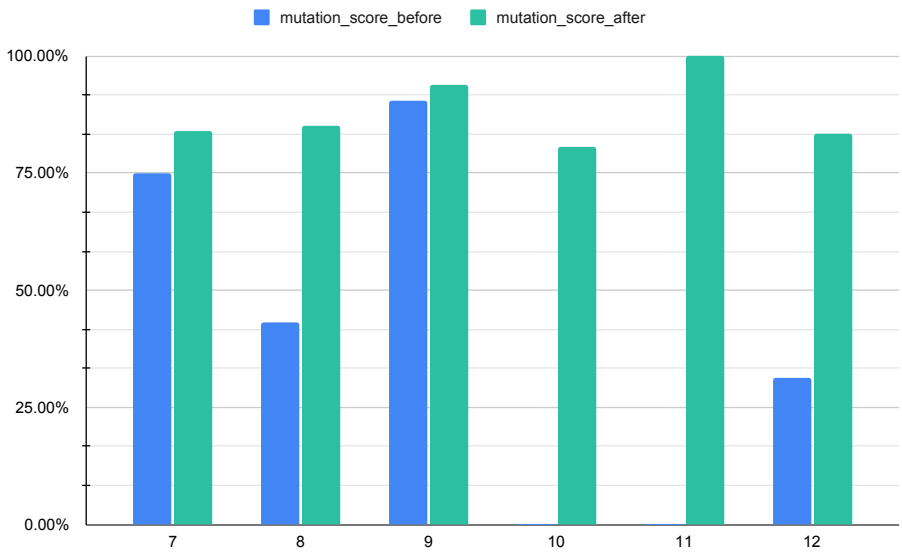

Figure 4.4: Mutation score before and after refactoring for the Bukkit runs

Refactoring Strategies

To understand the mutation score increase for Bukkit, we analyse the applied refactoring techniques. Table 4.11 provides an overview of which refactoring techniques the LLM used. The LLM used a variety of refactorings and sometimes used multiple strategies in one class. Some of these strategies are known refactorings, such as Extract Method [35]. Others are not standard refactoring names, but describe changes where the LLM added getter or query methods to expose internal state for testing.

In two cases, which are runs 7 and 10, the LLM used a query-type refactoring. This includes, for example, checking whether a channel still has an entry in the outgoing hashmap (in the class `StandardMessenger`), or checking whether a command is registered under a specific label in the internal command map (in the class `SimpleCommandMap`). Examples of query methods can be found in Listing 4.6 with the methods `hasOutgoingChannelMapEntry()` and `isCommandRegistered()`. The query-type strategies are closest to accessor-based refactorings, such as Encapsulate Variable [35], because they provide a controlled access to an otherwise hidden state. However, in this study, the added methods are used mainly to make selected internal state observable to tests. All the other refactoring techniques are used only once and will be described from top to bottom.

In the Add Registration Count Methods technique, the LLM introduced three count methods that expose selected registration counts from `StandardMessenger`. These three methods can be seen in Listing 4.7. The counts show how many registrations exist for a specific channel or plugin, and with this, the tests can inspect registration changes more directly. For context, Bukkit plugins can send and receive plugin messages through named channels. A simplified example of this internal registration state is shown in Listing 4.8.

Table 4.11: Overview of refactoring strategies and targets for Bukkit runs

ID	Class	# Methods	Refactoring Strategy	Refactoring Target
7	StandardMessenger	3	Add Map State Query Methods	Mutants in `removeFromOutgoing` (map state)
			Add Registration Count Methods	Mutants in `getIncomingChannelRegistrations` (distinguish from `emptySet`)
			Strengthened Validation Testing	Mutants that can be killed via invalid inputs
8	PluginMsgListenerReg	2	Extract Method	Hashing logic and internal comparison
9	MemorySection	14	Extract Type Validation Methods	Negated `instanceof` conditionals
			Extract Validation State Methods	Null/empty validation checks
10	SimpleCommandMap	4	Add Query Methods (state)	Constructor, return value, and conditional mutants
11	Command	5	Add Getter Methods	State-dependent registration and update behavior
12	FileConfiguration	5	Template Method Pattern	Validation calls, file I/O operations, and encoding logic (void methods)

Channel names in this example are "chat" and "trade". `StandardMessenger` keeps track of which plugins are registered for which channels.

The example shows that the class does not store all registrations in one collection. Instead, registrations are separated by direction, incoming or outgoing, and by lookup direction, channel or plugin. Therefore, one count method would not be sufficient, because a generated test may need to count registrations for a channel in one case, but registrations for a plugin in another case. The LLM added specifically three count methods because the generated tests needed to inspect a few specific registration counts that were not directly accessible before. Two of these methods count incoming registrations, either for a channel or for a specific plugin and channel. The third method counts outgoing channels for a plugin, which helps test whether registrations are cleaned up after unregistration.

Strengthened Validation Testing is not a refactoring technique by definition, because refactoring changes the internal structure of production code [35], while in Strengthened

```
// StandardMessenger
private final Map<String, Set<Plugin>> outgoingByChannel =
    new HashMap<String, Set<Plugin>>();

[...]

boolean hasOutgoingChannelMapEntry(String channel) {
    synchronized (outgoingLock) {
        return outgoingByChannel.containsKey(channel);
    }
}

// SimpleCommandMap
protected final Map<String, Command> knownCommands =
    new HashMap<String, Command>();

[...]

public boolean isCommandRegistered(String label) {
    return knownCommands.containsKey(label.toLowerCase());
}
```

Listing 4.6: Examples of query-type refactorings, where the LLM added helper methods to expose internal state in StandardMessenger and SimpleCommandMap.

Validation Testing, the LLM only extends the test suite by adding extra tests. However, the LLM listed it alongside the other refactoring techniques, and with this, it specifically targeted mutants that could be killed via invalid inputs by triggering exceptions. Since this affects the mutation score, it is included in the table. Extract Method [35] was applied to two methods of the `PluginMessageListenerRegistration` class, namely `equals()` and `hashCode()`. Extract Type Validation Methods introduce helper methods that check whether a value at a given path can be converted to specific types (e.g., integer, double, or list types). Extract Validation State Methods introduce checks for common state conditions, such as whether a path or section is null or empty. Both an example of an Extract Type Validation Method and Extract Validation State Method can be seen in Listing 4.9. The Extract Type Validation Methods and Extract Validation State Methods are variants of Extract Method [35]. In both cases, the LLM extracted validation logic into separate helper methods, making the validation behavior easier to test directly. Add Getter Methods is similar to the query-based techniques, but consists of direct getter methods rather than conditional queries. An example is just a method returning `return knownCommands.size()`, where `knownCommands` is a map.

Lastly, we have the Template Method Pattern, where methods encapsulate specific void calls in the original code. For example, `Validate.notNull(obj, message)` is wrapped in a method such as `validateNotNull(Object obj, String message)`, which is then

```
private final Map<String, Set<PluginMessageListenerRegistration>>
    incomingByChannel = new HashMap<String, Set<PluginMessageListenerRegistration>>();

private final Map<Plugin, Set<PluginMessageListenerRegistration>>
    incomingByPlugin = new HashMap<Plugin, Set<PluginMessageListenerRegistration>>();

private final Map<Plugin, Set<String>> outgoingByPlugin =
    new HashMap<Plugin, Set<String>>();

[...]

int getIncomingRegistrationCount(String channel) {
    synchronized (incomingLock) {
        Set<PluginMessageListenerRegistration> registrations =
            incomingByChannel.get(channel);
        return registrations != null ? registrations.size() : 0;
    }
}

int getIncomingRegistrationCount(Plugin plugin, String channel) {
    synchronized (incomingLock) {
        Set<PluginMessageListenerRegistration> registrations =
            incomingByPlugin.get(plugin);
        if (registrations == null) {
            return 0;
        }
        int count = 0;
        for (PluginMessageListenerRegistration registration : registrations) {
            if (registration.getChannel().equals(channel)) {
                count++;
            }
        }
        return count;
    }
}

int getOutgoingChannelCountForPlugin(Plugin plugin) {
    synchronized (outgoingLock) {
        Set<String> channels = outgoingByPlugin.get(plugin);
        return channels != null ? channels.size() : 0;
    }
}
```

Listing 4.7: Examples of the Add Registration Count Methods technique, where the LLM added helper methods to expose registration counts in `StandardMessenger`.

```
incomingByChannel:
"chat" -> [listener from PluginA, listener from PluginB]

incomingByPlugin:
PluginA -> [listener for "chat"]
PluginB -> [listener for "chat"]

outgoingByChannel:
"trade" -> [PluginA]

outgoingByPlugin:
PluginA -> ["trade"]
```

Listing 4.8: Simplified example of the different registration maps in `StandardMessenger`.

```
// Extract Type Validation Method
protected boolean isValidIntType(String path) {
    Object val = get(path, null);
    return val instanceof Number;
}

// Extract Validation State Method
protected boolean isValidPath(String path) {
    return path != null;
}
```

Listing 4.9: Examples of extracted validation helper methods, where the LLM made type and path validation logic directly checkable by tests.

replaced in the exact line of the original void call. An example can be seen in Listing 4.10. This is closely related to Extract Method [35], since we extract a part of the method in a helper method. However, the idea of hooks is closely related to a hook method from the design pattern Template Method [37]. In Template Method, a parent class defines the main algorithm, but there are individual steps that can be overridden by child classes. In Template Method Pattern in our run, the test suite has a class that overrides the introduced hook methods. The main difference between the design pattern Template Method and the refactoring strategy Template Method Pattern lies in the goal. The goal of the design pattern is to reuse and structure shared behavior by fixing the order of an algorithm, while still allowing subclasses to redefine selected steps. The goal of the refactoring strategy in this study is different, because it introduces overridable steps so that tests can observe whether specific operations are executed.

Thus, as we can see, most of these refactorings try to expose internal behavior either by adding query methods, validating states explicitly, or isolating logic into separate methods (e.g., Extract Method and Template Method Pattern).

```
public void load(String file) throws FileNotFoundException, IOException,
        InvalidConfigurationException {
    Validate.notNull(file, "File cannot be null");
    validateNotNull(file, "File cannot be null");

    load(new File(file));
}

protected void validateNotNull(Object obj, String message) {
    Validate.notNull(obj, message);
}
```

Listing 4.10: Example of extracting a validation hook method, where the direct null check is replaced by a protected helper method that can be overridden in tests.

Refactoring Targets and Observability Issues

The next step is to analyse the refactoring targets to understand which parts of the Bukkit code the LLM attempted to make more observable. The overview in Table 4.11 was already shown for discussing the refactoring techniques used, but next to this, it also shows the refactoring targets. The refactoring targets can broadly be grouped into 4 different categories. This grouping makes it easier to see which Bukkit runs targeted similar kinds of code and, thus, similar kinds of observability problems.

The first category is the internal state of field mutants. This includes the internal state of maps (e.g., `outgoingByPlugin` in run 7 or `knownCommands` in run 10), or other fields such as a String (`nextLabel`) or a `CommandMap` (`commandMap`) in run 11. The second category is validation and decision logic. These are invalid inputs (run 7), `instanceof` conditionals and null/empty checks (run 9), and conditional mutants (run 10). The third category is hashing logic and internal comparison, which is only seen in run 8. Lastly, there are side-effect operations that are targeted, which can be seen in run 12 and are void calls.

The main observation is that the issue was not always a lack of observability in the production code. In some runs, the LLM did address hidden behavior, such as private map state or internal comparison logic. In other runs, the targeted behavior was already observable through the public API, but the existing tests did not exercise or assert it precisely enough. This is especially the case for validation and decision logic and side-effect operations, where mutants could often be killed by adding stronger tests branches or invalid-input cases.

The targeted mutation operators for these runs are shown in Figure 4.5. The figure gives additional context for the refactoring targets in Table 4.11. The Bukkit targets are not dominated by one mutation operator, but include a mix of `Void Method Calls`, `Negate Conditionals`, different `Returns` mutants, and `Math` mutants. For example, runs 7, 9, and 10 include several `Negate Conditionals` mutants, which correspond to validation and decision logic. Run 8 contains both `Math` and `Negate Conditionals` mutants, which correspond to the extracted hashing and comparison logic. Run 12 mainly contains `Void Method Calls` mutants, which fit the side-effect operations.

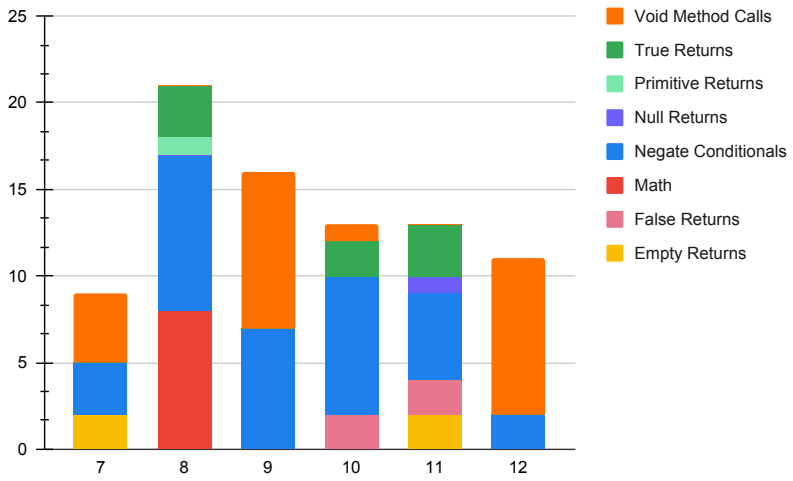

Figure 4.5: Targeted mutants by PIT mutation operator for the Bukkit runs

Targeted Mutant Effectiveness

After identifying the refactoring targets, these targets are then analysed to determine whether the LLM-guided refactorings and generated tests were effective in killing the initially targeted surviving mutants. The effectiveness of the targeted mutants can be seen in Figure 4.6 and Table 4.12. In Figure 4.6, killed mutants are shown in grey, while the remaining surviving mutants are grouped by PIT mutation operator. In every run except run 11, the LLM did not succeed in killing all of the targeted mutants. We will now go over the classes one by one to describe why some mutants still survive.

Table 4.12: Targeted mutant outcomes for the Bukkit runs

ID	Class	# Targeted	# Still Surviving	# Killed	Killed (%)
7	StandardMessenger	9	2	7	77.78
8	PluginMsgListenerReg	21	8	13	61.90
9	MemorySection	16	11	5	31.25
10	SimpleCommandMap	13	4	9	69.23
11	Command	13	0	13	100.00
12	FileConfiguration	11	4	7	63.64

The reason mutants survived in `StandardMessenger` is because the implemented refactoring did not target the mutant that the LLM stated it wanted to target in its planning phase. The surviving mutants replace an empty immutable-set return with another empty-set implementation. This corresponds to the `Empty Returns` mutation operator. Through the public API, both versions still behave as an empty set when no registrations exist. One representative example is shown in Listing 4.11, where the relevant return statement mutant is

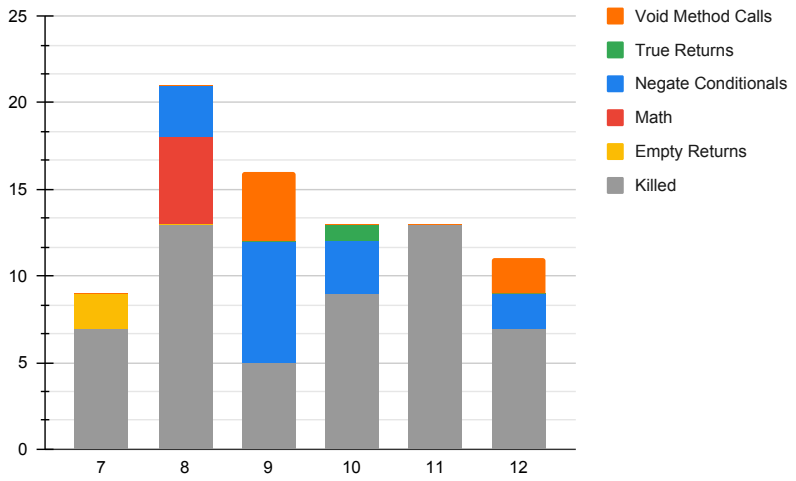

Figure 4.6: Killed and surviving targeted mutants for the Bukkit runs, with surviving mutants grouped by PIT mutation operator

highlighted in yellow. The LLM stated during its Plan mode that it wanted to kill the mutant that replaced `return ImmutableSet.of()` with `Collections.emptySet()`, and planned to do this with an assertion that the count should be bigger than 0 when registrations exist. However, this was targeting another mutant, which is a `Negate Conditionals` mutant that changed the condition of `if (registrations != null)` into `if (registrations == null)`. This condition is highlighted in red in Listing 4.11. As a side note, this count method is also redundant when we would like to kill the `Negate Conditionals` mutant, since with the public API we can also use the `.size()` method on `getIncomingChannelRegistrations()`.

In `PluginMsgListenerReg`, there are two types of mutants that still survived. The first type is a `Negate Conditionals` mutant for null checks (e.g., `this.plugin == null`) in the `equals()` method. This is highlighted in Listing 4.12 in the method `matchesMessenger()`. It is theoretically possible to add tests where fields of a registration are null. However, the constructor of the `PluginMessageListenerRegistration` does not allow one of the fields to be null. The second type are mutants that change addition into subtraction, and one mutant that changes multiplication into division. This corresponds to the `Math` mutation operator. These mutants survived because the tests did not check the exact hash code formula. These mutants are also highlighted in the `hashCode()` method in Listing 4.12.

`MemorySection` has the most surviving mutants from the Bukkit runs. There are also two types of mutants that are still surviving. The first type is what the LLM calls the Validation Observability mutants, of which the two lines are highlighted in the `MemorySection` method at the top in Listing 4.13. These `Void Method Call` mutants are very similar in the form `Validate.notNull(path, "Path cannot be null")` and survive because the tests do not test these lines directly, as they did with the 6 other Validation Observabil-

```
public Set<PluginMessageListenerRegistration>
        getIncomingChannelRegistrations(String channel) {
    validateChannel(channel);

    synchronized (incomingLock) {
        Set<PluginMessageListenerRegistration> registrations =
            incomingByChannel.get(channel);

        if (registrations != null) {
            return ImmutableSet.copyOf(registrations);
        } else {
            return ImmutableSet.of();
        }
    }
}
```

Listing 4.11: Return statements in `getIncomingChannelRegistrations()`, where the surviving mutant replaced the empty-set return with another empty-set implementation on the line that is highlighted yellow. The part that is highlighted in red is the mutant that the LLM actually targeted with its implemented refactoring strategy.

```
boolean matchesMessenger(PluginMessageListenerRegistration other) {
    if (this.messenger != other.messenger
            && (this.messenger == null
                    || !this.messenger.equals(other.messenger))) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 53 * hash + getMessengerHashContribution();
    hash = 53 * hash + getPluginHashContribution();
    hash = 53 * hash + getChannelHashContribution();
    hash = 53 * hash + getListenerHashContribution();
    return hash;
}
```

Listing 4.12: Examples of surviving mutants in `PluginMessageListenerRegistration`, where tests did not cover nullable field comparisons or the exact hash code formula.

ity mutants. The second type is called `Type Check Observability` mutants, and these mutants are also similar, which is a `Negate Conditional` mutant in the line that is highlighted in the method `getInt()` in Listing 4.13. The reason these mutants are still alive is that either the test calls the overloaded method directly like `getInt(path, def)` (or the similar-looking method like `getString()`), or because they provide a value for the path, which causes the default value to be ignored. Thus, a test only reaches this mutant when the requested value is missing from the section itself, but a numeric default value is available through `getDefault(path)`. The tests do not exercise the condition that depends on `getDefault(path)`, and the mutants remain undetected.

```
protected MemorySection(ConfigurationSection parent, String path) {
    Validate.notNull(parent, "Parent cannot be null");
    Validate.notNull(path, "Path cannot be null");

    this.path = path;
    this.parent = parent;
    this.root = parent.getRoot();

    Validate.notNull(root, "Path cannot be orphaned");

    this.fullPath = createPath(parent, path);
}

public int getInt(String path) {
    Object def = getDefault(path);
    return getInt(path, (def instanceof Number) ? toInt(def) : 0);
}
```

Listing 4.13: Examples of surviving validation and type-check mutants in `MemorySection`, where tests did not directly exercise the null checks or the default-value type check.

In `SimpleCommandMap`, the reason there are still mutants surviving is because the added tests did not observe the specific effect changed by the mutant. There are four surviving mutants, which are shown across three methods in Listing 4.14. The first surviving mutant is a `True Returns` mutant, because the tests only covered successful registrations, so the mutant forcing the method to return `true` did not change the test result. This return-value mutant is highlighted in the `register(String, Command)` method in Listing 4.14.

For the second surviving mutant, the changed condition, which is also highlighted in Listing 4.14 in the method `register(String, String, Command)`, affects whether an alias is removed from the command's own alias list. However, the added tests only checked whether the command was present in the command map. They did not check the contents of `command.getAliases()`. Therefore, the mutant could survive because the changed alias list was not observed by the tests.

