BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING

Pre-Processing the Prompt to generate an efficient Output for Unit Test Generation

Lomatul Mahzabin
200042113
Shahrier Al Tanzim
200042140
Zayed Hasan
200042153

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

Pre-Processing the Prompt to generate an efficient Output for Unit Test Generation

Lomatul Mahzabin
200042113
Shahrier Al Tanzim
200042140
Zayed Hasan
200042153

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

Declaration of Candidate

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by **Lomatul Mahzabin**, **Shahrier Al Tanzim**, and **Zayed Hasan** under the supervision of **Jibon Naher**, Lecturer, Department of Computer Science and Engineering Islamic University of Technology, Islamic University of Technology, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.

Jibon Naher
Lecturer
Department of Computer Science and Engineering Islamic University of Technology
Islamic University of Technology (IUT)
Date: September 31, 2025

Lomatul Mahzabin
Student ID: 200042113
Date: September 31, 2025

Shahrier Al Tanzim
Student ID: 200042140
Date: September 31, 2025

Zayed Hasan
Student ID: 200042153
Date: September 31, 2025

Contents

Abstract

Large Language Models (LLMs) have shown remarkable performance across a wide range of natural language processing tasks, yet their effectiveness remains highly sensitive to prompt formulation. Manual prompt engineering is often labor-intensive, inconsistent, and difficult to scale. This thesis investigates the landscape of automated prompt optimization, analyzing recent advancements such as natural language gradient-based refinement, Bayesian Optimization for discrete prompt tuning, evolutionary strategies, and joint fine-tuning approaches. Through a comprehensive comparison of these techniques, the study identifies key trends, limitations, and opportunities in existing methods. Building on these insights, we propose a novel framework that aims to combine the interpretability of natural language-based refinements with the efficiency of automated search strategies. Experimental evaluations demonstrate improved task performance and prompt robustness across multiple LLM benchmarks. This research contributes a unified perspective on automated prompt optimization and paves the way for more adaptive and accessible prompt engineering methodologies.

Chapter 1

Introduction

1.1 Introductory Information

Prompt engineering the art of designing inputs to get LLMs to do certain things, is an unavoidable but delicate part of using these models. Small changes in wording, presentation, or context can have a huge impact on outputs, creating instability and volatility in downstream processes. This sensitivity presents very serious issues, especially in applications with high stakes such as healthcare, legal systems, or finance, where dependability and predictability are critical.

To fill this, researchers have proposed a variety of methods for automatic optimization of prompts. These include some gradient descent-inspired methods founded on natural language feedback (such as ProTeGi), black-box optimization methods like Bayesian Optimization with relaxed token representations, and even hybrid methods that reconcile prompt tuning and model fine-tuning. Evolutionary algorithms have also emerged as an extremely promising direction for prompt strategy optimization between generations, and these come along with scalability and flexibility.

The consequences of auto-optimizing prompts in practical applications are significant. In use cases such as clinical decision support, intelligent tutoring systems, and document generation for the legal profession, minimizing the need to write prompts manually not only saves time but also yields more consistent model behavior. In addition, the increasing deployment of LLMs in edge computing, user applications, and low-resource environments makes efficient, generalizable prompt optimization techniques even more critical.

Throughout this thesis, we explore the current state of the art in prompt optimization

techniques for LLMs, analyze their strengths and weaknesses in a critical light, and present a unified framework that combines the interpretability of natural language feedback with the effectiveness of search-based methods. Our work aims to provide both practical and theoretical contributions towards enhancing prompt engineering with the development of progressively larger language models.

1.2 Motivations and Scope

The increasing use of Large Language Models (LLMs) in software engineering has created novel possibilities for automating tasks such as code generation, documentation, and test case creation. However, the effectiveness and usefulness of test cases generated by LLMs substantially depend on the way the task is defined and the prompt given to the model. Despite the promise, there is no coherent understanding of how prompt types influence the correctness, completeness, and usefulness of generated test cases.

This research is driven by the need to reduce the disparity between human intent and machine response in leveraging LLMs for software testing. Usually, the developers need to write good prompts that will drive LLMs to produce the right and helpful test cases. This project hopes to allow developers to make better use of LLMs so they can automate the generation of tests. This would result in reduced development times, higher-quality software, and increased adoption of AI-based testing in real-world projects.