The remaining surviving mutants occurred in the `register(String,Command,boolean,String)` method, also shown in the same listing. These mutants changed an earlier

condition in the method, but this did not change the final result that the tests checked. Therefore, the tests saw the same final behavior. This is related to the difference between weakly and strongly killed mutations. A mutant can already be different at an intermediate point in the method, which is the idea behind weakly killed mutation. However, for the test to kill the mutant in normal mutation testing, this difference also has to propagate to something the test observes, which is required in strongly killed mutation. In this case, the changed branch can be hidden by a later conflict check that still returns the same final value.

Lastly, in `FileConfiguration`, the reason some mutants survive is because the LLM added hook methods and mainly checked whether the hook is called, and not whether, for example, `reader.close()` is actually executed. A similar example was earlier shown in Listing 4.10, where it wraps `Validate.notNull()`.

Generated Test Analysis

Tests are also analysed because they are required to kill mutants. This part first reports the size of the generated test suites and the line coverage changes. We then discuss the main qualitative observations about the generated tests.

Table 4.13 shows the size of the existing direct tests and the amount of test code added by the LLM. Existing tests were only counted as direct tests when their assertions checked the selected class itself. Some classes did not have these direct tests, but were indirectly executed by other test classes, which are `PluginMessageListenerRegistration`, `SimpleCommandMap`, and `Command`. This is why they have zero initial test LOC, tests, and assertions in the table. Before refactoring, the existing direct Bukkit tests had an average LOC of 12.04 per test and 2.71 assertions per test. The generated tests had an average LOC of 11.06 per test and 2.41 assertions per test. In Table 4.14, one can see that the line coverage increased in all Bukkit runs, from 62.87% to 68.75% on average. The largest increases are seen in `PluginMsgListenerReg`, `SimpleCommandMap`, and `Command`. Because these classes had no direct tests before refactoring, a part of the mutation score increase could be explained by the generated tests covering behavior that was not directly tested before.

A qualitative analysis was also conducted for the generated Bukkit tests. Several tests check behavior that is meaningful, such as invalid channel names, failed command registration, null validation, cleanup behavior, and file writing. However, the mutation score increase does not always mean that the refactoring itself solved an observability problem in the original code. This can be seen in runs 9, 10, and 11. In these runs, many targeted mutants were killed by the generated tests without depending on the newly introduced refactoring methods. This was confirmed by commenting out the newly introduced methods and their respective lines in the tests that depend on them. So this suggests that the original problem was often missing test cases, rather than behavior that was impossible to observe from the original code. This also does not mean that observability did not improve at all, but it shows that the refactoring was not always necessary to kill the targeted mutants.

In run 12, the added hooks make side-effect operations easier to observe whether they were called, such as writer and reader operations. However, some tests mainly check

```
public boolean register(String fallbackPrefix, Command command) {
    return register(command.getName(), fallbackPrefix, command);
}

public boolean register(String label, String fallbackPrefix,
        Command command) {
    label = label.toLowerCase().trim();
    fallbackPrefix = fallbackPrefix.toLowerCase().trim();
    boolean registered = register(label, command, false, fallbackPrefix);

    Iterator<String> iterator = command.getAliases().iterator();
    while (iterator.hasNext()) {
        if (!register(iterator.next(), command, true, fallbackPrefix)) {
            iterator.remove();
        }
    }

    if (!registered) {
        command.setLabel(fallbackPrefix + ":" + label);
    }

    command.register(this);

    return registered;
}

private synchronized boolean register(String label, Command command,
        boolean isAlias, String fallbackPrefix) {
    knownCommands.put(fallbackPrefix + ":" + label, command);
    if ((command instanceof VanillaCommand || isAlias)
            && knownCommands.containsKey(label)) {
        return false;
    }

    boolean registered = true;

    Command conflict = knownCommands.get(label);
    if (conflict != null && conflict.getLabel().equals(label)) {
        return false;
    }

    if (!isAlias) {
        command.setLabel(label);
    }
    knownCommands.put(label, command);

    return registered;
}
```

Listing 4.14: The surviving mutants in `SimpleCommandMap` in three methods, where the changed behavior was not directly observed by the added tests.

whether the hook method was called, but do not check whether the original line (e.g., `reader.close()`) was actually executed. These correspond to the `Void Method Calls` mutation operator. This we will discuss more in-depth in Section 4.3 in the follow-up analysis.

Table 4.13: Overview of test suite size before and after generated tests for Bukkit

		Test LOC		Tests		Assertions	
ID	Class	Before	Added	Before	Added	Before	Added
7	StandardMessenger	239	175	16	16	61	55
8	PluginMsgListenerReg	0	278	0	27	0	28
9	MemorySection	424	305	42	37	99	91
10	SimpleCommandMap	0	171	0	11	0	52
11	Command	0	200	0	17	0	48
12	FileConfiguration	156	209	10	13	24	18
Average		**136.50**	**223.00**	**11.33**	**20.17**	**30.67**	**48.67**

Table 4.14: Line coverage before and after generated tests for Bukkit

ID	Class	Before	After	Δ
7	StandardMessenger	81.12%	81.75%	0.63
8	PluginMsgListenerReg	81.58%	92.59%	11.01
9	MemorySection	81.80%	82.58%	0.78
10	SimpleCommandMap	21.52%	29.59%	8.07
11	Command	28.57%	38.89%	10.32
12	FileConfiguration	82.61%	87.10%	4.49
Average		**62.87%**	**68.75%**	**5.88**

4.2.2 Readability

This subsection discusses how the Bukkit refactorings affected code readability. It first presents the quantitative metric changes for both class level and method level, and then gives a qualitative assessment of how the refactorings affected the code.

Quantitative

At class level, the readability-related metrics show noticeable changes for the Bukkit refactorings, which can be seen in Table 4.15. The largest changes can be seen in cyclomatic complexity (Δ WMC), lines of code (Δ LOC), and number of methods (Δ Methods). This is because the LLM often added helper methods like state-query methods, validation methods, or hook methods to try to make internal behavior easier to test, as we saw in Section 4.2.1.

However, both the maximum nesting depth and the coupling between objects do not increase in any run. This suggests that the refactorings did not make the control flow more deeply nested or introduce additional dependencies.

Table 4.15: Changes in readability-related metrics for Bukkit runs

ID	Class	Δ WMC	Δ Nesting	Δ LOC	Δ Methods	Δ Vars	Δ Loops	Δ CBO
7	StandardMessenger	10	0	37	5	4	1	0
8	PluginMsgListenerReg	12	0	36	8	0	0	0
9	MemorySection	15	0	40	11	7	0	0
10	SimpleCommandMap	9	0	21	4	1	1	0
11	Command	3	0	9	3	0	0	0
12	FileConfiguration	8	0	26	8	2	0	0

At method level, most targeted methods did not change. The LLM targeted 33 methods in total across all Bukkit runs, but only four targeted methods show changes. These methods are shown in Table 4.16. The table reports the changes in cyclomatic complexity (Δ WMC), maximum nesting depth (Δ Nesting), lines of code (Δ LOC), number of parameters (Δ Params), number of variables (Δ Vars), and number of loops (Δ Loops).

Table 4.16: Method-level readability-related metric changes for the affected Bukkit methods with non-zero changes

Class	Target Method	Δ WMC	Δ Nesting	Δ LOC	Δ Params	Δ Vars	Δ Loops
PluginMsgListenerReg	equals/1	-6	0	0	0	0	0
PluginMsgListenerReg	hashCode/0	-4	0	0	0	0	0
FileConfiguration	load/1	-2	0	1	0	1	0
FileConfiguration	save/1	-2	0	1	0	1	0

In `equals/1` and `hashCode/0`, the cyclomatic complexity decreases, but the other metrics stay unchanged. This corresponds to the refactoring Extract Method. In `load/1` and `save/1`, the cyclomatic complexity also decreases, but there is a small increase in lines of code and variables. For these methods, this is because a part of the method logic was moved into a helper method, while the original methods still keep the same overall structure. The reason that there is an increase in one line of code and the number of variables in `load/1` and `save/1` is because of a new `charset` variable. This can be seen in Listing 4.15.

Qualitative

The qualitative analysis provides a closer look at how the refactorings affected readability in practice. In the qualitative case studies for Bukkit, the local clarity of the targeted methods is mostly judged as neutral, because most refactorings did not change the body of the targeted method itself. In runs 7, 9, 10, and 11, the original targeted methods are not easier to understand because there is no local change to the targeted method. In these runs, helper methods were added. The methods are usually short and easy to read, but they mainly

```
Charset charset = selectCharsetForSave(); // new variable
Writer writer = new OutputStreamWriter(
    new FileOutputStream(file),
    UTF8_OVERRIDE && !UTF_BIG ? Charsets.UTF_8 :  Charset.defaultCharset()
    charset // the long line replaced with one variable
);

[...]

protected Charset selectCharsetForSave() {
    return UTF8_OVERRIDE && !UTF_BIG ? Charsets.UTF_8 :  Charset.defaultCharset();
}
```

Listing 4.15: Refactoring of inline charset selection into a helper method in `save/1`.

expose internal state or checks for testing. These classes now have a mix of functions that are for their functionality, but also specifically for their testing. Since, in most cases, these methods were not needed to test the original methods and their surviving mutants as seen before, these smaller helper methods make the class a bit heavier to read without adding meaningful new behavior.

We can mainly see an improvement in run 8, since parts of the comparison and hash-code calculation are moved into separate helper methods. This is shown in Listing 4.16. For example, a long condition in the `equals()` method becomes `!matchesMessenger(other)`, and part of the `hashCode()` calculation becomes `getMessengerHashContribution()`. This makes the intent of the code clearer because the method names describe what is being compared or calculated. However, several helper methods are added, but they remain small and focused, so the additional structure is judged as neutral.

Run 12 is mixed in terms of readability. Some extracted methods, such as the charset selection logic, make the method a bit easier to read. This can be seen in Listing 4.15, where the `UTF8_OVERRIDE` line was substituted with only `charset`. However, other methods mostly wrap existing calls, such as `Validate.notNull()`, writer operations, and reader operations. These wrappers add indirection and are mainly useful for testing side effects. Therefore, run 12 is judged as slightly improved for local clarity, because some extracted methods simplify specific expressions, but neutral for intent. This is because the added wrappers do not make the responsibility of the class clearer, but also at the same time, do not make it less clear.

4.3 Follow-up Analysis of Selected Surviving Mutants

The previous sections showed that some targeted mutants still survived after the LLM-guided refactoring and test generation. However, those sections mainly analysed the effectiveness of the LLM-generated changes and the reasons why targeted mutants survived. This follow-up analysis investigates a selected sample of remaining surviving mutants in more detail to better understand their implications. The goal is to determine whether these

```
// equals method
// Before
if (this.messenger != other.messenger
    && (this.messenger == null || this.messenger.equals(other.messenger))) {
    return false;
}

// After
if (!matchesMessenger(other)) {
    return false;
}

// hashCode method
// Before
this.messenger != null ?  this.messenger.hashCode() :  0

// After
getMessengerHashContribution()
```

Listing 4.16: Example of extracting comparison and hash-code logic into helper methods in `PluginMessageListenerRegistration`.

mutants could be killed by additional human-engineered tests, whether further refactoring would be needed, or whether the mutated behavior is difficult to distinguish meaningfully from the original behavior.

The selected cases, which are shown in Table 4.17, were chosen because they show different reasons why mutants remained alive. This is why the sample includes mutants from different classes and different mutation operators, rather than several similar mutants from the same location. The goal is not to classify all remaining surviving mutants, but to examine these selected cases in more detail. Some mutants survived because the generated assertions were too weak, while others survived because the relevant behavior was still difficult to observe after refactoring. For each case, we briefly restate why the mutant survived, since these mutants were already introduced in the *Targeted Mutant Effectiveness* sections.

4.3.1 JFreeChart

`XYPlot`: removed drawing call

The first non-killed mutant we investigated deeper is the mutant that survived in `XYPlot` in JFreeChart on line 2937, which was shown in Listing 4.1. This mutant is similar to several other surviving mutants, which are the mutants that remove calls to void methods (`VOID_METHOD_CALLS` mutation operator). In this class, these mutants survive because the added recording calls are still executed afterwards. In this case of the mutant on line 2937, the tests can observe that the drawing branch was reached, but not whether `drawDomainGridlines()` was executed, because the recording call is exactly placed after the execution call line.

Table 4.17: Selected surviving-mutant cases for follow-up analysis

Project	Class	Surviving mutant	Follow-up focus	# Mutants
JFreeChart	XYPlot	Removed drawing call *(Void Method Call)*	Drawing observability	1
JFreeChart	CategoryPlot	Removed crosshair anchor initialization *(Void Method Call)*	Assertion strength	2
Bukkit	StandardMessenger	Changed empty-set implementation *(Empty Returns)*	Implementation detail	1
Bukkit	PluginMsgListenerReg	Inverted null check for field *(Negate Conditional)*	Invalid object state	1
Bukkit	MemorySection	Inverted default-value type check *(Negate Conditional)*	Default-value behavior	1
Bukkit	SimpleCommandMap	Negated alias-registration check *(Negate Conditional)*	Alias-list observability	1
Bukkit	FileConfiguration	Removed save/close side-effect calls *(Void Method Call)*	Side-effect observability	2

If we only wanted to kill the mutant, it would have been sufficient to put the record operation inside the method `drawDomainGridlines()`, instead of after the drawing call. This small refactoring can be found in Listing 4.17. We verified that the mutant is killed, and this is because by removing the call to `drawDomainGridlines()`, it would then also prevent this state from being recorded with `recordDomainGridlinesDrawn()`. However, the test would still only check whether the method was actually executed, but we do not know whether the values sent to the renderer are correct.

We found a way to also check its behavior and not only its execution. Currently, we cannot observe its internal values, since the call is a void method. While analysing this refactoring, we found that this problem is similar to Feathers' example of separating a cash register display from the sale logic to make the displayed output testable [34, p.23-26]. In his example, he uses a fake object to test whether a `Sale` object sends the correct display information. He does not test whether the information is displayed correctly on a real screen. However, using such a test that checks if the correct information is sent to a fake display object can help developers see that the problem is not in `Sale` when there is a bug. This information can then be used to localize errors and save time. In our case, to make the behavior more observable (and simultaneously killing the mutant), we do not have to check pixel by pixel whether the domain gridlines are correctly drawn. We can use the insight of Feathers to make sure `lastDrawingOperations` (or any other object) checks whether the method sends the expected domain-gridline information to the renderer, including the tick

```
protected void drawDomainGridlines(Graphics2D g2, Rectangle2D dataArea,
                                   List<ValueTick> ticks) {

    [method body]

    this.lastDrawingOperations.recordDomainGridlinesDrawn();
}

[...]

// Inside the draw(Graphics2D, Rectangle2D, Point2D) method
if (domainAxisState != null) {
    drawDomainGridlines(g2, dataArea, domainAxisState.getTicks());
    this.lastDrawingOperations.recordDomainGridlinesDrawn();
    drawZeroDomainBaseline(g2, dataArea);
}
```

Listing 4.17: Refactoring of the domain-gridline recording call, where the recording is moved into `drawDomainGridlines()` instead of being executed separately after the method call.

```
The mutant on line 2958 in XYPlot.java, which is a void call method, survives.
     I want to make sure that the observability increases, so please refactor
    the code.

Surviving mutants may indicate insufficient observability when mutated
    behavior occurs internally but cannot be observed or asserted on by tests,
     leaving the test suite unable to distinguish behavioral differences.

On this line, I would like to know if drawDomainGridlines sends the correct
    information to drawDomainLine, which can be found on line 3600. Thus, can
    you refactor the code by creating a fake object, where the domain gridline
     information is sent to, to make sure information like the tick value,
    axis, data area, paint, and stroke is correctly sent? I want to make sure
    the values are correct.

Please also make tests for this and run the tests afterward.
```

Listing 4.18: Prompt used to guide the LLM in applying the fake-object based refactoring for the surviving `drawDomainGridlines` mutant.

value, axis, data area, paint, and stroke.

We tested this strategy. The prompt in Listing 4.18 is based on the earlier mutation-guided refactoring prompt, but only keeps the information needed for this specific case. Its purpose was to guide the LLM in turning the Feathers-inspired idea into a concrete refactoring and test.

The resulting implementation combines Feathers' fake-object idea with the LLM's earlier Observable State Pattern. The LLM added Observable State inside `XYPlot` to record the information that would be sent to the renderer. The LLM added a new `DomainGridlineRecord` class that stores tick value, axis reference, data area, paint, and stroke. Furthermore, it creates a recorder in `XYPlot` that stores these records during drawing. The newly added lines in the class can be seen in Listing 4.19. With these new additions, the tests can observe the values in the list `domainGridlineRecords`, and check whether these values are the ones that should be sent to the renderer.

However, if a mutant changes the actual call to `drawDomainLine()`, for example by replacing one of the arguments with `0`, the recorded line may still contain the correct values. In practice, this means that if a developer changes the `drawDomainLine()` call but does not update the `DomainGridlineRecord` creation, the tests may still pass while the original behavior is incorrect. This is why Feathers used a fake object, because the values are checked at the receiving side, without duplicating the call information in the class under test.

A possible solution is therefore to add a fake renderer that overrides `drawDomainLine()` and records the domain-gridline information it receives. This does not affect the production code, but the tests can then use behavior that is already accessible in the production design, because the renderer can be replaced and overridden. We also tested this approach, and the prompt can be seen in Listing 4.20. A hard constraint was added to the prompt because, in the first attempts, the LLM made wider changes to the test suite. And after this, PIT did no longer work. We did not investigate those failed attempts further, since this was not the focus of this small run. The focus was to check whether such a fake renderer could work. Thus, the prompt was altered to only append new tests.

The LLM added a test-only fake renderer by subclassing `AbstractXYItemRenderer` and overriding `drawDomainLine()`. This fake renderer is shown in Listing 4.21. The test now records values that are actually received by the renderer, instead of relying on values duplicated in the production code. The fake renderer stores the value, axis, data area, paint, and stroke. The tests check these values, for example, whether the tick values are in the domain range, whether the paint equals the configured paint, and whether the data area is non-null.

`CategoryPlot`: removed crosshair anchor initialization.

In `CategoryPlot`, the behavior was made observable by making `CategoryCrosshairState` directly accessible. Since this is similar to returning an object, in theory, the mutants should be able to be killed. However, the generated assertions were too weak to distinguish the original behavior from the mutant. We focus mainly on the mutants that survived on lines 3296 and 3297 in the original version, corresponding to lines 3314 and 3315 after the LLM refactoring. These are `crosshairState.setAnchorX(Double.NaN)` and `crosshairState.setAnchorY(Double.NaN)`. A test, as shown in Listing 4.22, is sufficient to kill these two mutants.

```
private List<DomainGridlineRecord> domainGridlineRecords;

public XYPlot(XYDataset dataset, ValueAxis domainAxis,
        ValueAxis rangeAxis, XYItemRenderer renderer) {
    [...]

    this.domainGridlineRecords = new ArrayList<>();

    [...]
}

public List<DomainGridlineRecord> getDomainGridlineRecords() {
    return Collections.unmodifiableList(this.domainGridlineRecords);
}

public void clearDomainGridlineRecords() {
    this.domainGridlineRecords.clear();
}

protected void drawDomainGridlines(Graphics2D g2, Rectangle2D dataArea,
                                   List<ValueTick> ticks) {
    [...]

    if ((r instanceof AbstractXYItemRenderer) && paintLine) {
        // Record gridline information for observability/testing
        DomainGridlineRecord record = new DomainGridlineRecord(
         tick.getValue(), getDomainAxis(), dataArea,
         gridPaint, gridStroke);
        this.domainGridlineRecords.add(record);

        ((AbstractXYItemRenderer) r).drawDomainLine(g2, this,
                getDomainAxis(), dataArea, tick.getValue(),
                gridPaint, gridStroke);
    }

    [...]
}
```

Listing 4.19: Refactoring of `drawDomainGridlines()`, where the LLM added recording state and helper methods to capture the parameters passed to `drawDomainLine()`.

```
Hard constraint: do not modify XYPlot.java or any production file. Do not
    modify, rewrite, or reorder any existing tests. Only append new code to
    XYPlotTest.java.

The mutant on line 2958 in XYPlot.java, which is a void call method, survives.
     I want to make sure that the observability increases, so please improve
    the tests using a test-only fake object. Surviving mutants may indicate
    insufficient observability when mutated behavior occurs internally but
    cannot be observed or asserted on by tests, leaving the test suite unable
    to distinguish behavioral differences. On this line, I would like to know
    if drawDomainGridlines sends the correct information to drawDomainLine,
    which can be found on line 3600. Thus, can you add a test-only fake
    renderer that overrides drawDomainLine and records the received domain
    gridline information, without adding recording state to XYPlot, to make
    sure information like the tick value, axis, data area, paint, and stroke
    is correctly sent? I want to make sure the values are correct.

Before editing, tell me the exact file you will change and confirm that the
    only intended change is appending a helper and new tests to XYPlotTest.
    java. Please also make tests for this and run only the newly added test
    method(s) afterward.
```

Listing 4.20: Follow-up prompt used to test the fake-renderer approach without changing production code.

4.3.2 Bukkit

`StandardMessenger`: changed empty-set implementation

The surviving mutant in `StandardMessenger` replaced the empty-set return `ImmutableSet.of()` with `Collections.emptySet()`. Both versions behave the same through the public API. When no registrations exist for a channel, the method returns an empty set. This was shown in Listing 4.11.

A test like in Listing 4.23 can kill this mutant by checking which empty-set implementation is returned. However, such a test is not very meaningful for the public API, because both versions return an empty set when no registrations exist. Therefore, this mutant is technically killable, but only by testing an implementation detail rather than relevant behavior.

`PluginMessageListenerRegistration`: inverted null check for field

The selected mutant changes the null-check condition `this.messenger == null` in the helper method `matchesMessenger()` that is shown in Listing 4.12. This mutant survived because the tests did not create a registration where the `messenger` field is null. However, this state is not reachable through normal object construction, since the constructor validates that the required fields are not null.

This mutant is different from the previous cases. The issue here is not only whether the test can observe the changed behavior, but also whether we can create the state needed to trigger it. So if one would like to kill this mutant, we have to bypass the normal construction.

```
static class FakeDomainLineRenderer extends AbstractXYItemRenderer {
    boolean drawDomainLineCalled = false;
    ValueAxis recordedAxis;
    Rectangle2D recordedDataArea;
    double recordedValue;
    Paint recordedPaint;
    Stroke recordedStroke;

    List<DomainLineCall> calls = new ArrayList<>();

    static class DomainLineCall {
        final double value;
        final Paint paint;
        final Stroke stroke;
        final Rectangle2D dataArea;

        DomainLineCall(double value, Paint paint, Stroke stroke,
                Rectangle2D dataArea) {
            this.value = value;
            this.paint = paint;
            this.stroke = stroke;
            this.dataArea = new Rectangle2D.Double(dataArea.getX(),
                    dataArea.getY(), dataArea.getWidth(),
                    dataArea.getHeight());
        }
    }

    @Override
    public void drawDomainLine(Graphics2D g2, XYPlot plot,
            ValueAxis axis, Rectangle2D dataArea, double value,
            Paint paint, Stroke stroke) {
      this.drawDomainLineCalled = true;
      this.recordedAxis = axis;
      this.recordedDataArea = dataArea;
      this.recordedValue = value;
      this.recordedPaint = paint;
      this.recordedStroke = stroke;

      calls.add(new DomainLineCall(value, paint, stroke, dataArea));
    }

    @Override
    public void drawItem(...) {
        // No-op for testing.
    }
}
```

Listing 4.21: Test-only fake renderer that records the parameters received by `drawDomainLine`.

```
@Test
public void drawWithNullAnchorGivesNaNForAnchors() {
    // create a normal drawable plot with one data value
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    dataset.addValue(1.0, "Row1", "Column1");

    CategoryPlot plot = new CategoryPlot(dataset,
            new CategoryAxis("X"), new NumberAxis("Y"),
            new BarRenderer());

    // then we draw without an anchor point
    BufferedImage image = new BufferedImage(400, 300,
            BufferedImage.TYPE_INT_RGB);
    Graphics2D g2 = image.createGraphics();
    plot.draw(g2, new Rectangle2D.Double(0, 0, 400, 300),
            null, null, null);
    g2.dispose();

    // the crosshair state should still be NaN
    CategoryCrosshairState state = plot.getLastCrosshairState();
    assertNotNull(state);
    assertTrue(Double.isNaN(state.getAnchorX()));
    assertTrue(Double.isNaN(state.getAnchorY()));
}
```

Listing 4.22: Test that checks whether drawing without an anchor initializes the recorded crosshair anchor coordinates to `NaN`.

```
@Test
public void emptyChannelReturnsImmutableSet() {
    Messenger messenger = getMessenger();

    Set<PluginMessageListenerRegistration> registrations =
            messenger.getIncomingChannelRegistrations("empty");

    assertTrue(registrations instanceof com.google.common.collect.ImmutableSet
    );
}
```

Listing 4.23: Test that kills the empty-set mutant by checking the concrete empty-set implementation returned by `getIncomingChannelRegistrations()`.

```
@Test
public void testUsingReflectionForNullField() throws Exception {
    Messenger messenger = getMessenger();
    Plugin plugin = getPlugin();
    PluginMessageListener listener = getListener();

    PluginMessageListenerRegistration reg1 =
            new PluginMessageListenerRegistration(
                    messenger, plugin, "test:channel", listener);

    PluginMessageListenerRegistration reg2 =
            new PluginMessageListenerRegistration(
                    messenger, plugin, "test:channel", listener);

    Field messengerField =
            PluginMessageListenerRegistration.class
                    .getDeclaredField("messenger");

    messengerField.setAccessible(true); // bypass private access
    messengerField.set(reg1, null);     // force invalid state

    assertFalse(reg1.matchesMessenger(reg2));
}
```

Listing 4.24: Reflection-based test that forces an invalid null messenger state to exercise the surviving null-check mutant.

A brute-force solution is changing the constructor which affects production code, or using reflection, which only affects the test suite. Reflection allows us to change fields at runtime, even when they are final. A working test that kills this mutant can be seen in Listing 4.24.

However, doing this is questionable because it tests an invalid object state, instead of behavior that can occur during normal use. This is related to Design by Contract, which Meyer explains that a class invariant should hold after object creation and should be preserved by the class methods [59]. In this case, the constructor prevents the required fields from being null. Therefore, forcing `messenger` to become null with reflection breaks the valid state that the constructor is meant to guarantee. Killing this mutant would therefore not improve testing of normal behavior, but would test a state that the class construction explicitly prevents. Next to this, allowing or exposing this invalid state would not improve the intended behavior of the class.

`MemorySection`: inverted default-value type check

The `MemorySection` class for this follow-up analysis was chosen because it had the highest number of surviving mutants in the Bukkit runs. The mutant chosen is also representative of the remaining type-check mutants, since the other cases follow the same pattern but with other return types, such as `double`, `long`, or `Vector`. The surviving mutant negates the check that tests whether the default value is a `Number`, which was shown in Listing 4.13.

This mutant can be killed by testing the case where the requested value is missing, but a default value exists. The test in Listing 4.25 kills this mutant because, in the original version, the default value `22.8` is recognized as a `Number`, converted to an integer, and returned. In the mutated version, the condition is negated, so the method uses `0` instead. And with this output, the test will fail.

```
@Test
public void getIntUsesTheDefaultWhenValueIsMissing() {
    // create a default configuration with 22.8 which is a Number
    MemoryConfiguration defaults = new MemoryConfiguration();
    defaults.set("test.value", 22.8);

    // create the real configuration and attach the default config
    MemoryConfiguration config = new MemoryConfiguration();
    config.setDefaults(defaults);

    // create the section, but do not set a real value in it
    MemorySection section = (MemorySection) config.createSection("test");

    // since the real value is missing, it will return the default
    assertEquals(22, section.getInt("value"));
}
```

Listing 4.25: Test that kills the surviving `getInt()` type-check mutant by checking that a numeric default value is used when the requested value is missing.

`SimpleCommandMap`: negated alias-registration check

The mutant we want to target is highlighted in the `register(String, String, Command)` method in Listing 4.14. This mutant changes `!register()` to `register()`, which affects whether an alias is removed from the command's own alias list. The reason this mutant survived is that the tests did not check the content of `command.getAliases()`. The test in Listing 4.26 is sufficient to kill this mutant, and at the same time kills the `iterator.remove()` void-call mutant. This is because when the condition is changed, or when the call to `iterator.remove()` is removed, the alias `"taken"` will not be deleted from the alias list, and the test will fail.

`FileConfiguration`: removed save/close side-effect calls

The mutants that survived are because most tests only checked whether the hook methods were called, but not whether the original lines were actually executed. We will investigate two specific mutants. These are the mutants on line 132, a file system operation line which is `save(new File(file))`, which got moved to the hook method and became `save(file)`, and line 223, an I/O operation line which is `input.close()` that got moved to the hook method and became `reader.close()`. This is shown in Listing 4.27. To test these two lines, we need to observe the effects of these lines.

```
@Test
public void failedAliasIsRemovedFromCommandAliases() {
    // first command already owns the label "taken"
    TestCommand existingCommand = new TestCommand("existing");
    commandMap.register("taken", "plugin", existingCommand);

    // we try to use "taken" as alias
    TestCommand newCommand = new TestCommand("newcommand");
    newCommand.getAliases().add("taken");

    // registering the second command will then fail for the alias "taken"
    commandMap.register("newcommand", "plugin", newCommand);

    // thus the failed alias should be removed from the second command's
    alias list
    assertFalse(newCommand.getAliases().contains("taken"));
}
```

Listing 4.26: Test that checks whether a failed alias registration is removed from the command's alias list.

```
protected void saveToFile(File file) throws IOException {
    save(file);
}

protected void closeReader(Reader reader) throws IOException {
    reader.close();
}
```

Listing 4.27: Hook methods added in `FileConfiguration`, where the highlighted delegated operations correspond to the surviving file-saving and reader-closing mutants.

For the `save()` function, a test like in Listing 4.28 is enough to kill the mutant. We are able to use observable behavior that is present in the public API to make sure that we can detect when the `save(file)` mutant is killed. Now, not only do we check whether the hook method got called, but also the actual `save()` call.

For the `input.close()` function, we can create a fake object, again inspired by Feathers, and with this, we can be sure the void call was correctly executed. This can be seen in Listing 4.29.

Overall, we can see that the remaining surviving mutants do not all indicate the same problem. Some mutants can be killed with stronger assertions on behavior that is already observable in the production code, either through the LLM refactoring or pre-existing observability, like in `CategoryPlot`, `MemorySection`, `SimpleCommandMap`, and the `save()` function in `FileConfiguration`. Other mutants required a small change in either the production code or the test suite. In `XYPlot`, for example, the mutant could be killed by

```
@Test
public void saveToFileActuallySavesFile()
        throws IOException, InvalidConfigurationException {
    // we create a new config file
    File file = testFolder.newFile("config.yml");

    // put in a key and value
    config.set("key", "value");

    config.saveToFile(file);

    // load the file
    YamlConfiguration loaded = new YamlConfiguration();
    loaded.load(file);

    // and when it saved properly we should be able to get its data
    assertEquals("value", loaded.getString("key"));
}
```

Listing 4.28: Test that checks the observable file-system effect of `saveToFile()` by saving a value and loading it again.

moving the recorder into the function. Alternatively, a fake renderer could be added to the test suite to observe the received values, instead of only checking whether the drawing call was reached. A similar test-side solution was used for the `close()` function in `FileConfiguration`. Lastly, there are mutants that are technically killable, but not necessarily meaningful to test. These include the mutants from `StandardMessenger` and `PluginMessageListenerRegistration`. Thus, this suggests that the remaining surviving mutants should not be treated as automatic evidence that more refactoring is needed, but also not as a reason to add tests only to kill more mutants. Instead, they need qualitative interpretation. Some surviving mutants can be killed with stronger assertions on behavior that is already observable, while others require the behavior to be made more observable, either through production-code refactoring or a test double. Other mutants are technically killable, but not meaningful to test.

```
@Test
public void closeReaderActuallyClosesReader() throws IOException {
    // create a new reader with the new fake object
    TrackingReader reader = new TrackingReader("key: value\n");

    config.closeReader(reader);

    // make sure the boolean is true
    assertTrue(reader.closed);
}

// create the fake object
private static class TrackingReader extends StringReader {
    boolean closed = false;

    TrackingReader(String content) {
        super(content);
    }

    @Override
    public void close() {
        closed = true;
        super.close();
    }
}
```

Listing 4.29: Test-only fake reader that records whether `close()` was called.

4.4 Summary of Findings

For each of the research questions, we will first give an overview of our main findings, and then give a summary answering our research questions. First, the observability-related research question is discussed, followed by the readability-related research question.

4.4.1 RQ1: Observability

The mutation score increased for both projects after applying the LLM-guided refactorings and generated tests. In JFreeChart, the average mutation score increased by 5.96%, and in Bukkit it increased by 47.89%. We should note that the average increase for Bukkit is affected by two runs that both started with a mutation score of 0.00%. Still, this shows that the test suites killed more mutants after the LLM-guided process. However, this is possible because of two different reasons: refactorings that made previously hidden behavior easier to observe, or generated tests that exercised behavior that was already observable but not tested before.

In JFreeChart, the refactorings mainly targeted parts of code with observability issues. These include void method calls that cannot be checked directly by tests when they are removed, calculations that are difficult to test because of them being mixed with graphical code, or the logic being implemented in a private method. These are hard to assert directly before the refactoring. By changing the return type, extracting calculation logic, changing its visibility, and exposing internal states, the LLM made these behaviors easier to test. However, Observable State Pattern was not always effective, because the generated tests sometimes only checked whether the operation was recorded or that the state object existed, but not whether the underlying behavior was correct. However, in our follow-up analysis, we saw it was possible to also check its underlying behavior.

In Bukkit, we can see that the mutation score increase was larger than in JFreeChart. However, runs 10 and 11 contributed to this larger increase, because both started with a mutation score of 0.00%. This happened because these classes had no direct tests before the run, so all covered mutants initially survived. At the same time, the refactorings were not always necessary to make the targeted behavior observable. This is because the same mutants could often already be killed through the observable behavior in the original public methods. These include invalid inputs, certain branches, or void calls like `Files.createParentsDirs(file)`. This means that in these cases, the main limitation was not always production-code observability, but not having direct tests. Some refactorings still improved direct observability that the tests made use of, for example, by exposing the internal map state, or extracting hashing logic and internal comparison to enable direct tests. However, in general, the Bukkit results show that mutation score improvements can also come from stronger tests rather than improving its observability.

The classification of the mutation operators gives some additional context for these results. In JFreeChart, the Observable State Pattern and Change Return Type runs mainly targeted `Void Method Calls` mutants, while the Extract Method runs mainly targeted `Math`

mutants. In Bukkit, the pattern was less clean, but some patterns were still visible. Extract Method targeted both `Math` and `Negate Conditionals` mutants in hashing and comparison logic, while the Template Method Pattern mainly targeted `Void Method Calls` mutants in side-effect operations. This does not mean that each refactoring technique only works for one mutation operator, but it shows which kinds of mutants the LLM targeted in these runs.

The targeted mutant results also show that killing mutants depends not only on exposing behaviour, but also on whether the generated tests assert the newly added behaviour precisely. In JFreeChart, four out of the six runs killed all mutants. The mutants in the two other runs were both related to testing the exposed state weakly. In one run in Bukkit, all the mutants were killed. In the remaining surviving mutants, they were often caused by not checking the exact changed behavior (e.g., not checking the alias list) or by states that were not allowed (e.g., having null in a field of an object). The follow-up analysis of selected surviving mutants further shows that these remaining mutants should be interpreted qualitatively. Some surviving mutants could be killed with stronger assertions on already observable behavior, while others required additional observability through a test double or a small production-code change. Other mutants were technically killable, but only through implementation-specific or invalid-state tests, making them less meaningful from a behavioral testing perspective.

Summary for RQ1

The results show that the extent to which LLM-guided refactoring improved production-code observability differed per case, depending on both the targeted mutants and the refactoring technique used. The refactorings were the most useful when they exposed hidden internal states (e.g., by changing return type or using helper methods) or behavior that is hard to access (e.g., by isolating calculations or changing the visibility of a method). This is because direct tests are then made possible. This was more visible in JFreeChart, where several mutants were located in graphical code that was difficult to assert through the original API. However, in Bukkit, many mutants could already be killed through the existing public behavior. Thus, the improvement in mutation score was in these cases because of the generated tests, rather than the newly enabled observability. Because of this, the mutation score alone cannot be interpreted as direct evidence of improved observability of production code, except when the killed mutants depend on behavior made observable by the refactoring. The follow-up analysis supports this, because remaining surviving mutants represented different situations: weak assertions, remaining observability gaps, and technically killable but behaviorally less meaningful mutants.

4.4.2 RQ2: Readability

Overall, the impact on readability was limited. However, there was a difference between JFreeChart and Bukkit. In JFreeChart at class level, the metric changes were small. The changes in cyclomatic complexity, maximum nesting depth, lines of code, number of methods, number of variables and loops, and the coupling are minimal. At method level, we can see that there is a notable decrease in lines of code, number of variables, and cyclomatic complexity for the methods where Extract Method was applied. This suggests that the JFreeChart refactorings did not strongly decrease readability, and in some cases even improved local clarity.

In Bukkit at class level, the metric changed more noticeably. The largest increases can be seen in the cyclomatic complexity, lines of code, and number of methods. This corresponds to the refactoring techniques used, where the LLM added helper methods like state-query methods, validation methods, or hook methods. However, the maximum nesting depth and the coupling did not increase.

The qualitative analysis can provide more insights than metrics alone. We judged Extract Method as having the biggest readability improvement in both projects, because it moved certain parts of code (e.g., calculations or comparison logic) into smaller methods with clear names. While this increases the overall size of the class, this change is usually very small and limited to the extracted part. However, helper methods that mainly existed for testing had a more mixed effect. They were often short and understandable individually, but they made the class heavier because they added extra methods without improving the original production-code behavior. This was mainly visible in Bukkit, where several methods were not necessary to kill the targeted mutants. In some cases, it even duplicated parts of the original logic. Thus, the main readability impact is not that the original methods became harder to understand, but the classes now consist of a mix between methods for their functionality and also for their testing.

Summary for RQ2

The readability impact of the LLM-guided refactorings was limited overall, but it varies between the refactoring types and projects. In JFreeChart we can see that the impact was minimal at class level, but was improved locally when Extract Method was used. This is because it replaced multiple lines of calculation logic with one clear method name. In Bukkit, the impact is more mixed. Most of the individual methods had no change in readability, but at class level there was an increase in size because of helper methods, which could make the classes a bit heavier to understand. This is because there is now a mix between functionality-related methods and methods that are only used for testing purposes. Thus, the approach did affect readability, but mainly through newly added code structures, rather than stronger coupling or a complex control flow.

Chapter 5

Discussion

This chapter discusses the meaning and limitations of the results presented in the previous chapter. We first answer the main research question, and then reflect on what mutation score improvements actually mean, the difference between execution observability and behavioral observability, and the implications for mutation-guided LLM refactoring. Finally, we discuss the threats to validity.

5.1 Answering the Main Research Question

Our main research question is: **To what extent can refactoring using LLMs, guided by mutation testing results, increase the observability of production code, while not decreasing the readability of the code?** The answer is that LLM-guided refactoring can effectively increase production-code observability with only a limited impact on readability, but mainly when the targeted mutants point to behavior that is difficult to observe through the existing code structure. Thus, the extent to which this happens depends on the targeted mutant, the refactoring technique, and whether the generated tests actually use behavior made observable by the refactoring.

For observability, a positive increase was especially seen for hidden internal states, private calculation logic, hard-to-access graphical behavior, and some void method calls. When the behavior was already observable through the public API, the improvement in mutation score came more from stronger generated tests than from increased production-code observability. This also means that an increase in mutation score cannot automatically be interpreted as an increase in production-code observability.

For readability, the impact was limited overall, but also depended on the refactoring technique. Extract Method was the refactoring with the most positive effect, because it could make behavior easier to test while also improving local clarity. Other refactorings, such as Added Query Methods or Extract Type Validation Methods, had a more mixed effect. These methods can make internal behavior easier to test directly, but they can also make production classes heavier. This is because the class then contains a mix of functionality-related code and testing-oriented code, and sometimes even contains unnec-

essary code duplication.

Thus, the approach can increase observability with only a limited impact on readability effectively, but this is not guaranteed. It works best when the refactoring exposes meaningful behavior that was previously difficult to assert, and when this is done with limited additional structure.

5.2 Execution Observability versus Behavioral Observability

As shown in the previous chapter, some refactorings made it possible to check whether something was executed, but not whether the executed behavior was correct. This can be seen in run 1 for `XYPlot` and in run 12 for `FileConfiguration`. In `XYPlot`, which was shown in Listing 4.1, the added state could record that a drawing operation happened, but not whether the correct values were passed to the renderer. In `FileConfiguration`, which was shown in Listing 4.27, the added hook methods could show that a reader-closing hook was called, but not automatically whether the reader was actually closed. The follow-up analysis showed that these cases can be tested more deeply by using test doubles. A fake renderer in `XYPlot` made it possible to check the values received by the renderer, while a fake reader in `FileConfiguration` made it possible to check whether the reader was actually closed. These solutions are different from the original goal of the refactoring runs, because they improved observability mainly through the tests, rather than by changing the production code itself. So, they used test doubles for objects that the production code already interacts with, such as the JFreeChart renderer and the reader, to check the behavior more directly in the tests. However, they did make the tests more informative, because they checked more than only whether a line or hook was reached.

This shows a difference between what we can call execution observability and behavioral observability. Execution observability means that a test can observe that a line of code or hook was executed. This is already useful, because it gives more information than not knowing whether the code was reached at all. However, it is still limited, because it does not necessarily show whether the executed behavior was correct. Based on these observations, we can use three levels of observability to interpret the tests in this study. The first level is an execution check, where the test checks whether the relevant line, method call, or hook was executed. The second level is an interaction check, where the test checks whether the correct values were sent to the object that receives the call. The third level is an end-to-end behavior check, where the test checks whether the final output was correct. The third level is the strongest, but it is not always possible, as with a real display in Feathers' example [34]. At other times, it is not very practical. For example, checking the final rendered output in graphical code pixel by pixel can make tests more brittle. In such cases, a fake object can be a useful middle ground. We can use Feathers' example of using a fake display to check whether the correct information is sent to the display, without checking whether the hardware display itself renders the output correctly [34].