This thesis places specific focus on how the different types of prompts impact the quality of test cases produced by LLM.It focuses on a narrower scope of testing Java test cases , looking at various approaches to prompting like elaborative versus sparse, formal versus conversational, in context versus out of context. The work isn't about building a new LLM or augmenting the base model itself; rather, it's about how humans can better interact with existing models in a way that results in better output.

The investigation involves qualitative and quantitative evaluation of test cases produced automatically by means of coverage, correctness, and relevance. Though the results are limited by current LLMs' capabilities, the conclusions are likely to provide developers and testers with pragmatic guidance on how to make better use of AI tools in their profession.

1.3 Problem Statement

As large language models (LLMs) become more widely used in software development tasks, including the generation of test cases, a new challenge has emerged: the quality and usefulness of the generated test cases heavily depend on how the prompt is written. Although models like ChatGPT and others have impressive capabilities, the response in most situations is still unpredictable, especially if the input query is subpar or lacks the right context.

This is a challenge for developers who want to leverage these tools for in-the-wild testing; sometimes, they may not even know how to properly ask for what they need so that they'll get useful and relevant test cases. Without proper guidance or prompting techniques, generated test cases will be too shallow, miss edge cases, or fail to capture the original code's logic.

The core problem this thesis addresses is the lack of understanding and guidance around how different types of prompts impact the quality of test cases generated by LLMs. There is currently no clear standard or best practice, and the trial-and-error approach many developers use is both time-consuming and unreliable. This research aims to explore and identify more effective prompting strategies that can consistently yield better results, making LLMs more practical and trustworthy tools in software testing workflows.

Table 1.1: Test File Coverage Summary for LLMs

Test Method	Gemini	Copilot	CODELLAMA
ZeroShot	100%	100%	55.19%
FewShot	100%	100%	87%
Cot	99%	100%	17%
Tot	26%	99%	69.33%
Cotself	100%	100%	85.86%

Aside from showing how unmanageable LLM test case generation becomes based on prompting strategy, Table 1.1 contrasts coverage achieved between the different models (Gemini, Copilot, and CodeLlama) within different approaches towards how to prompt them (ZeroShot, FewShot [3], Chain-of-Thought (Cot) [8], Tree-of-Thought (Tot), and Cotself) . While Gemini and Copilot maintained solid coverage for most prompts (in some cases getting as close as 100%), CodeLlama's performance was incredibly inconsistent with only 55.19% coverage using ZeroShot prompting and as little as 17% using Cot prompting. That inconsistency is a telltale sign of the primary issue: even slight variations in prompt generation can lead to significantly different quality levels of produced test cases, particularly between different LLMs. Lacking

structured prompting techniques, developers stand to get suboptimal or incomplete test suites, further affirming the imperative for structured guidance and inquiry into prompt engineering methods.

1.4 Research Challenges

1. Defining and Measuring Test Case Quality
    - Lack of standardized metrics to evaluate the effectiveness of LLM-generated test cases.
    - Trade-off between automated metrics (code coverage, mutation score) and human-judged quality (readability, relevance).
    - Difficulty in ensuring inter-rater reliability when multiple human evaluators assess test cases.
2. Prompt Design and Variability
    - High sensitivity of LLM outputs to minor changes in prompt wording.
    - Difficulty in determining the optimal prompt structure (zero-shot vs. few-shot vs. chain-of-thought).
    - Risk of overfitting prompts to a specific LLM (e.g., GPT-4) without generalization to other models.
3. Bias and Limitations in LLM-Generated Test Cases
    - LLMs may produce repetitive, superficial, or unrealistic test cases due to training data biases.
    - Difficulty in generating edge cases and negative test scenarios without explicit guidance.
    - Potential for false positives (incorrect test logic) or false negatives (missing critical test cases).
4. Existing Dataset challenge
    - Existing datasets primarily consist of isolated code snippets or individual methods, which are insufficient for evaluating project-level or class-level unit test generation and coverage analysis.

- Comprehensive datasets containing complete, real-world software projects are scarce.
- To address this gap, we collected 20 open-source Java projects from GitHub, ensuring a variety of code structures and complexities suitable for project-level evaluation.

Chapter 2

Related Works

Research in optimizing prompts for large language models (LLMs) has grown rapidly, as the performance of these models can significantly vary based on how prompts are constructed. Rather than treating each study individually, this section organizes existing literature into key thematic areas that align with broader challenges and innovations in the space: Automatic Prompt Optimization, Search and Evolution-Based Prompting, and Hybrid Strategies Combining Fine-Tuning and Prompting.