Therefore, execution checks should not necessarily immediately be seen as being wrong.

The exact level for testing that is appropriate depends on the context and on what the developer wants to verify. Deeper checks can give stronger evidence, but they can also make tests more complex or more coupled to implementation details. Therefore, mutation-guided refactoring should not only try to kill mutants, but should also consider what level of observability is useful for the behavior being tested.

5.3 Implications for Mutation-Guided LLM Refactoring

The results suggest that, for mutation-guided LLM refactoring, increasing the mutation score should not be the only objective when improving observability. It also matters how the score is increased and what effect the chosen solution has on the production code and the generated tests. If the LLM is only guided by the goal of killing surviving mutants, it may choose solutions that improve the mutation score, but are not the best solution for the codebase. For example, it may add helper methods to production code when stronger tests through the existing public API would have been enough.

However, mutation testing is a useful basis for guiding the LLM, because it gives concrete targets where observability issues may exist. At the same time, instead of asking the LLM to refactor immediately, the workflow should first make the LLM reason about what kind of problem the mutant represents. Mutation-guided LLM refactoring should therefore not be treated as only an immediate refactoring task, but more like a decision problem. A surviving mutant may be because of missing tests for behavior that is already observable, a real production-code observability problem, an implementation detail that is not meaningful to test, or an invalid state. This is related to Brandt et al. [12], who found in their study that developers first consider whether a coverage gap in the source code is worth testing before deciding how to address it. Some gaps were not considered worth the effort because the code was unlikely to contain a bug, unlikely to be reached, or already covered by other tests. This supports the idea that signals from testing should first be interpreted before they are treated as something that must be fixed.

After a refactoring has happened, a still-surviving mutant can also point to a weakly generated assertion. For example, this can be an assertion that always holds, or a case where the test observes a separate recording call that still runs even when the original void call is removed. This relates to the test oracle problem, because the test may observe something, but the assertion still has to check whether the observed behavior is correct [9].

This also means that the LLM has several possible responses to a surviving mutant, both in its initial run and after its first iteration. It can add a stronger test through the public API, use a test double, extract a method, expose state by returning an object or storing it in a field with a getter, add a hook, change visibility, or ignore the mutant. The results suggest that these options are not equally desirable in every situation. Extract Method, changing the return type, visibility changes, and the Observable State Pattern were useful when they exposed behavior that was previously difficult to assert, while having a limited impact on readability. In contrast, query methods or other helper methods are less useful when the same behavior is already visible from the outside. In those cases, the LLM may

add production-code structure that is not needed.

Another implication is that improving observability should not automatically mean exposing more of the production code. Some refactorings that make mutants easier to kill can conflict with object-oriented design principles [11], especially encapsulation. An example is when private methods are made public or protected only for testing purposes [84]. Therefore, the LLM should not only ask whether a refactoring can kill a mutant, but also whether the refactoring is justified compared to alternatives that also may be possible, such as stronger assertions, a test double, or ignoring a mutant that is not meaningful to test.
Thus, future prompts should steer the LLM to choose the smallest useful change for the specific mutant, instead of directly asking it to refactor the production code. Before changing production code, the LLM should first check whether the mutant can be killed through the existing public API. If this is possible, stronger tests may be sufficient. If this is not possible, production-code refactoring (if justified when conflicting with OO design principles) or a test double may be more appropriate. The prompt should also make the LLM consider what level of observability is appropriate for the specific context. This is important when applying the approach to larger codebases, because otherwise the LLM may create extra helper methods or testing-oriented code that improves the mutation score, but makes the production code heavier to maintain.

5.4 Bug Discovery During LLM-Guided Refactoring

The LLM was able to fix a bug during one of our runs. This happened in `StandardMessenger` and can be seen in Listing 5.1. One of the tests that the LLM created failed because of the difference between what the LLM expected to happen and what actually happened. Thus, the LLM saw this and corrected it in the production code.

What actually happened in the test is shown in Listing 5.2. The two maps `outgoingByChannel` and `outgoingByPlugin` show a relationship that goes both ways. When the test unregisters plugin 1, which is connected with `"test:channel"`, we expect plugin 1 to be removed from the map `outgoingByPlugin` when the `channels` set of plugin 1 is empty. Thus, the line `outgoingByPlugin.remove(plugin)` should run. However, in the buggy code, `outgoingByChannel.remove(channel)` is run, and thus `"test:channel"` is removed, while `"test:channel"` still has plugin 2 in its set.

Because of this, we made a pull request (PR) to let a moderator check whether they also agreed that it was a bug. This PR got accepted and is now in the original source code [14]. We made a change to the test, since it was possible to create a test without the helper methods introduced by the LLM that still showed it was a bug. This was done to comply with the rule of "no unnecessary code changes", and we judged that the helper methods were not needed to show that it was a bug. However, in the original run, the LLM still had a reason to add the helper method, because there was no possibility to check whether plugin 1 is really deleted from the `outgoingByPlugin` map. This is because `getOutgoingChannels(Plugin plugin)` returns an empty set when `channels` is null, so we cannot differentiate whether plugin 1 is properly cleaned up from `outgoingByPlugin`

without a helper method. But with this fix locally, we already saw that it works, so the PR test only showed the larger error that `"test:channel"` got deleted. We also stated that we deleted the helper methods, and asked whether Bukkit prefers avoiding such helper methods when the bug can be tested through the public API. MD_5, who is the moderator, answered that they think this kind of helper method would be unnecessary in this case given the public API, and thus agreed with us.

This suggests that mutation-guided LLM refactoring with testing can expose behavior that is a real defect.

```
private final Map<String, Set<Plugin>> outgoingByChannel =
    new HashMap<String, Set<Plugin>>();

private final Map<Plugin, Set<String>> outgoingByPlugin =
    new HashMap<Plugin, Set<String>>();

[...]

private void removeFromOutgoing(Plugin plugin, String channel) {
    synchronized (outgoingLock) {
        Set<Plugin> plugins = outgoingByChannel.get(channel);
        Set<String> channels = outgoingByPlugin.get(plugin);

        if (plugins != null) {
            plugins.remove(plugin);

            if (plugins.isEmpty()) {
                outgoingByChannel.remove(channel);
            }
        }

        if (channels != null) {
            channels.remove(channel);

            if (channels.isEmpty()) {
                outgoingByChannel.remove(channel);
                outgoingByPlugin.remove(plugin);
            }
        }
    }
}
```

Listing 5.1: Bug fix in `StandardMessenger`, where the LLM corrected the cleanup of the plugin-to-channel map in `removeFromOutgoing`.

```
Before unregistering plugin1:

    outgoingByChannel:
    "test:channel" -> { plugin1, plugin2 }

    outgoingByPlugin:
    plugin1 -> { "test:channel" }
    plugin2 -> { "test:channel" }

Expected after unregistering plugin1:

    outgoingByChannel:
    "test:channel" -> { plugin2 }

    outgoingByPlugin:
    plugin2 -> { "test:channel" }

Incorrect old behavior:

    outgoingByChannel:
    "test:channel" entry removed

    outgoingByPlugin:
    plugin1 -> { }
    plugin2 -> { "test:channel" }
```

Listing 5.2: Expected and incorrect outgoing-channel state when unregistering one plugin from a shared channel.

5.5 Threats to Validity

This section discusses the main limitations of the study. Following Wohlin et al. [80], we discuss threats to construct validity, internal validity, and external validity.

5.5.1 Construct Validity

Construct validity is about whether we measured the concepts that we intended to measure. In this study, to measure observability, the mutation score is used as an indicator of observability, but not as a direct measure. This is because tests are required to kill mutants, and an increase in mutation score can come from different causes. It can come from refactorings that make previously hard-to-observe behavior more observable with their respective tests, but it can also come from stronger tests for behavior that was already observable. Therefore, we do not interpret the mutation score alone as evidence that production-code observability increased.

To reduce this threat, we also analysed the refactoring that was applied, the targeted mutants, the effectiveness of the refactoring, and the generated tests. This made it possible

to interpret whether the mutation score increase was caused by newly enabled observability or by stronger tests for already observable behavior.

A similar limitation applies to readability. Metrics such as LOC, WMC, maximum nesting depth, number of methods, and similar metrics cannot fully determine whether readability improved, since readability also depends on human interpretation. Therefore, we added a qualitative analysis. However, this introduces researcher judgment, which means that there is a potential risk of bias. To limit this bias, we used the same criteria and the same scale of improved, neutral, or worsened for each run.

Lastly, equivalent mutants were not manually removed. This can affect the exact mutation score, because some surviving mutants may not represent meaningful behavioral differences. However, this is a common approach when the mutation score is used as a relative comparison [83].

5.5.2 Internal Validity

Internal validity is about whether the results may have been affected by factors that were not intended as part of the study. In our study, the LLM both refactored the production code and generated tests. Therefore, it is difficult to fully isolate the effect of the refactoring from the effect of test generation. As we have seen, some mutation score increases were caused by stronger tests and not by improved production-code observability. We tried to mitigate this by analysing the generated tests and the targeted mutants, mainly by checking whether the tests depended on the new refactorings and whether the targeted mutants were killed after the changes.

Another threat is that the LLM may have changed the behavior during the refactoring. This could distort the mutation results, because changes in killed, survived, or no-coverage mutants may then be caused by changed program behavior rather than improved observability. For example, as we have shown in Section 3.2.3, if a refactoring accidentally introduces an early return, code that was previously executed may no longer be covered. And this causes some mutants to become no-coverage. Running the existing tests reduces this risk, but it cannot prove that all behavior stayed the same, since the changed behavior may not be covered by the tests. Therefore, we also manually checked whether the refactorings introduced unintended behavior changes.

Lastly, the refactorings themselves can introduce new mutants or change the path that the tests execute, like in the hook methods in run 12. Because of this, changes in killed and surviving mutants are not always a direct one-to-one effect of the targeted refactoring. This is why we interpreted the mutation results together with the applied refactorings and the targeted mutant outcomes.

5.5.3 External Validity

External validity is about whether the findings can be generalized beyond the studied projects. The main study includes two open-source Java projects, which limits generalizability. Both projects are Maven-compatible and have existing test suites. The results could be different for other programming languages, projects with weaker test suites, other LLMs, other

prompts, and other agentic workflows. However, using two projects allowed us to do a deeper analysis of each refactoring run in the given timeframe. JFreeChart and Bukkit are also different enough to show that the approach behaves differently depending on the codebase and the type of mutants. Future replications on more classes, written in different programming languages, are advised.

5.6 Ethical Considerations

Ethics are important in computing because there can be consequences outside the system itself, including affecting people, society, organizations, and the environment. This is the reason why the ACM Code of Ethics and Professional Conduct states that the public good should be the primary consideration [7]. In our thesis's case study, one of the ethical considerations is energy and water consumption. This issue has already been studied in prior work. For example, Luccioni et al. measured the energy consumption and carbon emissions of different models [56], and Li et al. presented a methodology to estimate AI's water footprint, which showed that large AI models can consume millions of liters of water for training [54]. This environmental consideration corresponds to *1.1 Contribute to society and human well-being* and *1.2 Avoid harm* of the ACM Code of Ethics [7]. Next to this, we did not have to consider privacy risks around personal data from human participants or confidentiality risks around company code in our study, relating to *1.6 Respect privacy* and *1.7 Honor confidentiality* [7]. This is because we do not use human participants and run everything on open-source code. However, it is important to be ethical when using open-source code, so we kept the licences with JFreeChart having the LGPL-2.1 licence and Bukkit having the GPL-3.0 licence. This corresponds to *1.5 Respect the work* [7].

In practice, when applied in a company, we have to consider confidentiality risks around company information. This includes the source code on which the LLM is used, because in our workflow, one of the inputs is the whole codebase. When using external LLM providers, we need to consider how the data is stored, processed, and secured. In Schuberg Philis, this is the reason why they have AI Security Guidelines and Guardrails. Also, another practical ethical consideration is how to use AI responsibly. This is because AI can hallucinate, and, for example, suggest packages that are outdated and vulnerable. Ways to reduce this risk are that only engineers with relevant domain knowledge and judgement will direct and evaluate AI output, and that humans take full accountability for what is delivered, as also required by Schuberg Philis. In our study, we reduced the risk of incorrect code by running the existing and generated tests after the changes. We also manually inspected the generated changes to check carefully whether the refactoring introduced unintended behaviour changes.

5.7 Use of Generative AI

We used generative AI in this thesis in two ways. First, it was part of the research method itself, since an LLM agent was used for the refactoring process as we described earlier. Second, AI was used as support during development, analysis, and writing. This included step-by-step assistance for implementing the MCP server and scripts, debugging the projects

during the experimental setup, helping to formulate the structured prompts, helping to articulate reasoning that we had already developed during the analysis (like explanations for surviving mutants), improving the replication package documentation, helping with the captions and tables, and revising already-written report text for clarity, grammar, spelling, and punctuation. We checked all AI-generated suggestions, code, and text before we used them and corrected or adapted them when needed.

We did not use AI to search or select the papers used in the Introduction and Related Work, since we searched these ourselves using Google Scholar [39] and tracked which ones to choose in Google Sheets [40]. Also, the case-study design, project selection, class-selection criteria, and evaluation metrics were chosen by us based on our research goals and related work.

Chapter 6

Related Work

This chapter discusses the related work for this thesis. We start with the work that is closest to our topic, which is the use of mutation testing and surviving mutants as an indication of observability. After this, we look at automated refactoring more generally, showing how refactoring was automated before the recent use and rise of LLMs. Then, we discuss LLM-based refactoring. We first look at studies that examine whether LLMs can improve code quality and reduce complexity. After that, we will briefly discuss studies on LLM-based refactoring methods, frameworks, and prompting. Lastly, we look at readability. We discuss work where LLMs are used to improve readability, and then studies on how readability can be assessed.

6.1 Mutation Testing and Code Observability

As we have seen before, mutation testing is a well-known fault-based software testing technique that has been widely studied since the 1970s [49]. Zhu et al. [84] introduced a new way to interpret mutation scores by analyzing them through the lens of testability and observability. They investigated whether testability and observability metrics are linked to mutation score on six open-source Java projects. They found that observability metrics were more strongly related to mutation score than existing code quality metrics. Among the observability metrics, test directness was especially important. Test directness captures how directly the tests exercise the production code under test.

Based on these results, Zhu et al. proposed mutation score anti-patterns. These anti-patterns indicate situations where mutants may survive because the behavior is difficult to observe or because the tests do not check it directly enough. In their case study, they showed that removing these anti-patterns can increase the mutation score. This included adding direct tests, adding assertions, changing private methods to protected or public methods, changing void methods to non-void methods, and adding getters for private state.

Several refactorings observed in this thesis are similar to these anti-pattern removals. For example, in `PiePlot3D` of `JFreeChart`, the LLM changed the visibility of production code. In `PiePlot` of `JFreeChart`, it changed a return type, and in several `Bukkit` runs it added getters or query methods. This shows how this thesis builds on the work of Zhu et

al. [84], because both use surviving mutants as a signal that some behavior may be difficult to observe. However, this thesis studies a different setting, because the refactorings are produced by an LLM agent guided by mutation testing results, instead of being manually derived from predefined anti-patterns.

6.2 Automated Refactoring

Since this thesis uses an LLM agent to refactor production code, it is useful to first consider prior work on automated refactoring more generally. This shows how refactoring was automated before the recent use of LLMs, what kind of information was used to decide where to refactor, and how the code should then be refactored. A lot of studies have already been done to refactor code automatically, mainly with the goal to reduce the manual effort for developers by automatically identifying design problems and proposing refactoring operations [8].

TrueRefactor [41], for example, is a tool that supports automated refactoring using UML models. It takes a Java codebase as input, detects code smells, and uses a genetic algorithm to search for a sequence of refactorings that removes as many code smells as possible. The tool focuses on improving the comprehensibility of legacy systems, for example by improving understandability, maintainability, and reusability. However, the version described in the paper does not yet directly rewrite the Java source code. Instead, it applies the refactorings to UML models and generates updated UML class diagrams.

There is another study done, where Hunold et al. [45] propose a pattern-based transformation process for legacy software systems, which improves the source code automatically. This is also done through an intermediate model. First, they convert the source code into an abstract software model that stores the structure of the source code and the relationships between code elements. Then, in the refactoring step, predefined patterns are searched for. Lastly, a predefined transformation rule is applied. In this study, they focus on making the code less coupled by reducing the number of dependencies within a software architecture to improve code quality.

However, as one can see, these two studies rely on models or intermediate representations. There are also studies done on refactoring applied directly to the source code. Examples of this are Spartanizer [38] and FaultBuster [74]. Spartanizer [38] focuses on making code more minimal and compact by reducing size complexity, and FaultBuster [74] focuses on identifying code smells and fixing them. There has also been a study that experimented with whether cohesion-focused automated refactoring improves program testability [20]. However, the results were inconclusive as most developers had no preference between the pre-refactored code and post-refactored code.

From these automated refactoring studies, we can see the difference with this thesis. Some approaches rely on models or intermediate representations, while this thesis refactors the production source code directly. Other approaches operate directly on source code, which is already closer to our thesis, but they mainly focus on reducing the manual effort for developers, identifying code smells, or reducing complexity. This thesis instead focuses mainly on using surviving mutants to guide refactoring toward increased observability. And

of course, our thesis refactors with an LLM, while these refactoring studies do not use an LLM.

6.3 LLM-based Refactoring

Because of the significant advancement in Large Language Model (LLM) technology, there has been an increase in studies examining the use of LLMs for refactoring. We will first go over the studies that focus on the ability of LLMs to generate refactorings, and then go over some LLM-based refactoring methods and frameworks.

6.3.1 LLMs for Code Quality and Complexity

Some studies focus on the ability of LLMs to generate refactorings that improve code quality and complexity metrics [25, 72, 67, 55, 26, 18]. For example, in the study of Cordeiro et al. [25], they evaluate the effectiveness of code refactored by LLMs and compare it to that of developers. They found out in their study that StarCoder2, the LLM used, which was chosen to mitigate the risk of data leakage, reduced code smells more often than developers, with a reduction of 44.36% compared to 24.27% for developers. They also found that StarCoder2 improved several code-quality metrics. However, the results also show a limitation. StarCoder2 performed better on systematic and repetitive issues, such as shortening long statements or long parameter lists, while developers performed better on more complex and context-dependent code smells, such as broken modularization or deficient encapsulation. This is relevant for this thesis because it shows why LLMs are promising for refactoring, but they also found out that more complex refactorings still require guidance and evaluation. In this thesis, the guidance comes from surviving mutants, and the refactorings are evaluated in terms of observability and readability instead of only code-smell reduction or general code-quality metrics.

Another study used GPT-3.5 to suggest less complex versions of user-written Python programs [72]. The authors generated multiple refactoring candidates and only kept programs that were semantically correct. They found that most programs could be refactored, with reductions in average cyclomatic complexity and number of lines. This is relevant because it shows that LLMs can reduce structural complexity, although their focus is different from this thesis. Their work focuses on making programs less complex, while this thesis uses surviving mutants to guide refactoring toward increased observability.

Liu et al. [55] investigated the potential of LLMs for automated software refactoring, focusing on identifying refactoring opportunities and recommending refactoring solutions. Their results show that LLMs can struggle when the task is too open-ended, but that performance improves when the prompt explains the expected refactoring categories and narrows the search space. They also found that some LLM-generated refactorings were unsafe, because they changed functionality or introduced syntax errors. This is why they propose `RefactoringMirror`, which is a detect-and-reapply tactic to avoid refactorings that are unsafe. This is related to the limitations discussed by Cordeiro et al. [26]. They argue that LLM-based refactoring can suffer from limited contextual awareness, hallucinations, a lack

of understanding of the purpose behind the refactoring, and insufficient tests to validate the generated changes.

These two studies are relevant for this thesis because mutation-guided refactoring also requires the LLM to understand the purpose of the change. This is the reason why our prompt explicitly stated that surviving mutants can indicate observability problems and that the refactoring should make the affected behavior easier to assert in tests. Next to that, since LLM-generated refactorings can be unsafe, the prompt also instructed the agent to run the existing tests after making changes. If the code did not compile or the tests failed, the agent had to fix the changes before continuing. This does not prove that the refactoring is fully safe, because the existing tests may not cover all behavior. However, it reduces the risk of accepting refactorings that clearly break the current functionality.

6.3.2 LLM Refactoring Methods, Agents, and Prompting

Other studies focus on LLM-based refactoring methods and frameworks [19, 51, 63]. For example, one study used an iterative approach where methods with high cyclomatic complexity were selected and then refactored by an LLM. After several iterations, the average cyclomatic complexity decreased, with the largest reduction being 10.4% after 20 iterations [19]. This shows that LLMs can also be used in a repeated refactoring process, instead of only asking for one refactoring once. RefactorGPT [51] also takes a more structured approach. Instead of treating refactoring as one single task, it divides the process into multiple steps handled by different agents. The authors report improvements in modularity, readability, and decomposition of the code. RefAgent [63] is another multi-agent framework for LLM-based refactoring. In their evaluation, most generated refactorings passed the unit tests, and the number of code smells was reduced.