2.1 Constructing a Thematic Literature Narrative

1. **Automatic Prompt Optimization Techniques**
   Zero of the main research areas is creating methods that automatically optimize prompt formulations to reduce human effort and increase consistency between tasks. The Paper titled "The Prompt Alchemist: Automated LLM-Tailored Prompt Optimization for Test Case Generation" [1] presents a systematic, iterative framework named "MAPS" that guides LLMs to generate and optimize diverse, context-aware prompts for test case generation using domain knowledge, performance feedback, and error-driven refinement. Leverages LLM feedback and domain info to improve prompt quality over iterations.

   The paper titled "Automatic Prompt Optimization with 'Gradient Descent' and Beam Search" [4] introduces ProTeGi, an algorithm that simulates natural language gradients to iteratively optimize prompts. This method employs a beam search action to expand vague task descriptions into more defined instructions, and it is especially useful in the case of annotation and data labeling. It shows how prompt engineering can be reduced to a more scalable, data-driven process.

Likewise, the research article "Prompt Optimization in Large Language Models" [5] employs Bayesian Optimization using continuous relaxation for efficiently searching within the prompt space. The specific interest to black-box LLMs, the research is particularly relevant to realistic situations where access to model internals is not a possibility. Both research articles remedy the shared tendency of deviating from trial-and-error prompt development towards systematic and automated methods.

2. **Evolutionary and Search-Based Approaches**
   Another potential solution is to pair evolutionary algorithms with search-based solutions to fully leverage what LLMs can provide. In the paper "Deep Insights into Automated Optimization with Large Language Models and Evolutionary Algorithms" [9] the authors introduce a paradigm where LLMs handle generating or improving optimization techniques and evolutionary algorithms with the task of finding the most optimal solutions. This two-stage method seeks to push beyond the limitations of current optimization methods by combining the inventive power of LLMs with the flexibility of evolutionary search. The assumption that LLMs are actually capable of assisting in the process of optimizing their own optimization methods is an exciting area of research with immense potential for future research.

3. **Joint Optimization of Prompts and Model Parameters**
   Finally, some work explores combining prompt optimization with model fine-tuning, showing that the two processes can reinforce Zero another. The paper “Fine-Tuning and Prompt Optimization: Two Great Steps that Work Better Together” [6] presents a method where prompts and model weights are optimized in alternating steps. This is particularly impactful for multi-module systems where intermediate outputs aren’t labeled or interpretable. It illustrates how integrating prompt engineering with internal model adaptation can lead to improved outcomes for complex NLP tasks.

2.2 Analysis of Related Works

Research on prompt engineering for large language models (LLMs) has rapidly evolved, with multiple approaches emerging to optimize prompt design and maximize model performance.This section categorizes the literature into three main thematic areas: automated prompt optimization techniques, search and evolutionary-based strategies, and combined fine-tuning and prompt strategies.

The other main area of research is maximizing the efficiency and minimizing the reliance on human trial-and-error for prompt design. "Automatic Prompt Optimization with 'Gradient Descent' and Beam Search" [4] discusses ProTeGi, a system that mimics gradient descent with natural language feedback and optimizes prompts using beam search. Its strength lies in automating prompt optimization using model output learning, without the need for human-in-the-loop optimization. But the method makes an assumption that there is enough good-quality data around, and it can be noisy with regard to model output.

Similarly, "Prompt Optimization in Large Language Models"[5] solves the issue using Bayesian Optimization and a continuous relaxation technique to explore the vast prompt space. This approach is very effective for black-box models, for which gradients aren't available and hence gradient-based methods for tuning can't be utilized. Its primary strength lies in its efficiency and scalability, though the goodness of the search heavily depends upon the initial sample space and objective function used.

Both papers prefer data-driven and automated prompt adjustment, but are in different directions. ProTeGi focuses on natural language search and transformations, while the latter adopts probabilistic modeling. They coincide, however, on a similar goal: abolishing the human-initiated prompt creation bottleneck.

A more inquisitive and biologically driven approach comes in the article "Deep Insights into Automated Optimization with Large Language Models and Evolutionary Algorithms." [9] The article proposes an LLM-EA hybrid where LLMs create or construct optimization approaches and evolutionary algorithms (EAs) are used to optimize towards well-performing solutions. The strength of this work is that it unites the generative capability of LLMs with the adaptive search ability of EAs, potentially scaling to problems for which traditional methods are insufficient.