Lastly, there is also research on prompt engineering. Alomar et al. [2] studied refactoring conversations between developers and ChatGPT. They looked at how developers ask for refactorings and how ChatGPT responds to these requests. Based on this, they proposed a prompt template that helps developers get better refactoring answers with fewer interactions. This is relevant for this thesis, because our approach also uses one main prompt without repeatedly reprompting the agent. Therefore, the prompt had to contain enough context about the goal, the mutation results, and the expected refactoring direction.

As we have seen, prior work suggests that LLMs can be useful for refactoring. They can improve code quality, reduce code smells, and lower complexity in some settings [25, 72, 19]. However, most prior work on LLM-generated refactorings focuses on structural code quality improvements, such as reducing code smells or reducing cyclomatic complexity. Refactoring with an LLM from a testing perspective is still less explored. This is what this thesis studies, by using surviving mutants to guide refactoring toward increased observability.

6.4 Readability

Since our thesis also looks at readability, we will first discuss a study that uses LLMs in IDEs and the impact on readability, and then discuss studies that try to assess code readability.

6.4.1 LLMs and Readability

Rusum [67] studied the use of LLMs in IDEs, including their use for refactoring. Their results suggest that LLM-assisted refactoring can improve readability. In their study, readability increased from 63% without LLM support to 86% with LLM support. They explain this by stating that LLMs can restructure code, improve variable names, and simplify complex logic. They also report a decrease in cognitive complexity, from 100% in the baseline to 60% with LLM support. This is relevant for this thesis because it shows that LLM-based refactoring can affect readability-related properties of the code, not only code smells or cyclomatic complexity.

However, the same paper also points out limitations of LLM-based development, especially hallucinations. These are changes that may look syntactically correct but are semantically wrong or unsafe. Therefore, LLM-generated refactorings should not only be judged by whether they look cleaner or more readable, but also by whether they preserve behavior. This is why in our study, we also checked if the behavior was preserved manually.

6.4.2 Assessing Readability

Next to this, readability is hard to assess since it is based on human judgment. Therefore, we also looked at work on how readability can be estimated automatically. Buse and Weimer [16] constructed an automated software readability metric. They first collected readability judgments from 120 human annotators, and then used machine learning to learn which local code features were related to these judgments. Their work is relevant because it shows that readability can be estimated automatically to some extent, but also that it should still be interpreted carefully because readability starts from human judgment. The reason we did not use their metrics is because they are very localized. They look at the readability per line, but in our case, the refactoring will target bigger parts of the code, like extracting lines out of a method. Per line, the difference is then minimal.

Buse and Weimer are not the only ones who have tried to estimate readability automatically. Namani and Kumar [62] also proposed a code readability metric based on rules collected from senior software engineers, such as lines of code, line length, comments, blank lines, and number of methods. They implemented this in a prototype application and compared the results with readability scores given by 50 software engineers. The results were close to the human scores, which suggests that automatic metrics can estimate readability to some extent.

Rahman et al. [65] also developed a tool to measure program comprehensibility. Their approach focuses on textual elements in the source code, such as package names, class names, method names, and variable names. The tool gives scores at class and project level, which can help developers identify code that may need refactoring.

However, automatic readability models still have clear limitations. Fakhoury et al. [33] found that existing readability models often did not detect readability improvements that developers explicitly made in practice. This shows that readability improvements cannot be fully captured by one automatic model. A similar point can be seen in related work on test understandability, which is closely related to readability. Deljouyi and Zaidman [29] distinguish readability from understandability and state that understandability is more qualitative and difficult to capture in a predictive model. Deljouyi et al. [30] study generated test understandability through several elements, such as comments, test data, test names, variable names, and assertions. Because of this, this thesis does not use one readability model as the final answer. Instead, we use several static metrics as signals of whether the refactoring made the code structurally heavier or simpler.

This choice is also supported by prior work that links readability to complexity. Studies show a negative relation between complexity and readability [1]. Other work shows that reducing nesting, loops, and cyclomatic complexity is often connected to improved readability [71, 33, 50]. Therefore, this thesis uses metrics such as complexity, nesting, lines of code, variables, loops, and coupling as readability-related indicators. The books of Clean Code [58] and Refactoring [35] emphasize simple and direct code, focused responsibilities, and code that communicates its purpose, on which our criteria for the qualitative readability analysis is based on.

Chapter 7

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

7.1 Contributions

In this thesis we make the following contributions:

- We created an execution-validation workflow where surviving mutants are used to guide an LLM agent toward refactoring with the goal of increasing observability. This workflow consists of the LLM analysing mutation results using an MCP server, selecting surviving mutants, refactoring the code, running the test suite, generating new tests, and finally rerunning PIT for evaluation. This builds on the work of Zhu et al. [84], where they connect mutation score and observability. This thesis studies what happens when an LLM agent is tasked to do this automatically.
- An empirical case study was conducted on two Java projects using this new workflow. The study reports mutation score changes, the refactoring strategies that were used, the targeted mutant outcomes, an analysis of the generated tests, readability-related metrics, and a qualitative analysis of readability.
- We provide a qualitative interpretation of mutation score improvements. This showed that the mutation score should only be used as an indicator of observability. In some cases, the mutation score increased because of better tests, rather than because of improved production-code observability. Other tests only checked execution, and not correctness. We also discussed that the appropriate level of observability depends on the context and on what the developer wants to verify in Section 5.2.
- We show that mutation-guided LLM refactoring should be treated as a decision problem, not only as an immediate refactoring task. A surviving mutant may require a stronger test, a test double, a production-code refactoring, or no action when the mutant is not behaviorally meaningful. For the remaining surviving mutants, we found

that some mutants are technically killable but behaviorally not meaningful, some require test doubles or deeper assertions, and some point to invalid states.

- We provided insights on refactoring strategies and readability trade-offs. Extract Method was often the cleanest refactoring, while query methods, recording state, and hook methods had a more mixed effect because they sometimes exposed behavior that was already testable through the existing public API, making production classes heavier by mixing functionality-related code with testing-oriented code.
- We observed that mutation-guided LLM refactoring with generated tests can also expose real defects, which we explained in Section 5.4. In one Bukkit run, the generated test revealed a bug in `StandardMessenger`, which was later confirmed through an accepted pull request.
- We provide a replication package that includes the prompts, scripts, post-run analysis data, and links to the MCP server and experiment repositories with the modified project setup, where each run is stored on a separate branch with its respective PIT report [32].

7.2 Conclusions

Before drawing the final conclusion, we will first return to the research questions discussed in Section 4.4 and Section 5.1. The main research question of this thesis was:

> To what extent can refactoring using LLMs, guided by mutation testing results, increase the observability of production code, while not decreasing the readability of the code?

For RQ1, which focused mainly on the impact on observability, the results show that the extent to which LLM-guided refactoring improved production-code observability differed per case, depending on both the targeted mutants and the refactoring technique used. The mutation score increased in all runs, which means that the final test suites killed more mutants after the LLM-guided process. But this does not always mean that the production code itself became more observable. In some cases, which we mainly saw in JFreeChart, the refactorings made behavior easier to observe by changing the return type, isolating calculations, or changing the visibility of a method. In other cases, especially in Bukkit, many mutants could already be killed through the existing public API, and the improvement came mainly from stronger generated tests. From our study, this also indicates that the mutation score is useful as an indicator, but not as direct evidence that production-code observability improved.

For RQ2, which focused mainly on readability, the results show that the readability impact was overall limited. However, it varies between the refactoring types and projects. We judged Extract Method as the most positive refactoring, because it replaced calculation logic or comparison logic with a clear method name, making the code easier to understand and

easier to test directly. Other refactorings, such as query methods, recording state, and hook methods, had a more mixed effect. They could make internal behavior more observable, but sometimes also made the production classes heavier by adding testing-oriented structure, which mixes production functionality with helper methods specifically added for testing. Thus, the approach did not strongly decrease readability, but the readability impact depends on the kind of refactoring that is applied.

In conclusion, LLM-guided refactoring can effectively increase production-code observability with only a limited impact on readability, but mainly when the targeted mutants point to behavior that is difficult to observe through the existing code structure. In those cases, refactoring gave the tests new ways to check the behavior. When the behavior was already observable through the public API, the increase mainly came from stronger tests instead of from improved production-code observability. Thus, the extent to which this happens depends on the targeted mutant, the refactoring technique, and whether the generated tests actually use behavior made observable by the refactoring. The results also suggest that the readability impact can stay limited when the refactoring exposes meaningful behavior without adding unnecessary testing-oriented helper methods.

In practice, this means that mutation-guided LLM refactoring should not treat every surviving mutant as an automatic reason to refactor, but also not as an automatic reason to add more tests. The mutation results are useful because they show concrete places where the current tests do not distinguish the original code from the mutated code. However, the main challenge is not necessarily increasing the mutation score, but deciding what kind of problem each surviving mutant represents. Sometimes the right response is a refactoring, but sometimes it is only a better test, a test double, or no change at all when the mutant is not behaviorally meaningful. In addition, generated tests still need to be inspected, because a passing test may contain a weak assertion that does not meaningfully check the intended behavior. The appropriate level of observability can also differ per mutant, which can range from checking execution to checking interactions or final behavior. Therefore, the approach is most useful when the LLM is guided not only to increase mutation score, but also to choose the smallest meaningful change that improves observability at the level needed for the specific context without adding unnecessary structure to the production code.

7.3 Future work

Based on the findings and limitations of this study, several directions for future work can be identified:

- As discussed in Section 5.5 under external validity, this study uses two open-source Java projects, which limits generalizability, but allowed us to go deeper into the results within the given timeframe. However, future work can evaluate the approach on more projects with different characteristics. This includes different project sizes, different kinds of functional code, different contexts, and possibly even other programming languages to make the generalizability stronger.

- The second direction is comparing workflows with and without an MCP server. This research cannot say whether the MCP server was valuable, but in our case, it was useful to give scoped mutation information to the LLM. However, the LLM can also access the mutation results in other ways. This includes, for example, using the full PIT report and letting the LLM agent open the file directly.

- The third direction is comparing different agent environments. In this study, we used Cline, but there are also other agent wrappers, such as Claude Code and Cursor. These tools can differ in file access, planning behavior, and how much control the user has. Future work can compare the same model and prompt across different environments.

- The fourth direction is separating the refactoring effects from the test-generation effects. In this study, to mimic real-world practice as much as possible, as in Schuberg Philis, we let the LLM both refactor the production code and generate the tests. However, if one wants to isolate the different effects more clearly, as discussed in internal validity in Section 5.5, future work could compare separate conditions. For example, the LLM could only refactor the production code, after which humans write the tests.

- The fifth direction is improving the prompt with a decision step. Instead of directly asking the LLM to refactor the code, the LLM could first classify whether the behavior is already observable and only needs stronger tests, whether it is not observable and needs refactoring, whether it can be better tested with a test double, or whether the mutant should be ignored because it represents an implementation detail or an invalid state.

- Lastly, future work could ask developers to review the refactorings and decide whether they would accept them. This connects to the pull request, since maintainers may reject observability helpers even if they help mutation testing. It also connects to object-oriented design principles [11], because some refactorings may increase observability, while they conflict with these design principles, such as encapsulation. Next to this, asking developers to review refactorings is also relevant for readability, because readability is based on human judgment, so a human evaluation would be useful.

Bibliography

[1] Duaa Alawad, Manisha Panta, Minhaz F. Zibran, and Md. Rakibul Islam. An empirical study of the relationships between code readability and software complexity. *CoRR*, abs/1909.01760, 2019. URL http://arxiv.org/abs/1909.01760.

[2] Eman Abdullah AlOmar, Luo Xu, Sofia Martinez, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. Chatgpt for code refactoring: Analyzing topics, interaction, and effective prompts. In *2025 IEEE International Conference on Collaborative Advances in Software and COmputiNg (CASCON)*, pages 389–398, 2025. doi: 10.1109/CASCON66301.2025.00068.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1 edition, 2008. ISBN 0521880386.

[4] Maurício Aniche. Ck: A java library for measuring code metrics. https://github.com/mauricioaniche/ck, 2026. Accessed: 10-04-2026.

[5] Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, 2024. Website. Accessed: 24-05-2026.

[6] Anthropic PBC. Model context protocol python sdk. https://github.com/modelcontextprotocol/python-sdk, 2024. Accessed: 05-05-2026.

[7] Association for Computing Machinery. ACM Code of Ethics and Professional Conduct. https://www.acm.org/code-of-ethics, 2018. Adopted by ACM Council on June 22, 2018. Accessed: 2026-05-31.

[8] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, June 2020. ISSN 1573-1367. doi: 10.1007/s11219-019-09477-y. URL https://doi.org/10.1007/s11219-019-09477-y.

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

[10] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 73–87, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132530. doi: 10.1145/336512.336534. URL https://doi.org/10.1145/336512.336534.

[11] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented analysis and design with applications, third edition*. Addison-Wesley Professional, third edition, 2007. ISBN 9780201895513.

[12] Carolin Brandt, Marco Castelluccio, Christian Holler, Jason Kratzer, Andy Zaidman, and Alberto Bacchelli. Mind the gap: What working with developers on fuzz tests taught us about coverage gaps. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 157–167, 2024. doi: 10.1145/3639477.3639721.

[13] Hongyu Pei Breivold, Ivica Crnkovic, and Peter J. Eriksson. Analyzing software evolvability. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 327–330, 2008. doi: 10.1109/COMPSAC.2008.50.

[14] Bukkit. StandardMessenger source code. https://hub.spigotmc.org/stash/projects/SPIGOT/repos/bukkit/browse/src/main/java/org/bukkit/plugin/messaging/StandardMessenger.java. Accessed: 2026-05-20.

[15] Bukkit. Bukkit. https://github.com/Bukkit/Bukkit, 2015. Version 1.7.9-R0.2.

[16] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. doi: 10.1109/TSE.2009.70.

[17] Nicholas Carlini. Building a c compiler with a team of parallel claudes. https://www.anthropic.com/engineering/building-c-compiler, 2026. Website. Accessed: 24-05-2026.

[18] Sivajeet Chand, Melih Kilic, Roland Würsching, Sushant Kumar Pandey, and Alexander Pretschner. Automated extract method refactoring with open-source llms: A comparative study, 2025. URL https://arxiv.org/abs/2510.26480.

[19] Jinsu Choi, Gabin An, and Shin Yoo. Iterative refactoring of real-world open-source programs with large language models. In Gunel Jahangirova and Foutse Khomh, editors, *Search-Based Software Engineering*, pages 49–55, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-64573-0.

[20] Mel Ó. Cinnéide, Dermot Boyle, and Iman Hemati Moghadam. Automated refactoring for testability. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 437–443, 2011. doi: 10.1109/ICSTW.2011.23.

[21] Cline Contributors. Cline documentation: Authorization. https://docs.cline.bot/getting-started/authorizing-with-cline, 2026. Documentation page. Accessed: 24-05-2026.

[22] Cline Contributors. Cline documentation: Plan and act workflow. https://docs.cline.bot/core-workflows/plan-and-act, 2026. Documentation page. Accessed: 16-04-2026.

[23] Cline Contributors. Cline: Autonomous coding agent for vs code. https://github.com/cline/cline, 2026. GitHub repository. Accessed: 17-04-2026.

[24] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2948707. URL https://doi.org/10.1145/2931037.2948707.

[25] Jonathan Cordeiro, Shayan Noei, and Ying Zou. An empirical study on the code refactoring capability of large language models, 2024. URL https://arxiv.org/abs/2411.02320.

[26] Jonathan Cordeiro, Shayan Noei, and Ying Zou. Llm-driven code refactoring: Opportunities and limitations. In *2025 IEEE/ACM Second IDE Workshop (IDE)*, pages 32–36, 2025. doi: 10.1109/IDE66625.2025.00011.

[27] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992. ISSN 1055-6400. doi: 10.1145/157710.157715. URL https://doi-org.tudelft.idm.oclc.org/10.1145/157710.157715.

[28] Al Danial. cloc: Count lines of code. https://github.com/aldanial/cloc, 2026. GitHub repository. Accessed: 15-04-2026.

[29] Amirhossein Deljouyi and Andy Zaidman. Generating understandable unit tests through end-to-end test scenario carving. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 107–118, 2023. doi: 10.1109/SCAM59687.2023.00021.

[30] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1449–1461, 2025. doi: 10.1109/ICSE55347.2025.00032.

[31] Natanael Djajadi. Mcp server for inspecting pit mutation testing results, June 2026. URL https://doi.org/10.5281/zenodo.20527491.

[32] Natanael Djajadi. Replication package for llm-guided refactoring using mutation testing, June 2026. URL https://doi.org/10.5281/zenodo.20528114.

[33] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. Improving source code readability: Theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 2–12, 2019. doi: 10.1109/ICPC.2019.00014.

[34] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, USA, 2004. ISBN 0131177052.

[35] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[36] G. Fraser and A. Andrea. EvoSuite: Automatic test suite generation for object-oriented software. SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13) Szeged, Hungary, September 2011. doi: 10.1145/2025113.2025179. URL https://www.researchgate.net/publication/221560749_EvoSuite_Automatic_test_suite_generation_for_object-oriented_software.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1994. ISBN 0-201-63361-2.

[38] Yossi Gil and Matteo Orrù. The spartanizer: Massive automatic refactoring. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 477–481, 2017. doi: 10.1109/SANER.2017.7884657.

[39] Google. Google scholar. https://scholar.google.com/, 2026. Website. Accessed: 31-05-2026.

[40] Google. Google sheets. https://workspace.google.com/products/sheets/, 2026. Website. Accessed: 31-05-2026.

[41] Isaac Griffith, Scott Wahl, and Clemente Izurieta. Truerefactor : An automated refactoring tool to improve legacy system and application comprehensibility. 2011. URL https://api.semanticscholar.org/CorpusID:17887724.

[42] Alex Grönholm. *AnyIO Documentation*, 2024. URL https://anyio.readthedocs.io/en/stable/. Version: 4.3.0 (Stable). Accessed: 05-05-2026.

[43] Henry Coles. Pit mutation testing: Basic concepts. https://pitest.org/quickstart/basic_concepts/, 2026. Documentation page. Accessed: 20-05-2026.

[44] Henry Coles. Pit mutation testing: Mutators. https://pitest.org/quickstart/mutators/, 2026. Documentation page. Accessed: 17-04-2026.

[45] Sascha Hunold, Björn Krellner, Thomas Rauber, Thomas Reichel, and Gudula Rünger. Pattern-based refactoring of legacy software systems. In Joaquim Filipe and José Cordeiro, editors, *Enterprise Information Systems*, pages 78–89, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-01347-8.

[46] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568271. URL https://doi.org/10.1145/2568225.2568271.

[47] ISO/IEC. Software engineering – Product quality – Part 1: Quality model. International Standard ISO/IEC 9126-1:2001, International Organization for Standardization, Geneva, Switzerland, June 2001.

[48] JFreeChart. JFreeChart. https://github.com/jfree/jfreechart, 2024. Version 1.5.6.

[49] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, 2011. doi: 10.1109/TSE.2010.62. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-80053606092&doi=10.1109%2fTSE.2010.62&partnerID=40&md5=c38f4e062de31deeff6089fef1a7000e. Cited by: 1360.

[50] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. An empirical study assessing source code readability in comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 513–523, 2019. doi: 10.1109/ICSME.2019.00085.

[51] Muhammed Abdulhamid Karabiyik. Refactorgpt: a chatgpt-based multi-agent framework for automated code refactoring. *PeerJ Computer Science*, 11:e3257, October 2025. ISSN 2376-5992. doi: 10.7717/peerj-cs.3257. URL https://doi.org/10.7717/peerj-cs.3257.

[52] M. M. Lehman. *Programs, Cities, Students— Limits to Growth?*, pages 42–69. Springer New York, New York, NY, 1978. ISBN 978-1-4612-6315-9. doi: 10.1007/978-1-4612-6315-9_6. URL https://doi.org/10.1007/978-1-4612-6315-9_6.

[53] M.M. Lehman. Program evolution. *Information Processing Management*, 20(1):19–36, 1984. ISSN 0306-4573. doi: https://doi.org/10.1016/0306-4573(84)90037-2. URL https://www.sciencedirect.com/science/article/pii/0306457384900372. Special Issue Empirical Foundations of Information and Software Science.

[54] Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. Making ai less 'thirsty'. *Communications of the ACM*, 68(7):54–61, June 2025. ISSN 0001-0782. doi: 10.1145/3724499. URL https://doi-org.tudelft.idm.oclc.org/10.1145/3724499.

[55] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. An empirical study on the potential of llms in automated software refactoring, 2024. URL https://arxiv.org/abs/2411.04444.

[56] Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of ai deployment? In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '24, page 85–99, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704505. doi: 10.1145/3630106.3658542. URL https://doi-org.tudelft.idm.oclc.org/10.1145/3630106.3658542.

[57] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

[58] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008. ISBN 0132350882.

[59] Bertrand Meyer. Applying "Design by Contract" . *Computer*, 25(10):40–51, October 1992. ISSN 1558-0814. doi: 10.1109/2.161279. URL https://doi.ieeecomputersociety.org/10.1109/2.161279.

[60] Seyed-Hassan Mirian-Hosseinabadi. Formal analysis of reachability, infection and propagation conditions in mutation testing, 2024. URL https://arxiv.org/abs/2410.21904.

[61] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. *On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension*, pages 173–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-76440-3. doi: 10.1007/978-3-540-76440-3_8. URL https://doi.org/10.1007/978-3-540-76440-3_8.

[62] Rajendar Namani. A new metric for code readability. *IOSR Journal of Computer Engineering*, 6:44–48, 01 2012. doi: 10.9790/0661-0664448.

[63] Khouloud Oueslati, Maxime Lamothe, and Foutse Khomh. Refagent: A multi-agent llm-based framework for automatic software refactoring, 2025. URL https://arxiv.org/abs/2511.03153.

[64] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938657. doi: 10.1145/1297846.1297902. URL https://doi.org/10.1145/1297846.1297902.

[65] Md. Masudur Rahman, Zenun Chowdhury, and Raqeebir Rab. A tool for measuring program comprehensibility using readability-driven metrics. *Software Impacts*, 25: 100782, 08 2025. doi: 10.1016/j.simpa.2025.100782.

[66] D. Rowe and J. Leaney. Evaluating evolvability of computer based systems architectures-an ontological approach. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, pages 360–367, 1997. doi: 10.1109/ECBS.1997.581903.

[67] G. P. Rusum. Large language models in IDEs: Context-aware coding, refactoring, and documentation. *International Journal of Emerging Technologies in Computer Science & Information Technology (IJETCSIT)*, 4(2):101–110, June 2023. doi: 10.63282/3050-9246.IJETCSIT-V4I2P110.

[68] Schuberg Philis. How we work. `https://schubergphilis.com/how-we-work`, 2026. Website. Accessed: 24-05-2026.

[69] Schuberg Philis. Lab271. `https://schubergphilis.com/how-we-work/lab-271`, 2026. Website. Accessed: 24-05-2026.

[70] Schuberg Philis. Our purpose, vision and mission. `https://schubergphilis.com/who-we-are/our-purpose-vision-and-mission`, 2026. Website. Accessed: 24-05-2026.

[71] Todd Sedano. Code readability testing, an empirical study. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 111–117, 2016. doi: 10.1109/CSEET.2016.36.

[72] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 151–160, 2023. doi: 10.1109/APSEC60848.2023.00025.

[73] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Better testing through oracle selection (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 892–895, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985936. URL `https://doi.org/10.1145/1985793.1985936`.

[74] Gábor Szőke, Csaba Nagy, Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. Faultbuster: An automatic code smell refactoring toolset. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 253–258, 2015. doi: 10.1109/SCAM.2015.7335422.

[75] Tiago Samuel Rodrigues Teixeira, Fábio Fagundes Silveira, and Eduardo Martins Guerra. Moving towards a mutant-based testing tool for verifying behavior maintenance in test code refactorings. *Computers*, 12(11), 2023. ISSN 2073-431X. doi: 10.3390/computers12110230. URL `https://www.mdpi.com/2073-431X/12/11/230`.

[76] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2012.12.052. URL https://www.sciencedirect.com/science/article/pii/S0164121213000022.

[77] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL https://arxiv.org/abs/2201.11903.

[78] Michael Whalen, Gregory Gay, Dongjiang You, Mats P. E. Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 102–111, 2013. doi: 10.1109/ICSE.2013.6606556.

[79] Whitelane Research. Netherlands 2025. https://whitelane.com/netherlands-2025/, 2025. Website. Accessed: 24-05-2026.

[80] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. In *Experimental Software Engineering Network*, 2003. URL https://api.semanticscholar.org/CorpusID:2710755.

[81] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. The rise and potential of large language model based agents: a survey. *Science China Information Sciences*, 68 (2):121101, 2025. doi: 10.1007/s11432-024-4222-0. URL https://doi.org/10.1007/s11432-024-4222-0.

[82] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 45(9):898–918, 2019. doi: 10.1109/TSE.2018.2809496.

[83] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. A systematic literature review of how mutation testing supports quality assurance processes. *Software Testing, Verification and Reliability*, 28(6):e1675, 2018. doi: https://doi.org/10.1002/stvr.1675. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1675. e1675 stvr.1675.

[84] Qianqian Zhu, Andy Zaidman, and Annibale Panichella. How to kill them all: An exploratory study on the impact of code observability on mutation testing. *Journal of Systems and Software*, 173:110864, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110864. URL https://www.sciencedirect.com/science/article/pii/S0164121220302545.

Appendix A

18 Candidate Projects

Table A.1: Candidate projects considered during project selection

Project / version	Source paper(s)	Maven	Further tested	Screening result
Apache POI	[46]	No	No	Excluded because Maven compatibility was required for the PIT-based setup.
Closure Compiler	[46]	Yes	Partly	Excluded because it was only partially compatible with the setup.
HSQLDB	[46]	No	No	Excluded because Maven compatibility was required for the PIT-based setup.
JFreeChart-1.5.6	[46, 84]	Yes	Yes	Selected for the case study. The project is Java-based, frequently used in mutation testing research, and has a large codebase and test suite.
Bukkit-1.7.9-R0.2	[84]	Yes	Yes	Selected for the case study. The project is Java-based, used in mutation testing research, and provides a smaller codebase for comparison with JFreeChart.
Joda-Time-v2.14.0	[46, 82]	Yes	Yes	Compatible with the setup, but not selected to keep the case study focused on a smaller number of projects.
Commons Lang-3.20.0	[84, 82]	Yes	Yes	Compatible with the setup, but not selected to keep the case study focused on a smaller number of projects.
Commons Math	[84]	Yes	No	Maven-compatible candidate, but not selected for further execution.
Java APNS-0.2.3	[84, 82]	Yes	Yes	Further tested, but not selected because the setup was unstable during execution.
PySonar2	[84]	Yes	Yes	Further tested, but not selected to keep the case study focused on a smaller number of projects.
Pushy	[84]	Yes	No	Maven-compatible candidate, but not selected for further execution.
Commons CSV-1.14.1	[75]	Yes	Yes	Compatible with the setup, but not selected to keep the case study focused on a smaller number of projects.
AssertJ	[82]	Yes	No	Maven-compatible candidate, but not selected for further execution.
Linear Algebra for Java	[82]	Yes	No	Maven-compatible candidate, but not selected for further execution.
MessagePack for Java	[82]	No	No	Excluded because Maven compatibility was required for the PIT-based setup.
UAA	[82]	No	No	Excluded because Maven compatibility was required for the PIT-based setup.
VRaptor	[82]	Yes	Yes	Further tested, but not selected because dependency compatibility issues prevented stable execution.
Wire Mobile Protocol Buffers	[82]	No	No	Excluded because Maven compatibility was required for the PIT-based setup.

Appendix B

Additional Mutation Results

This appendix provides grouped mutation results for the targeted mutants. The first part shows which mutants were targeted at the start of each run, grouped by run and mutation operator. The second part summarizes the outcome after refactoring, separating killed mutants from mutants that remained surviving. The full row-level data is provided in the replication package.

B.1 Targeted Mutants Selected at the Start of Each Run

Table B.1: Targeted JFreeChart mutants grouped by run and mutator

Run	Cond. Boundary	Math	Neg. Cond.	True Returns	Void Method Calls	Total
1	0	0	0	0	15	15
2	0	0	3	0	10	13
3	0	0	0	0	6	6
4	0	19	0	0	0	19
5	2	0	3	2	0	7
6	0	6	0	0	0	6
Total	2	25	6	2	31	66

Table B.2: Targeted Bukkit mutants grouped by run and mutator

Run	Empty	False	Math	Neg. Cond.	Null	Prim.	True	Void	Total
7	2	0	0	3	0	0	0	4	9
8	0	0	8	9	0	1	3	0	21
9	0	0	0	7	0	0	0	9	16
10	0	2	0	8	0	0	2	1	13
11	2	2	0	5	1	0	3	0	13
12	0	0	0	2	0	0	0	9	11
Total	4	4	8	34	1	1	8	23	83

B.2 Outcome of Targeted Mutants After Refactoring

Table B.3: Outcome of targeted JFreeChart mutants after refactoring

Run	Neg. Cond.	Void Calls	Killed	Total
1	0	15	0	15
2	3	6	4	13
3	0	0	6	6
4	0	0	19	19
5	0	0	7	7
6	0	0	6	6
Total	3	21	42	66

Table B.4: Outcome of targeted Bukkit mutants after refactoring

Run	Empty	Math	Neg. Cond.	True	Void	Killed	Total
7	2	0	0	0	0	7	9
8	0	5	3	0	0	13	21
9	0	0	7	0	4	5	16
10	0	0	3	1	0	9	13
11	0	0	0	0	0	13	13
12	0	0	2	0	2	7	11
Total	2	5	15	1	6	54	83

Appendix C

JFreeChart Experiment Logs

C.0.1 Project: JFreeChart

Experiment 1

Class: org.jfree.chart.plot.XYPlot

Mutation Result Explanation The mutation score increased from 49.86% to 52.44%, thus, having an increase of 2%. There are 24 more mutants killed, and 13 fewer surviving mutants. This is because by adding checks to determine whether specific operations were executed, the tests can detect differences in behavior and kill mutants in earlier negated conditionals.

Targeted Mutant Analysis

Mutant	Location	Result
Drawing Operations Mutants	Lines 2931, 2934, 2937, 2938, 2941, 2942, 2960, 2964, 2979, 2997, 3045, 3049, 3052, 3055, 3070	Survived (15)

Refactoring Analysis

What kind of refactoring was applied?
The LLM calls the refactoring applied Observable State Patterns with Drawing Operations Tracker. The very superficial closest refactoring is Encapsulate Variable [35], since we introduce a private field and expose it through a getter. But in Encapulate Variable, it applies to an existing private field we can't access directly. With the refactoring, we add getters and setters to make refactoring easier (since we can move/change data more easily), reduce coupling since code now depends on functions and not raw data, and let you add logic in one place. In this refactoring, however, the purpose is to increase observability to make sure functions are called.

Refactoring Target The LLM states that the `draw()` method contains many void method calls (for example `drawDomainTickBands` and `drawDomainGridlines`) whose execution could not be observed from outside. These internal operations were thus being mutated by PIT, but tests could not detect the changes.

LLM Reasoning Implicitly, the LLM argues that now the internal operations can be observed from the outside. However, the mutants are not killed. The tests could kill mutants in conditionals before the drawing calls, because changing the condition prevents the expected operation from being recorded. It is also interesting to note that it did not record all the operations, like the `drawDomainTickBands` on lines 2931 and 2934.

Behavior Changes There are no behavior changes.

Test Failures and Fixes The tests failed because the LLM was calling methods on `DrawingOperations` that do not exist. This was fixed by reading the `DrawingOperations.java` file, and changing, for example, `wasOperationRecorded` to `containOperation`.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests use valid inputs, but it does not check boundaries or edge cases. This is fine, because it is not necessary when the goal is to verify that the lines are not deleted. Thus, the tests are for triggering the specific setups.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
The test `testDrawRecordsPlotRenderingInfoOperations` is useful since it checks whether the tracker recorded that `setPlotArea` and `setDataArea` are called. This is stronger since this corresponds to a specific state of `PlotRenderingInfo`. `testDrawOperationResetOnEachDraw` is also worth keeping, since we make sure that the tracker state is always freshly made, so the tests are legitimate. The other tests are only to verify the operations were called, but not whether the rendered result is correct.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
It mainly checks the instrumented observable behavior, which means they just verify if internal drawing steps were recorded, rather than validating rendered output directly.

Conclusion of the Test Suite:
The tests use the new behavior to make sure internal drawing steps were recorded, but it does not validate the rendered output directly.

Main conclusion The refactoring made it possible to see which internal steps were executed, which helped mutants in conditionals. However, since the tests only check whether operations were called and not whether the final result is correct, most mutants still survive,

and the improvement is limited.

Readability

Class-level:
At class level, the change is almost negligible. The cyclomatic complexity and total methods increased by only 1 due to the addition of a new method with no internal control flow. The number of variables increased by one because of the added private field `lastDrawingOperations`, and the CBO also increased by one. This is because the class now depends on the `DrawingOperations` class.

Method-level:
At method level, the only change is that there are 11 more lines of code. In terms of the CoC, nesting, number of parameters, local variables, loop quantity, and CBO, they do not change.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: There is additional recording logic that the reader needs to realize is also for testing, however, they use clear method names.

2. Intent - Does the code better express what it does?
Neutral: It makes the operations explicit, but it now combines drawing logic with tracking logic

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Worsened: The introduction of a new state object, multiple calls, and getter, makes the reader need to go through them (even if it is quick) to understand what their role is

---

Experiment 2

Class: org.jfree.chart.plot.CategoryPlot

Mutation Result Explanation The mutation score increased from 54.68% to 55.97%, with an increase of 24 killed mutants and an increase of 1 survived mutant. The total amount of mutants, including no coverage, is also 1. This means with the tests, a lot of the previous `NO_COVERAGE` mutants are now killed, but also a lot survived, which explains the small difference in mutation score.

This was verified by removing the assertions from the generated tests. The number of killed mutants went up by only 3, and there are 22 more surviving mutants. This means the assertions are the ones to make the killings of mutants possible, however, it was not able to kill all of the surviving mutants.

Targeted Mutant Analysis

Mutant	Location	Result
Crosshair State Updates	Lines 3296, 3297, 3315, 3408, 3409, 3410, 3420, 3430, 3431	Survived (9)
Crosshair State Updates	Lines 3290, 3291, 3313, 3314	Killed (4)

Refactoring Analysis

What kind of refactoring was applied?
The LLM calls the technique Extract Observable State Pattern, with the explanation that it exposed internal crosshair state to enable testing of previously unobservable behavior.

Refactoring Target In the task summary, the LLM states it focused on 48 surviving mutants in the `draw()` function related to crosshair calculations:

- Crosshair state initialization and updates
- Range crosshair calculations (locked/unlocked on data)
- Domain crosshair key tracking
- Anchor point handling
- Distance calculations for crosshair snapping
- Dataset index tracking

However, during the Plan mode, it only targets 13 mutants, which are all related to the crosshair state updates.

LLM Reasoning The LLM reasoned that the crosshair state is updated internally, but the tests have no way to verify it. After refactoring, the tests can now query the crosshair state to verify it was correctly calculated based on the anchor point and data.

Behavior Changes There are no behavior changes.

Test Failures and Fixes The issue was that the new `lastCrosshairState` field should be ignored in the `equals`/`hasCode` verification since it is a transient runtime state, and not part of the plot's configuration. This was fixed by ignoring the field `lastCrosshairState` in `testEqualsHashCode` (wordings taken from the LLM during its process).

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests use representative, valid cases.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
Some tests could be worth keeping if the assertions are not too weak. An example is the second generated test, which is `testRangeCrosshairStateWithLockedOnData`. Theoretically, a mutant can break the crosshair computation and return a NaN, and the test will still pass. Another example is where tests would only check that a state object exists.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
It checks the observable state, but leans more toward the internal states than the behavior.

Conclusion of the Test Suite:
The test suite is mixed. The refactoring itself is defensible, because storing `lastCrosshairState` makes the result of the internal crosshair calculation observable after `draw()` finishes. This is similar to returning the computed state, but avoids changing the signature of an overridden `void` method. However, several tests use this new observability weakly. Some only check that the state object exists, and one test even allows `NaN` as a valid result.

Main conclusion The mutation score increased slightly because many previously uncovered mutants were executed, and some were killed. However, most crosshair-related mutants still survived. The refactoring made the internal state observable, but the tests are relatively weak because the assertions are lenient.

Readability

Class-level:
The change in readability is minimal. There is an increase in CoC, because of one added method with no internal control flow. The number of variables also increased by one, because of the added private field `lastCrosshairState`.

Method-level:
On method-level, the change is neglectable. The only change is in terms of the method length, which only increased by 1. This corresponds to the line `this.lastCrosshairState = crosshairState` on line 3468.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: Because there is only a one-line change, which is just saving the state, the effect is negligible.

2. Intent - Does the code better express what it does?
Neutral: The reader may ask themselves why `lastCrosshairState` is stored. However, because the change is small and clearly named, the effect on intent is negligible.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: Only a private field and a getter were added, so the codebase does not become much heavier to understand.

Experiment 3

Class: org.jfree.chart.plot.PiePlot

Mutation Result Explanation There is an increase in mutation score from 36.81% to 43.56%, with 43 more killed mutants, and 23 fewer survived mutants. This is because we can now have direct assertions on the `PiePlotState`.

Targeted Mutant Analysis

Mutant	Location	Result
VoidMethodCallMutators	Lines 2236, 2255-2258	Killed (6)

Refactoring Analysis

What kind of refactoring was applied?
The LLM changed `drawPie` return type from void to `PiePlotState`. This is called Extract Result pattern by the LLM, making internal computations observable by exposing them through return values rather than keeping them hidden in void method side effects.

Refactoring Target The LLM states that the VoidMethodCallMutator survivors indicate that `PiePlotState` is being updated internally during rendering, but these state changes are not observable to tests.

LLM Reasoning The LLM states that this exposes the internal state that was previously unobservable, allowing tests to verify that state-setting methods were actually called. By returning the state and asserting on its values, tests can now distinguish correct behavior (state values set properly) from mutated behavior (state values not set).

Behavior Changes No behavior changes.

Test Failures and Fixes The tests did not fail.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests use representative inputs.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
The tests could be worth keeping, because they verify the outcome of the `drawPie` method through its returned `PiePlotState`. This includes the computed center point, radii, and total.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The assertion checks use the observable behavior.

Conclusion of the Test Suite:
The tests assert on the method's returned result, and therefore check meaningful method behavior instead of only internal execution steps.

Main conclusion Refactoring `drawPie()` to return `PiePlotState` makes previously hidden state updates observable, allowing tests to detect mutants that could not be detected before.

Readability

Class-level & Method-level: The only readability change on both class-level and method-level is that there was a line added. This line is added on line 2294, where it returns the state at the end of the `drawPie` method.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: Because there is only a two-liner change, which is just changing `void` into `PiePlotState` and returning the state at the end, the effect is negligible. The method logic itself remains unchanged.

2. Intent - Does the code better express what it does?
Neutral: The method now returns `PiePlotState`, but this returned value is mainly used for testing and is not used elsewhere in the production code. Therefore, the change does not really make the production intent clearer.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: There are no extra code structures.

Experiment 4

Class: org.jfree.chart.plot.MeterPlot

Mutation Result Explanation The mutation score increased from 25.76% to 36.60%, with 29 more killed and 28 fewer surviving mutants. Because the LLM uses extract method, the tests can now directly verify the behavior.

Targeted Mutant Analysis

Mutant	Location	Result
Tick Position Calculations	Lines 1070-1081	Killed (19)

Refactoring Analysis

What kind of refactoring was applied?
The refactoring technique extract method was applied.

Refactoring Target The LLM states that `MeterPlot` has extensive surviving mutants in two main methods:

- `draw()`: 73 surviving mutants
- `drawTick()`: 46 surviving mutants

The LLM says also: "The surviving mutants reveal a critical observability problem: the code performs complex graphical calculations and drawing operations, but tests cannot observe the actual rendering results. The current test suite (MeterPlotTest.java) only tests:
- Property getters/setters - Equals/cloning/serialization - NO tests verify actual drawing correctness"

LLM Reasoning The LLM pasted as reasoning (verbatim):

> Before: Coordinate calculations were embedded in ‘drawTick()‘ method which performs graphics operations. Since graphics output cannot be easily tested, the 27 mathematical mutants in coordinate calculations survived (0% mutation score).
>
> After: By extracting ‘calculateTickEndpoints()‘ as a pure function that returns a testable value object, we can:
>
> 1. Directly test coordinate calculations without graphics context
> 2. Assert on exact coordinate values
> 3. Distinguish correct calculations from mutated alternatives

4. Achieve 100% mutation coverage for the mathematical operations

The refactoring demonstrates how Extract Method increases observability by separating testable logic (calculations) from untestable operations (graphics rendering), enabling tests to verify the correctness of previously unobservable internal behavior.

Behavior Changes There are no behavior changes.

Test Failures and Fixes The LLM used the wrong base angle for some `meterAngle` values. It guessed the geometry, but it did not use the formula in the code, which is `double baseAngle = 180 + ((this.meterAngle - 180) / 2.0)`. By checking the calculation of `valueToAngle`, it was able to correct the tests.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
Yes, the tests use lower bound, upper bound, midpoints, and normal rectangular meter areas.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
When the behavior of the source code is correct, the tests would be worth keeping since they test the behavior of the code well.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
They are useful when they check meaningful geometric behavior, but they become weak when the expected values are calculated by copying the implementation instead of verifying the behavior independently.

Conclusion of the Test Suite:
The tests check important behavior of the calculations, using both general checks and exact values. However, some of the stricter tests depend on the same formulas as the implementation, which makes them less independent.

Main conclusion The refactoring improved observability by extracting the coordinate calculations into a separate method. This allowed tests to directly verify the behavior, as reflected in the mutations core. However, some of the tests are a bit dependent on the source code.

Readability

Class-level:
At class level, the change is very small. Because of extracting the method, the total method quantity and the cyclomatic complexity both went up by 1. Since the LLM uses extract method, the change in lines of code is also very small (3), and because it introduces a new

class `TickEndPoints` `MeterPlot` now depends on, the CBO also increases by 1. All the other metrics stayed the same.