This approach also predicts an exciting future: models that help optimize their own optimization processes. Nevertheless, its success hinges on proper interaction between the LLM and the evolutionary algorithm, and it is too early to ascertain its scalability on diverse NLP tasks. Unlike ProTeGi and Bayesians, the LLM-EA paradigm allows for a more exploratory search process rather than converging optimization.

While the first two categories attempt to optimize prompts alZero, "Fine-Tuning and Prompt Optimization: Two Great Steps that Work Better Together" [6] argues for a more holistic pipeline. It explores alternating between fine-tuning LLM weights and prompt optimization in iterative cycles. This strategy enables the model to refine both its internal representations and external task descriptions, which is especially valu-

able in complex multi-module NLP systems where intermediate annotations are unavailable.

The novelty of this approach is its co-adaptive mechanism: prompts evolve while the model itself improves, creating a feedback loop that strengthens task performance. However, the process can be computationally expensive and assumes access to model internals for fine-tuning, limiting its use with proprietary black-box LLMs like GPT-4.

2.3 Compare and Contrast Different Works

While ProTeGi applies a natural language "gradient descent" strategy for optimizing prompts by beam search towards iterative human-readable improvement of prompts, the Bayesian Optimization with continuous relaxation accomplishes the reverse by formulating prompt engineering as a combinatorial optimization task over n-grams and utilizing Bayesian techniques to conduct search in the discrete space efficiently. ProTeGi emphasizes interpretability and usability by creating more transparent task-focused instructions, whereas the Bayesian method focuses on black-box optimization performance without the requirement of model internals.

Unlike this, the LLM-EA method extends beyond the optimization of prompts only by proposing a generalized optimization framework where LLMs yield strategies and Evolutionary Algorithms search. The strategy is customized to solve broader optimization problems and reduces the amount of human effort that goes into designing a strategy, unlike ProTeGi and Bayesian Optimization that are more tightly bound to prompts. The LLM-EA paradigm is characterized by automating not only the prompts but also task solution strategies themselves with the aim of being adaptive across problem spaces.

By contrast, BetterTogether takes a different approach by interweaving prompt optimization with LM weight fine-tuning, alternating between the two to enable a bootstrapped form of self-improvement. Unlike ProTeGi, which treats prompt refinement as a standalZero task, or the LLM-EA paradigm, which separates generation and search, BetterTogether views the LM as a unified system that can be trained in parallel flattening the model-centric vs. prompt-centric optimization dichotomy.

Hence, while all four works address the challenge of improving LLM performance through the use of optimization, the four pieces differ immensely in scope, methodology, and assumptions. ProTeGi and Bayesian Optimization focus on optimizing prompts in interpretable and computationally efficient ways. The LLM-EA paradigm

generalizes optimization through the use of a hybrid AI approach, and BetterTogether explicitly integrates prompt engineering into model learning, offering a new direction towards modular LM systems for sophisticated NLP pipelines. Together, these papers show a rich picture of solutions, from quick-in parameters to integrated, adaptive LLM optimization systems.

2.4 Gaps and Opportunities

Despite their achievements, current methodologies for rapidity and optimization strategy enhancement in LLMs unveil several significant gaps that leave scope for improvement.

Second, while ProTeGi emphasizes interpretability with natural language gradients, it is largely set in the scenario of a limited beam search framework and may struggle to scale with more prompt-complex or domain-specific applications. It also fails to actively address multi-module LLM pipelines or cross-prompt dependencies, which are increasingly relevant in state-of-the-art NLP systems.

The Bayesian Optimization with continuous relaxation method demonstrates outstanding performance in black-box environments but is inherently constrained by dependence on fixed-length n-gram sequences and inability to support responsiveness to dynamically or semantically richer prompt structures. Also, the strategy prioritizes performance over interpretability, with minimal explanation provided for why certain prompts are superior. Rendering it less useful in human-in-the-loop usage.

The LLM-EA paradigm, though ambitious, is yet to be fully explored in task-specific, real-world benchmarks. It frames optimization in general terms but will probably suffer from lacking domain constraints and therefore find it difficult to determine the appropriateness of the evolved strategies to highly specialized tasks. Its dependence on successful representation and operator design also means the need for more formalization and experimentation across different domains.

BetterTogether offers a novel bootstrapping method via alternating prompt and weight optimization, but it relies on the presence of rich training data and computational resources to facilitate frequent fine-tuning a requirement that is not fulfilled in most low-resource or real-time environments. Moreover, its application to tasks with non-differentiable evaluation metrics remains an open area of research.