Method-level:
At method level, there is a decrease of 9 lines and 7 variables, which indicates readability went up on this level. All the other metrics stayed the same.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Improved: Because the coordinate calculation is moved out, the remaining method is shorter and easier to follow.

2. Intent - Does the code better express what it does?
Improved: `calculateTickEndpoints` clearly communicates its purpose instead of leaving it implicit in a long sequence of calculations without any comments.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: There is a slight increase because of the introduction of a helper method, but it is limited and contained.

Experiment 5

Class: org.jfree.chart.plot.PiePlot3D

Mutation Result Explanation The mutation score increased from 10.15% to 15.58%, with 11 more mutants killed, and 9 fewer surviving mutants. This is because the LLM used a combination of extract method and changing private to protected, thus having more targeted tests. This way, it can reach code that was originally unreachable.

Targeted Mutant Analysis

Mutant	Location	Result
Mutants in `isAngleAtFront`	Lines 960-962	Killed (3)
Mutants in `isAngleAtBack`	Lines 973-975	Killed (3)
Mutants in `getDarker`	Lines 594-596	Killed (1)

Refactoring Analysis

What kind of refactoring was applied?
Both extract method, as well as changing private to protected, were applied.

Refactoring Target and Reasoning (verbatim)

Phase 1: Extract and Test Angle Utilities (6 survivors)

Target: `isAngleAtFront()` and `isAngleAtBack()` methods (3 survivors each)

- Refactoring: Make these methods `public` instead of `private` (or `protected`)
- Why: These are pure mathematical functions that should be directly testable
- New tests: Test boundary cases (0°, 90°, 180°, 270°, 360°) and various angles to distinguish correct behavior from mutated alternatives

Phase 2: Make Paint Modification Observable (2+ survivors)

Target: `darkerSides` feature and `getDarkerSides()` method

- Refactoring: Extract `Paint applyDarkerSides(Paint original, boolean darker)` method
- Why: Currently the `darkerSides` feature is only tested via `equals()`, not functionally. The paint modification happens internally in `drawSide()` and isn't observable.
- New tests: Test that the method correctly darkens colors when enabled, returns original when disabled

Behavior Changes There are no behavior changes.

Test Failures and Fixes The LLM reasoned from a geometric intuition, rather than from the strict inequality in the code. So more specifically, angles of 0, 180, and 360 degrees where sin = 0 should return false for the `isAngleAtBack` function. This was fixed by changing the assertion of true to false. Thus, the boundary assertions were wrong. But the tests did not pass a second time for `isAngleAtBack` for 180 degrees, and `isAngleAtFront` for 360 degrees, and assumed sin(180 degrees) and sin(360 degrees) are both 0, so it asserted with false. However, in Java:

$$\sin(\text{rad}(180.0)) \approx 1.2246 \times 10^{-16} > 0$$

$$\sin(\text{rad}(360.0)) \approx -2.4493 \times 10^{-16} < 0$$

Thus, both return true. So either the LLM should have chosen between testing the code by changing it to `assertTrue`, or testing expectations of what the code should do, and thus changing the implementation. However, the LLM now chose neither and chose to assert where for every angle the assertions of `assertTrue(!result180 || !plot.isAngle-AtBack(180.0), "180 degrees should not be both at front and back");` and the lookalike assertions also pass. And no number satisfies this. The proof can be found below:

Proof. Assume there exists some angle $x \in \mathbb{R}$ such that both predicates return `true`. Then:

$$\sin(x) < 0 \tag{1}$$
$$\sin(x) > 0 \tag{2}$$

Combining (1) and (2), we obtain:

$$\sin(x) < 0 \ \wedge \ \sin(x) > 0.$$

This implies that $\sin(x)$ is simultaneously less than zero and greater than zero. However, for any real number $n \in \mathbb{R}$, it is impossible that

$$n < 0 \ \wedge \ n > 0.$$

By the trichotomy law, every real number is exactly one of the following: negative, zero, or positive. Therefore, it cannot satisfy both inequalities at the same time.

Hence, our assumption leads to a contradiction. Therefore, no such angle x exists. □

Remark. This result does not depend on the specific function $\sin(x)$. The argument holds for any real-valued function $f(x)$, since the contradiction arises solely from the structure of the inequalities:

$$f(x) < 0 \ \wedge \ f(x) > 0.$$

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests include boundary, typical, and edge cases relevant to angle logic and color transformation.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
The tests would be worth keeping, since they now test real behavior.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The assertions use the new observable behavior.

Conclusion of the Test Suite:
The tests check behavior that became visible through the refactoring, such as angle classification and color changes. These are meaningful checks that a developer would still care about, even without mutation testing. However, some specific assertions should be reviewed.

Main conclusion The refactoring improved observability and helped kill more mutants by making internal logic directly testable. But the test suite generated indicates that unclear expectations at boundaries can lead to weak assertions.

Readability

Class-level:
The change of readability metrics is negligible. The cyclomatic complexity went up by 1 because the LLM used extract method.

Method-level:
For the `drawSide` method, the cyclomatic complexity went down by 1 and has a decrease of 2 lines. This change is very small. For the other two methods `isAngleAtBack` and `isAngleAtFront`, there is no change in metrics at all.

Readability - Qualitative - Extract Method

1. Local clarity - Is the individual method logic easier to understand?
Improved: The paint modification is extracted into `getModifiedPaintForSides`, which makes `drawSide` slightly shorter and easier to follow.

2. Intent - Does the code better express what it does?
Improved: The new method clearly expresses that the paint may be modified for drawing sides, instead of this logic being embedded implicitly in `drawSide`.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: A new helper method was introduced, but it is small and focused, so the added structure is limited.

Readability - Qualitative - Visibility Change

1. Local clarity - Is the individual method logic easier to understand?
Neutral: The method logic is unchanged, since only the visibility is modified.

2. Intent - Does the code better express what it does?
Neutral: Changing the visibility does not affect how the method expresses its purpose.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: No additional code structure is introduced, only the visibility of existing methods is changed.

Experiment 6

Class: org.jfree.chart.axis.CategoryAxis

Mutation Result Explanation The mutation score went from 27.69% to 36.55%, with an increase of 24 more mutants killed, and 17 fewer survived mutants. By extracting the calculation methods, the LLM was able to make targeted tests for the extracted method.

Targeted Mutant Analysis

Mutant	Location	Result
Arithmetic mutants	Lines 970, 971, 978, 979, 989	Killed (6)

Refactoring Analysis

What kind of refactoring was applied?
The refactoring extract method of calculating the category label bounds was applied, in combination with a CategoryLabelBounds Value Object.

Refactoring Target The target of the LLM is the drawCategoryLabels, which it analysed has 25 survivors in total, of which they are primarily coordinate calculations and graphics operations. Because the LLM states that the surviving mutants indicate that internal calculations and graphics operations cannot be observed by the tests. Specifically

1. Coordinate calculations, which have complex arithmetic for positioning that happens internally but is not exposed
2. Intermediate values, because computed values like label positions are used internally, however, they are not exposed.

LLM Reasoning So the LLM then separates pure coordinate/position calculations from rendering code. The LLM will do so by creating methods that return coordinate data structures, which expose calculations for direct testing without graphics. And by introducing value objects through creating the class CategoryLabelBounds to capture calculated geometry, it is now possible to return from calculation methods for test verification. And this enables asserting on specific coordinate values.

Behavior Changes No behavior changes.

Test Failures and Fixes The tests did not fail.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests cover different edges, offset, and margins, which are relevant inputs.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
There is a mix of general checks (whether bounds are within the data area, for example) and specific numbers. If we assume the implementation does not change, then they are worth keeping.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The expected values are computed using the same logic as the method, so they follow the implementation more closely.

Conclusion of the Test Suite:
If we assume the current implementation is correct and does not change, the tests are valid.

Main conclusion Extracting the coordinate calculations and adding a value object made it possible to test logic that was previously hidden. This increased the mutation score by killing arithmetic mutants. But the tests mostly follow the same calculations as the code.

Readability

Class-level:
At class level, the change is also negligible. Because of extract method, the total number of methods went up by 1, thus, the cyclomatic complexity also went up by 1. Because we introduce a new class `CategoryLabelBounds`, the CBO also goes up by 1. The number of variables also went up by 1, because this new variable encapsulates the output of the extracted method.

Method-level:
There is a noticeable decrease in complexity, as the cyclomatic complexity decreased by 4, the lines of code by 27, and the number of variables by 3.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Improved: The coordinate calculation is moved out of `drawCategoryLabels`, so the drawing method becomes shorter and easier to follow.

2. Intent - Does the code better express what it does?
Improved: The new method `calculateCategoryLabelBounds` clearly shows that this part of the code calculates the bounds of a category label. This makes the purpose of the calculation more explicit.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: The refactoring adds a helper method and a new `CategoryLabelBounds` class. This adds more structure to the codebase, but the class is simple and directly related to the extracted calculation.

Appendix D

Bukkit Experiment Logs

D.0.1 Project: Bukkit

Experiment 7

Class: org.bukkit.plugin.messaging.StandardMessenger

Context The class manages the registration and routing of plugin messages by tracking which plugins send and receive on which channels and dispatching messages to the appropriate listeners.

Mutation Result Explanation The mutation score increased from 75% to 84.07%, an increase of 9.07%. There are 20 more killed, and 7 fewer survived. This is because of the added methods to expose internal map states, and being able to return the number of incoming registrations for either a channel or both a plugin and a channel, and adding focused tests.

By only adding the internal map states with their respective tests, the LLM was able to kill 12 more mutants, and there is one survived mutant more. By adding the registration count methods with their respective tests, the LLM was able to kill 4 more mutants, and there is a decrease of 4 surviving mutants. Lastly, by adding the focused tests, the total is 20 killed, and 7 less survived mutants.

Targeted Mutant Analysis

Mutant	Location	Result
Mutants in removeFromOutgoing method	Lines 48, 51, 59	Killed (3)
Mutants in getIncomingChannelRegistrations	Lines 328, 353	Survived (2)
Mutant in GetIncomingChannelRegistrations(channel)	Line 320	Killed (1)
Mutant in getIncomingChannelRegistrations(plugin, channel)	Line 337	Killed (1)
Mutant in isIncomingChannelRegistered(plugin, channel)	Line 378	Killed (1)
Mutant in isOutgoingChannelRegistered(plugin, channel)	Line 399	Killed (1)

The reason why the mutants in getIncomingChannelRegistrations survived is that, in theory, the mutant may be distinguishable through implementation-specific assertions, but not through behaviorally meaningful assertions, since both the original and mutated versions still return an empty set to the caller.

Refactoring Analysis

What kind of refactoring was applied?
There are two refactorings applied. The first one is a Map State Query Methods. The LLM explained that it exposed internal map state through three package-private query methods. There are already methods that determine, for example, whether a specific channel belongs to a plugin. However, there was no way yet to access the internal map state to be able to distinguish if a specific plugin or channel has already been deleted from the map. An example would be:

```
// 1. Register a plugin for a channel
messenger.registerOutgoingPluginChannel(myPlugin, "my:channel");

// 2. Unregister it
messenger.unregisterOutgoingPluginChannel(myPlugin, "my:channel");

// 3. Check via API
// returns false
messenger.isOutgoingChannelRegistered(myPlugin, "my:channel");
```

```
// returns empty set
messenger.getOutgoingChannels(myPlugin);
```

And at step 3, both can look identical, but both the following cases cannot be distinguished through the API directly:

```
// Scenario A - key fully removed (correct cleanup):
outgoingByPlugin = {}

// Scenario B - stale key with empty set (bug/leak):
outgoingByPlugin = { myPlugin -> [] }
```

The second refactoring is adding registration count methods that return integers before converting to immutable sets. However, while this refactoring can make sense theoretically, it was not able to kill the mutants it was trying to target. This refactoring was not able to distinguish the mutant that changes `ImmutableSet.of()` into `Collections.emptySet`. Next to this, the registration count was redundant, because if one really wanted to target the mutant that alters the `registrations != null` and thus check whether we receive back a certain amount of registrations or because of the mutant an empty set, it would have been possible to use:

```
messenger.registerIncomingPluginChannel(plugin1, "test-chan", listener1);
messenger.registerIncomingPluginChannel(plugin2, "test-chan", listener2);

Set<PluginMessageListenerRegistration> registrations =
    messenger.getIncomingChannelRegistrations("test-chan");

assertEquals(2, registrations.size());
```

Lastly, it also strengthened the validation testing by adding eight new tests using invalid inputs. This includes mainly ChannelNameTooLongException and IllegalArgumentException. However, this did not alter any source code.

Refactoring Target For the first refactoring, the targets were the mutants in the removeFromOutgoing method. The reasoning is because the tests could not observe whether internal maps are properly cleaned up after removal. For the second refactoring, the LLM stated the tests could not distinguish "truly empty" from "mutated to return emptySet()" when methods return empty sets. And it wants to assert on when registrations exist, then the count should be bigger than 0. It states that without the extra tests, the current test suite could not detect when validation was bypassed. However, these mutants were already killed earlier by the method `testGetIncomingChannelRegistrations_String` on line 287. However, it also stated it wanted to target line 328 and 353, which are the mutants that change `ImmutableSet.of()` into `Collections.emptySet`.

LLM Reasoning By adding the map state query methods, it is now able to observe whether internal maps are properly cleaned up after removal. By adding the registration count methods and checking if the count is bigger than zero, the tests can verify registrations exist before checking set contents. By adding tests, it was able to detect when validation was bypassed.

Behavior Changes There is a functional behavior change in the removeFromOutgoing function. The generated tests show that there is a bug in the code. Specifically, `StandardMessengerTest.testRemoveFromOutgoing_LastPluginCleansUpChannelMap` shows the bug clearly. When there is one channel and two plugins, in the test, we remove one plugin from the channel, and after this, the channel map should still contain the second plugin. However, this test fails because in the original code, when the plugin's set of channels becomes empty, the code incorrectly removes the channel from the channel map instead of removing the plugin from the plugin map. Thus, this behavioral change was a bug, and the LLM was able to fix it.

Test Failures and Fixes The first failure was because the LLM was using the custom assertEquals method, which is for Collections, but it needed to use JUnit's assertEquals for integer comparisons. The second test failure was because of a bug, which was fixed in the source code. The third failure was because of a missing import for Set, which was added. Lastly, the LLM used the String.repeat() method, which does not exist in Java 8. It fixed it with a for loop.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
Yes, this includes removing the first and last plugin or channel, unregistering a channel that never registered before, and invalid channel names. The tests do not use arbitrary values, but use relevant names and cases.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
The tests are worth to keep, like the cleanup behavior and validation tests for invalid channel names.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
They check observable behavior. They are partly implementation-focused, since they verify that internal maps are cleaned up correctly, and not only that the public registration result is correct.

Conclusion of the Test Suite:
So there is evidence for observability improvement, mainly for the first refactoring. However, at least eight of the tests are tests to specifically kill mutants through invalid inputs.

There is also something worth noting. In particular, when looking at the test that the LLM was trying to target on line 328 of the `getIncomingChannelRegistrations` method.

The LLM wanted to distinguish truly empty from mutated to return emptySet(). It uses the test `testGetIncomingChannelRegistrations_String_DistinguishFromEmptyMutant`, but this test would kill the mutant that changes the if statement of `registrations != null` into `registrations == null`. However, the other test `testGetIncomingChannelRegistrations_String_EmptyChannel` would have been able to distinguish `Collections.emptySet` from `ImmutableSet.of` by asserting implementation-specific properties of the returned set. The downside is that such a test would be tightly coupled to the current implementation rather than to the intended behavior of the method. Since both return an empty set, the mutant is not distinguishable through behaviorally meaningful assertions.

Main conclusion The LLM added helper methods to make hidden internal states directly observable for tests. However, the second phase was not necessarily needed, and the respective tests did not target the mutants it initially wanted to target. Lastly, the LLM added new tests to kill surviving mutants by using invalid inputs.

Readability

Class-level:
At class level, one can see that the cyclomatic complexity went up by 10, with 5 more methods, 4 new variables, and one more loop. This total increased the lines of code by 37. This can indicate that the readability may be negatively affected, because a higher number of cyclomatic complexity is usually correlated with lower readability.

Method-level:
At method level, the targeted methods `getIncomingChannelRegistrations` and `removeFromOutgoing` do not have any change in terms of readability metrics.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: There is no change to one individual method. The added helper methods are short and easy to understand, but they do not make the original targeted methods easier to understand.

2. Intent - Does the code better express what it does?
Neutral: The added method makes internal cleanup more explicit, but are only used for testing. So now there is a mix of responsibilities in the class. However, the comments help with understanding that these methods are only for testing.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Worsened: Because the refactoring added several methods, together they add more code and make the class a bit heavier to understand (since there is now more to read and understand).

Experiment 8

Class: org.bukkit.plugin.messaging.PluginMessageListenerRegistration

Mutation Result Explanation The mutation score increased by 41.94%, with a difference of 30 more killed, and 13 fewer survived. Because of extracting methods, the internal comparison and hashing logic in the `equals()` and `hashCode()` methods can be directly callable and testable. Before the refactoring, tests only see the final boolean (`equals()`) or int (`hashCode()`) results, and not intermediate steps. By adding these separate package-private methods, the tests are able to make the internal decision points more testable, because they can verify each field comparison or hash contribution independently, and enable targeted assertions. There is also an increase in the total amount of mutants, because extract method introduces additional code that can be mutated. An example in this case is the boundary point with the if statement.

Targeted Mutant Analysis

Mutant	Location	Result
Mutants in `hashCode()`	Lines 98-102	Out of the 13 initial survived, 5 survived, and 8 are killed
Mutants in `equals()`	Lines 80-90	Out of the 8 initial survived, 3 survived, 5 killed

In the `equals()` method, however, there was a total of 12 surviving mutants (survived and no coverage), and a total of 8 more were killed. This can be seen in Table D.1. Out of the four no-coverage mutants, three of them are killed. The reason there are still surviving mutants on the original lines 80, 83, and 89 is that the test does not have a null check. For example, by adding the following test, the mutant on the original line 80 can be killed. However, the constructor of the `PluginMessageListenerRegistration` does not allow one of the fields to be null.

```
@Test
public void testMatchesMessenger_OneNull() {
    PluginMessageListenerRegistration reg1 =
        new PluginMessageListenerRegistration(null, getPlugin(), "c",
        getListener());
    PluginMessageListenerRegistration reg2 =
        new PluginMessageListenerRegistration(getMessenger(), getPlugin(),
        "c", getListener());

    assertFalse(reg1.matchesMessenger(reg2));
}
```

In the `hashCode()` method, the mutants that are still alive have to do with the mutants changing addition into subtraction. Because the tests use `assertEquals` on the same registration, or the tests are using `assertNotEquals` on different registrations. When addition changes into subtraction, then for the same registration, it gives the same output, or for different registrations, it still gives a different output. Thus, the tests cannot differentiate between the two.

The multiplication-to-division mutants were mostly killed because changing multiplication into division has a stronger effect on the hash calculation. In the later lines of `hashCode()`, the intermediate hash value is already larger than 53. Because Java uses integer division, `53 / hash` then becomes 0. As a result, earlier parts of the hash calculation are removed. This can make two different registrations produce the same hash code, so the `assertNotEquals` tests can detect the mutant.

The first multiplication-to-division mutant is different. At that point, the intermediate hash value is still only 7. Therefore, `53 / 7` still gives a non-zero value. The hash code is still calculated wrongly, but the messenger contribution is not fully removed. Because the tests do not check the exact hash-code formula, this mutant can still survive.

Line	#Mutants	Before Refactoring			After Refactoring		
		Survived	No Coverage	Killed	Survived	No Coverage	Killed
80	3	1	2	0	1	0	2
81	1	0	1	0	0	0	1
83	3	3	0	0	1	0	2
84	1	1	0	0	0	0	1
86	3	0	1	2	0	1	2
87	1	1	0	0	0	0	1
89	3	1	0	2	1	0	2
90	1	1	0	0	0	0	1

Table D.1: Mutation Testing Results Before and After Refactoring for Lines 80-90

Refactoring Analysis

What kind of refactoring was applied?
Extract method was applied for both the `equals()` and `hashCode()` methods to extract internal comparison and hashing logic into separate observable methods, respectively. Both refactorings produced four new methods.

Refactoring Target The targets were specifically, firstly, all mutants from line 98-102 of the `hashCode()` method, because of math mutations, negated conditionals on null checks, and return value mutation, which means returning 0 instead of the computed hash. Secondly, the LLM targeted the mutants on lines 80-90, which are negated conditionals on null checks and field comparisons, and boolean return mutations, which means returning true instead of false.

LLM Reasoning The LLM stated, "The refactoring increases observability by making previously internal, untestable behavior directly observable through extracted methods, enabling comprehensive testing that distinguishes correct from mutated behavior".

Behavior Changes There are no behavior changes.

Test Failures and Fixes The tests did not fail.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests use valid and targeted inputs that distinguish the newly exposed decision logic. For the `equals()` method, there are positive and negative cases for each extracted comparison, and for the `hashCode()` method, there are tests for the individual hash contributions.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
In particular, the public `equals()` and `hashCode()` contract tests are worth keeping. The other tests could also be worth keeping, depending if these methods are also still used in the future. Judging by the straightforwardness of these methods, these are also worth it.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The assertions make use of the newly observable behavior.

Conclusion of the Test Suite:
The tests are straightforward and check both the extracted methods and the original tests. This shows that the tests can more easily verify the behavior.

Main conclusion By extracting methods from the `equals()` and `hashCode()` methods, internal decision points that were more difficult to test are now more exposed and can be tested directly. There are surviving mutants of the original targeted mutants. However, because of the constructor, it would not be possible to create tests to check the null fields.

Readability

Class-level:
At class level, only looking at the metrics can indicate that the readability has gone down. There are in total 8 new methods, the cyclomatic complexity went up by 12, and the lines of code went up by 36.
Method-level:
At method level, we can only see that the cyclomatic complexity went down by 6 for the `equals` method, and for `hashCode` it went down by 4. This could indicate locally that the readability increased a bit.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Improved: The `equals` and `hashCode` methods are shorter and easier to follow. For the `equals` method, it is immediately clearer that we match the messenger, plugin, channel, and listener in one word than having to go through the line. For `hashCode` it is even a bigger change, since now we can see the complex calculation is the contribution of, for example, the messenger.

2. Intent - Does the code better express what it does?
Improved: The code now expresses better what it does, since the method names make the checks and calculations more explicit.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Neutral: While there are many small helper methods, they are small and contained.

Experiment 9

Class: org.bukkit.configuration.MemorySection

Mutation Result Explanation The mutation score increased from 90.34% to 93.91%, with a total of 29 mutants killed and 6 less survived mutants. The mutants that the LLM wanted to target, as shown in Table D.2, are not killed. Combined with the results of the number killed and fewer survivors, this means that 23 mutants were introduced and then killed. This was verified, where all, except one of the newly introduced methods, had 2 mutants introduced and killed. That specific method was `isValidNonEmptyPath`, and had three newly introduced mutants instead of 2. Mutants that were killed were on lines 137, 163, 202, 234, 763 (now 887), and 779 (903).

Targeted Mutant Analysis .

Mutant	Location	Result
Type Check Observability	Lines 296, 326, 341, 357, 620, 650, 665	Survived (7)
Validation Observability	Lines 59, 60, 66, 701	Survived (4)
Validation Observability	Lines 137, 163, 202, 234, 779	Killed (5)

Table D.2: Overview of mutation testing results

The mutants that were killed have in common that they are all removed calls as mutants, like `Validate.notNull(path, "Path cannot be null")`. However, these calls are observable even before the refactoring and can be killed by just adding a test without

the newly added methods. An example would be line 137, which is killed with the test `testAddDefaultWithNullPathThrowsException`, which is a test where it passes a null as input and expects an `IllegalArgumentException`.

The mutants that are alive are either not directly tested as line 137, which are the survived Validation Observability mutants, which are also calls to `notNull`, or because the tests do not exercise the conditional that depends on `getDefault(path)`. The tests either call the overloaded `getInt(path, def)` method directly, or provide a value for the path. This causes the default value to be ignored. Thus, both the original and mutated behavior produce the same observable outcome, and the mutant remains undetected. An example can be seen below. Since both the original and mutated behavior result in the same observable outcome, mutants in this branch remain undetected.

```
public int getInt(String path) {
    Object def = getDefault(path);
    return getInt(path, (def instanceof Number) ? toInt(def) : 0);
}
```

Refactoring Analysis

What kind of refactoring was applied?
The LLM calls the refactorings "Extract Type Validation Methods" and "Extract Validation State Methods". This means the LLM added protected methods to check whether the object stored in the `MemorySection` is a specific type or whether, for example, the path is valid.

These added methods are largely based on logic that was already present in the original code, but embedded within the methods. An example is `isValidIntType`, where similar type-checking logic is already present in the `getInt` method. However, in the original method, that part is more deeply nested. Another example is on line 137, where the original uses `Validate.notNull(path, "Path cannot be null"` and the added helper method is `isValidPath()` which exposes this validation logic by returning `path != null`.

Refactoring Target The refactoring targets are listed in Table 2. The LLM argues the Type Check Mutants survive because tests can't distinguish when type checking logic is inverted. For the Validation Observability, the LLM argues that they survive because the tests never pass invalid inputs to observe validation behavior.

LLM Reasoning For the Type Validation Methods, the LLM stated that by making `instanceof` checks observable, it enables tests to verify type validation logic directly. For the Extract Validation State Methods, the LLM stated that by making null or empty validation checks observable, this enables tests to verify that validation logic works correctly.

Behavior Changes There are no behavior changes.

Test Failures and Fixes The tests did not fail.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests use behaviorally meaningful inputs to distinguish valid and invalid cases. The invalid cases include the tests that expect an `IllegalArgumentException`.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
They would be worth it, since they test the actual behavior of the code.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The assertion checks observable behavior, but only the behavior that was already observable before the refactoring.

Conclusion of the Test Suite:
The tests are good tests, but they do not test anything that was previously unobservable.

Main conclusion The refactoring added protected methods that are based on logic already present in the original code. It did not increase observability of the original behavior itself, but instead added separate methods that expose parts of the existing logic. While the original behavior was already indirectly observable, like the `Validate.notNull()` calls, the previous test suite did not provide inputs that distinguished these cases. By introducing the new methods, the tests are able to target these decisions more directly, which contributes to killing additional mutants. Furthermore, the extraction also introduces new mutation points that are easier to test in isolation.

Readability

Class-level:
At class level, there is an increase of 11 methods, an increase of cyclomatic complexity of 15, an increase of 7 variables, and an increase of 40 lines of code. This could indicate that readability went down.
Method-level:
At method level, there are no changes to the target methods in terms of metrics.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: There is no change to one individual method. The added helper methods are short and easy to understand, but do not make the original targeted methods easier to understand.

2. Intent - Does the code better express what it does?
Neutral: The method names make the checks explicit, but they mostly restate very simple

conditions. And the class now has a mix between functions that are for its functionality and for its testing.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Worsened: Multiple small helper methods that are added mainly for testing, which makes the class heavier without adding meaningful new behavior.

Experiment 10

Class: org.bukkit.command.SimpleCommandMap

Mutation Result Explanation The mutation score increased from 0.00% to 80.65%, with 25 mutants killed, and 9 fewer mutants survived. Only adding the tests without the newly introduced methods increases the mutation score from 0.00% to 73.68%, with 14 more killed and 10 less survived mutants.

Targeted Mutant Analysis

Mutant	Location	Result
Constructor Mutant	Line 27	Killed (1)
Return Value Mutants	Lines 96	1 Killed, 1 Survived
Conditional Mutants	Lines 108, 115, 122, 149, 153	Killed (6)
Conditional Mutants	Lines 109	Survived (1)
Conditional Mutants	Lines 138	1 Killed, 2 Survived

If the newly introduced methods are commented out with their respective lines in the test cases, line 27 will still survive. All the other results stay the same.

Refactoring Analysis

What kind of refactoring was applied?
The LLM calls the refactoring technique “Add Query Methods”. It introduces four new methods that make specific properties of the internal known Commands registry directly observable, such as label presence, primary-label mapping, total registry size, and label conflicts.

Refactoring Target The targets of the LLM were the mutant in the constructor on line 27, the return value mutant on line 96, and several conditional mutants between lines 108 and 153. The LLM argued these surviving mutants exist because internal state and decision-making logic cannot be observed from outside the class. This includes that the `knownCommands` map is private with no public getter, and when conditionals are negated,

the tests cannot detect the difference because the final observable behavior is the same. The reason the first mutant survived is because the tests always check only a successful registration, and not where it returns false. And thus, the mutant that changes the boolean condition to true always passes. The second mutant is still alive because the alias is still added to the command map before it is removed from the command's alias list. The current test only checks the command map, so it does not notice that the alias list was changed incorrectly. A simple fix would be to check the command's own alias list also. The last two surviving mutants survive because, a few lines after line 138, there is also another check that rejects the registration, even with the mutant on line 138. Thus, in the original code, line 138 rejects it, and it returns false. In the mutated code, line 138 does not reject it, but still on line 149, it rejects it and still returns false. Thus, the tests do not see a difference in behavior. This can be fixed with a more specific test.

In more detail, the test does this:

```
TestCommand regularCmd = new TestCommand("regularcommand");
commandMap.register("mycommand", "plugin", regularCmd);

TestVanillaCommand vanillaCmd = new TestVanillaCommand("vanilla");
boolean result = commandMap.register("mycommand", "bukkit", vanillaCmd);
```

And thus before registering `vanillaCmd`, the map already contains `mycommand -> regularCmd`. Then the test tries to register a `VanillaCommand` under the same label where `mycommand -> vanillaCmd`. In the original code, the values are `label = "mycommand"`. `command = vanillaCmd, isAlias = false, fallbackPrefix = "bukkit"`. So when we look at the first condition, it becomes

```
if ((command instanceof VanillaCommand || isAlias)
    && knownCommands.containsKey(label)) {
        return false;
}

if ((true || false) && true) {
    return false;
}
```

And this returns false. So in the original code, this condition rejects `VanillaCommand` because "mycommand" already exists. But with the mutant, it becomes `(false || false) && true` becoming false. But later on, conflict is not null, and thus it still returns false. The same happens with changing isAlias.

LLM Reasoning The LLM states that this approach increases observability by exposing internal state through query methods, enabling tests to distinguish correct behavior from mutated behavior from mutated alternatives without changing the class's core functionality.

Behavior Changes There are no behavior changes.

Test Failures and Fixes The tests did not fail.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
Yes, some tests reflect the real behavior, but other tests were mainly created to inspect internal state with the new helper methods, which do not check whether the behavior of the code is correct in a broader sense. Examples are `testRegisterCommandReturnsCorrectValueAndUpdatesState` and `testRegisterWithLabelReturnsCorrectValueAndUpdatesState`, because these tests mainly check the internal state. An example of a test that checks behavior is `testFailedRegistrationUpdatesLabelCorrectly`, because it checks the outcome of a conflict.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
Some of the tests that are more coupled with the implementation may not be worth it because it will be harder to change in the future, but the behavioral ones are valuable.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The tests use the newly observable behavior through the introduced methods.

Conclusion of the Test Suite:
The tests do show that the refactoring makes internal behavior more observable, but a significant part of the test suite is mainly useful for the mutation testing than long-term, implementation-dependent validation.

Main conclusion The LLM introduced methods to make specific properties of the Commands registry more observable. However, by commenting out the newly introduced methods with their respective lines, the LLM was still able to kill the mutants it wanted to target, except the constructor mutant on line 27. This indicates that most of the mutation score improvement came from the added tests themselves, not from the refactoring. Therefore, the refactoring had only limited added value for this experiment.

Readability

Class-level:
At class level, the LLM added 4 new methods, and this increased the cyclomatic complexity by 9, the number of variables and number of loops by 1, and the lines of code by 21. This could indicate a small decrease in readability.

Method-level:
At method level, there are no changes at all for the metrics for `SimpleCommandMap` and `register`.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: There is no change to one individual method. The added helper methods are short and easy to understand, but do not make the original targeted methods easier to understand.

2. Intent - Does the code better express what it does?
Neutral: The method names make the checks explicit, but the class now has a mix between functions that are for its functionality and for its testing.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Worsened: Multiple small helper methods that are added mainly for testing, which makes the class heavier without adding meaningful new behavior.

Experiment 11

Class: org.bukkit.command.Command

Mutation Result Explanation The mutation score increased from 0.00% to 100.00%, with 22 more killed mutants and 12 less survived mutants. Three of the killed mutants can be attributed to the three introduced methods, where every method adds one more killed mutant.

Targeted Mutant Analysis

Mutant	**Location**	**Result**
isRegistered() survivors	256	Killed (2)
allowChangesFrom() survivors	218-223	Killed (3)
register() survivors	247	Killed (2)
setLabel() survivors	202, 204	Killed (2)
setAliases() survivors	307, 310	Killed (2)
getName() survivors	109	Killed with only tests (1)
getAliases() survivors	265	Killed with only tests (1)

Refactoring Analysis

What kind of refactoring was applied?
The LLM did not give a name to the refactoring applied, but it just stated what it did as the refactoring technique. The three explanations are:

1. Add a `getCommandMap()` method that returns the registered CommandMap

2. Add a `getNextLabel()` method to expose the pending label
3. Add a `getConfiguredAliases()` method to return the `aliases` field

Refactoring Target The targets of the three refactorings are listed below in the same order as the explanations of what kind of refactorings are applied

1. `isRegistered()`, `allowChangesFrom()`, `register()` survivors
2. `setLabel()` survivors (lines 202, 204)
3. `setAliases()` survivors (lines 307, 310)

Next to that, the LLM also targets `getName()` empty string mutant, `getAliases()` empty list mutant. Tests that verify non-empty name returns non-empty string, and non-empty aliases return non-empty list. This was verified, and they were reasonable choices. Refactoring these one-line methods that simply return a name or a list of strings would be unnecessary since their behavior is already clear and easily verifiable through tests.

LLM Reasoning The reasoning of the LLM is shown below (verbatim), corresponding to the order of the explanations of what kind of refactorings are applied.

1. Currently tests can only check if some CommandMap is registered via `isRegistered()`, but can't verify which one. This allows mutants that return wrong boolean values or negate conditionals to survive. Tests can verify `command.getCommandMap() == expectedCommandMap`.
2. When a command is registered, `setLabel()` sets `nextLabel` but not `label`. Tests can't observe this distinction, so mutants that negate the conditional or return false survive. Tests can verify `command.getNextLabel().equals("newLabel")` vs `command.getLabel()`.
3. `getAliases()` returns `activeAliases`, but `setAliases()` sets the `aliases` field. When registered, these differ, and tests can't observe this. Tests can distinguish between configured and active aliases.

However, by commenting out the newly introduced methods and the respective assertions in the tests that use those methods, the mutation score stays the same. This indicates that the newly introduced getter methods and the assertions depending on them are not necessarily needed to kill the mutants, but the LLM's goal was to increase observability. An example is the `setLabel(String name)` function. `this.nextLabel` is unobservable outside the method. A mutation that removes or alters this assignment would therefore not be detected by tests. By introducing `getNextLabel()`, tests can directly verify whether the internal state was updated correctly, allowing such mutants to be killed. Also, the claims of "this allows mutants that return wrong boolean values or negate conditionals to survive", or any of the implications are not true. It may be true that the original API could not directly expose the fields like `CommandMap`, but it does not follow that the mutants survived because

of that. Also, it can be useful to know which `CommandMap` a command is registered to, because the class does not only store whether a command is registered but also which map is allowed to modify it. Checking only `isRegistered()` is less precise, because it only shows that some map is registered. However, in this experiment, this extra getter may not been necessary. The same rule could already be tested through the existing public behavior. This can be seen in the example below:

```
    public boolean register(CommandMap commandMap) {
    if (allowChangesFrom(commandMap)) {
        this.commandMap = commandMap;
        return true;
    }

    return false;
}
```

```
    assertTrue(command.register(commandMap1));
    assertFalse(command.register(commandMap2));
    assertTrue(command.unregister(commandMap1));
```

Behavior Changes There are no behavior changes.

Test Failures and Fixes There is one test failure, where the `CommandMap` mock class did not follow the exact signature of the dispatch method in the `CommandMap` interface. This was quickly fixed by updating the signature to throw a `CommandException`.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
The tests use behaviorally meaningful inputs and checks even realistic usage of the API, like the `testRegisterReturnsFalseWhenAlreadyRegistered`.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
These tests can still be worth keeping, since, as stated before, some checks meaningful behavior, like for the registration behavior, it prevents re-registration to another map. Some tests are more coupled to the current implementation with direct checks of internal fields, and depend more on how the class is implemented, not what it does.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The assertion checks use the newly observable behavior.

Conclusion of the Test Suite:
The tests are using the newly introduced getter methods well to check the internal state of variables. However, these tests are also, as a consequence, more coupled to the current implementation.

Main conclusion The LLM added getter methods, which increase the internal observability of the methods. These, together with the assertions that depend on them, were not necessarily needed to increase the mutation score directly, but to increase the observability.

Readability

Class-level:
At class level, the LLM introduced 3 new methods, and the cyclomatic complexity increased by 3. There is also an increase of 9 in terms of lines of code. Because this change is small, this could indicate that there is not a big impact on readability.

Method-level:
At method level, there are no changes at all for the metrics for `allowChangesFrom`, `isRegistered`, `register`, `setAliases`, and `setLabel`.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Neutral: There is no change to one individual method. The added helper methods are short and easy to understand, but do not make the original targeted methods easier to understand.

2. Intent - Does the code better express what it does?
Neutral: The method names make the checks explicit, but the class now has a mix between functions that are for its functionality and for its testing.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Worsened: Multiple small helper methods that are added mainly for testing, which makes the class a tiny bit heavier without adding meaningful new behavior.

Experiment 12

Class: org.bukkit.configuration.file.FileConfiguration

Mutation Result Explanation The mutation score increased from 31.25% to 83.33%, with 15 more killed mutants and 7 fewer surviving mutants. The refactoring technique used (see below) introduced moved the original mutations to a new method, and introduced new mutants in the same place.

Targeted Mutant Analysis

Mutant	Location	Result
Validation calls	99, 130, 165, 248	Killed (4)
File system operations	101	Killed (1)
File system operations	132	Survived (1)
I/O operations	108, 110	Killed (2)
I/O operations	223	Survived (1)
Encoding logic	105, 169	Survived (2)

Refactoring Analysis

What kind of refactoring was applied?
The LLM calls the refactoring "Template Method Pattern". It extracted internal operations into protected hook methods that the tests can use.

Refactoring Target It targeted validation calls, file system operations, I/O operations, and encoding logic mutants. The LLM states that these mutants survive because internal operations occur as side effects with no observable state exposed for testing.

LLM Reasoning By adding Template Method Pattern, the LLM states

- It maintains backward compatibility, because there are no changes to the public API
- It increases observability, since internal operations become trackable
- It enables targeted testing, because tests can verify specific operations
- It follows SOLID principles, since it enables single responsibility

As also written in the output of the LLM, it states that it killed all the mutants on the original lines, which is correct. However, if we analyse the original code with its respective mutants (because the original mutants just got moved to a new function), which is shown in the table, some mutants are still surviving. The LLM also argues why this is the case. It states that hook methods are simple delegations, and testing their internals would require complex mock verification. And for the `selectCharsetForSave/Load()`, it states that charset conditionals may be system-independent.

However, saying that testing the internals would require complex mock verification is not fully correct. The behavior of `closeReader` can be tested with a custom `Reader` that tracks whether `close()` is called. The mutant survives because the current tests only check that the hook is called, and not whether `reader.close()` is actually executed. The problem with this approach is again that we do not change the source code, and this only verifies that the `close()` function is called, not whether its behavior is working correctly.

Behavior Changes There are no behavior changes.

Test Failures and Fixes The tests did not fail.

Test Analysis

1. Do the tests use behaviorally meaningful inputs?
Yes the tests use behaviorally meaningful inputs and checks real behavior. There are other tests like `testSave_ClosesWriter` which mainly use simple inputs just to trigger hook calls to make sure those calls got called.

2. Would these tests still be worth keeping if mutation testing did not exist, or if a human rewrote the implementation differently?
Some tests are worth it, like tests for file creation, content writing, and null validation. However, some other tests that are checking `config.operations` are more tied to the current implementation.

3. Do the assertions check observable behavior, or do they mainly mirror the code?
The assertions use the observable behavior.

Conclusion of the Test Suite:
So the tests show that the refactoring made internal behavior observable, but many tests are more implementation-focused.

Main conclusion The LLM added the getter methods mainly to make the calls to the hook methods observable, so that mutants in the callers of these hooks can be detected. However, this does not make the internal behavior of the hook methods themselves observable. As a result, the observability is limited to whether the hooks are invoked, rather than whether their internal effects are executed.

Readability

Class-level:
At class level, the LLM introduced 8 methods, the cyclomatic complexity went up by 8, the number of variables by 2, and the lines of code increased by 28. This could indicate a slight decrease in readability.
Method-level:
At method level, there are 5 methods targeted. Out of the three `load` methods, only one changed with a decrease of 2 in cyclomatic complexity, an increase of 1 in lines of code and the number of variables. For the two `save` functions, there is only one that had a change. This overload has a decrease of 2 in cyclomatic complexity, 1 in lines of code and the number of variables. For these, this could indicate that the readability went slightly up.

Readability - Qualitative

1. Local clarity - Is the individual method logic easier to understand?
Improved (but could be neutral also): the method names are clear to know what is happening in the flow of the method, and extracting the UTF_OVERRIDE part is nicer to read.

However, there is some indirection, and when a developer would read `validateNotNull` instead of `Validate.notNull`, he/she might think why (since we know it is for testing).

2. Intent - Does the code better express what it does?
Neutral: The charset-related methods improve intent, but wrappers like `validateNotNull` do not add real meaning over `Validate.notNull`.

3. Additional Code Structure - Did the refactoring add extra code structure that makes the codebase heavier to understand?
Worsened: There are many small hook methods that were added to wrap existing calls, which adds indirection and makes the structure heavier without improving core logic.