Taken together, these limitations suggest several research directions:

1. Engineering domain-adaptive prompt optimization methods that go beyond static

forms and involve user or task context.

2. Engineering resource-lean optimization algorithms that can be applied in low-data or real-time settings without fine-tuning the full LM.
3. Exploring interpretable optimization techniques that retain transparency while maintaining performance gains, bridging usability and accuracy.
4. Scaling current models to accommodate multi-module LLM systems with dependent prompts, which are now more and more common in advanced NLP pipelines.
5. By bridging these gaps, future work can provide more solid, adaptable, and scalable means of LLM prompt and strategy optimization. Moving towards generalized, human-centered AI systems.

2.5 The Role of Transition Words in Literature Synthesis

2.5.1 Compare and Contrast of Prompt Optimization Approaches

ProTeGi, say, suggests a natural language "gradient descent" mechanism for iteratively refining prompts via beam search. This focus on interpretability and usability differs from the efforts in Bayesian Optimization, which, instead, represent prompt tuning as a combinatorial problem and employ a continuous relaxation approach for efficient black-box search.

Conversely, the LLM-EA paradigm goes Zero step further by optimizing strategies themselves, as opposed to prompt structures. While ProTeGi and Bayesian methods are centered on prompting optimization to enhance performance, the LLM-EA approach leverages the generation potential of LLMs and Evolutionary Algorithms to create optimization strategies, thereby reducing manual heuristic design needs.

Combining the limitations of prompt-only approaches, the BetterTogether framework proposes an integrated strategy, alternating between prompt tuning and model weight fine-tuning. This approach is novel in that it enables the model to effectively "teach itself" with minimal supervision, a major advantage in modular NLP pipelines where intermediate labels are not present.

Together, these approaches show the range of solutions currently available as attempts to solve and optimize the prompt and strategy. Some emphasize interpretability and

usability (e.g., ProTeGi), others performance within resource-limited environments (e.g., Bayesian Optimization), or propose unified frameworks for learning and reasoning (e.g., BetterTogether). Each, despite its heterogeneity, is trying to tackle different features of the same fundamental problem: improving LLM performance via wiser, automated prompting techniques.

2.5.2 Gaps and Opportunities in Existing Literature

While these improvements exist, there remain important lacunae in the literature that contain high potential for future research. For example, while ProTeGi generates understandable improvements with natural language feedback, it does not yet scale to extremely complex or domain-specific prompts.

Second, the Bayesian Optimization approach, as powerful as it is, can only work on strict token orderings and lacks adaptive mechanisms to adjust task needs over time. It also cannot produce human-interpretable reasons why certain prompts work better. Something of crucial importance in human-in-the-loop systems.

Conversely, the LLM-EA paradigm, as abstract and flexible in theory, nZerotheless lacks standards of actual deployment and testing in real-world settings. Hence, its true effectiveness on domain-specific tasks remains uncertain. BetterTogether, as innovative in integrating the optimization of prompt and weights, relies on the availability of large training data and computing resources, which may be unavailable in low-resource environments.

Taken together, these limitations necessitate new research directions. To be precise, it can be explored how domain-adaptive, context-aware prompt optimization methods can be explored; low-resource and real-time optimization methods can be developed; and understandable but high-performing frameworks that can optimize prompts in modular, multi-stage LLM architectures can be designed.

By solving these challenges, work in the future can close the gap between real-world applicability and theoretical innovation and making efficient optimization techniques not only effective but also practical, understandable, and able to be adapted to new use cases.

2.6 Summary

The current chapter has discussed the shifting momentum of prompt optimization and automated processes in the domain of Large Language Models (LLMs). Under-

pinning all of the literature in the field is a similar strand of increasing recognition of prompt sensitivity as a performance predictor for LLMs and thus the necessity for automated processes with the capacity to adapt to task-oriented demands. From gradient-inspired methods like ProTeGi, to Bayesian Optimization for black-box optimization, to more general paradigms like LLM-EA, researchers are actively looking into ways to reduce reliance on manual prompt engineering and improve efficiency, interpretability, and adaptability.

Despite these advances, there are several notable gaps that still exist. Most existing methods either sacrifice interpretability for performance or do not scale and generalize in real-world settings. Moreover, techniques like BetterTogether, while new in the sense of combining prompt and weight optimization, also tend to assume access to large computational resources. Excluding their practical application in resource-limited settings.

Collectively, these results highlight the pressing need for an adaptive and interpretable efficient optimization paradigm that is capable of operating under low-resource conditions and can be applied to modular or black-box LLM pipelines. These results also reinforce the central problem being addressed in this thesis: discovering a generalizable, low-latency, and human-aligned prompt optimization solution.

In the next chapter, we detail these deficiencies by describing our proposed method, aimed at filling gaps identified in current literature and providing a valuable, scalable contribution to the nascent field of LLM prompt engineering.

Chapter 3

Proposed Methodology

The proposed methodology focuses on enhancing the quality of prompts to improve Large Language Models' (LLMs) performance in unit test generation. Unit test outputs from LLMs can vary significantly depending on the structure, clarity, and specificity of the input prompt. Therefore, this work emphasizes systematically processing and refining prompts to create more precise and effective queries. By optimizing the prompt structure, the goal is to achieve more accurate, consistent, and contextually relevant unit test generation responses.

3.1 Overview of the Methodology

The Methodology can be divided into the following components.

3.1.1 Experiments

Initially, a series of experiments was conducted using a variety of prompt types, including zero-shot, few-shot, chain-of-thought (CoT), tree-of-thought (ToT), and CoT with self-thought prompting strategies. These prompts were utilized to generate unit tests for Python methods, some of which involved inter-class dependencies. Observations revealed that the quality and nature of the generated unit tests varied significantly depending on the type of prompt used. For instance, CoT and ToT prompts often provided better explanatory depth but struggled to produce quality, executable test cases, whereas few-shot prompts, despite offering less detailed reasoning, resulted in more structurally sound and practically useful unit tests.
Moreover, it was observed that overly complex or irrelevantly large prompts often led to hallucinations in the LLM responses, generating unnecessarily complicated and

off-topic outputs. Such prompt constructions not only degraded the quality of the generated unit tests but also introduced elements that were unrelated to the original method, thereby complicating the overall outcome. [11], different methods responded better to different prompt strategies; no single prompting style consistently performed best across all cases. This further highlighted that unit test generation is highly sensitive not only to the structure and clarity of the prompt but also to the characteristics of the underlying code being tested, and that results remain inconsistent across different LLMs and scenarios.

For quality analysis, each generated unit test was rigorously evaluated using Pylint, which was instrumental in identifying syntax errors, semantic errors, code quality warnings, and adherence to established coding conventions. To further assess the comprehensiveness of the test cases, the Python Coverage library was employed to measure the percentage of code lines covered by the tests, offering insights into their depth and effectiveness. Alongside these tools, several key performance metrics were systematically recorded for each prompt category. These included the number of test failures and test successes, error counts in the generated tests, and the frequency of warnings and convention break counts, which collectively provided a measure of overall code quality. Additionally, success rate and error rate were computed to capture the reliability of the test generation process, while branch coverage was monitored to evaluate the extent to which different execution paths in the code were exercised.

We changed our method as the study progressed because our initial approach proved insufficient for capturing the level of detail and reliability we needed. At the beginning, we aimed to evaluate unit test generation at the project level, but most of the available projects were written in Java, which led us to focus on Java projects specifically. From there, we selected 20 Java projects consisting of 442 classes to establish a robust and diverse evaluation benchmark. In the first phase of experimentation, we applied Chain of Thought (CoT), Zero-shot, Few-shot, Tree-of-thought (ToT), CoT with self-thought (COTself) prompting to guide the LLM in reasoning through unit test generation. However, we quickly observed that CoT alone was not providing enough contextual information about the code, which limited the quality and coverage of the generated test cases.

To address this, we shifted toward incorporating context retrieval techniques, with a particular emphasis on the Abstract Syntax Tree (AST). Inspired by the insights from The Paper named " TestBench: Evaluating Class-Level Test Case Generation Capability of Large Language Models [10], we learned that by utilizing ASTs, it was possible to reduce unnecessary complexity while still preserving critical structural information.

Specifically, TestBench demonstrated that removing all function bodies and variable initialization statements, while retaining only declarations, effectively reduced the length of contextual input without losing essential details. This transformation, referred to as the simple context, helped streamline the prompt while keeping it informative for the LLM.

After refining the prompt using the AST-based context, we introduced an additional layer of improvement by designing an instruction set to guide the LLM more effectively in generating high-quality unit test cases. [7] Initially, this instruction set was created based on our own experience and observations during the early stages of experimentation. These rules primarily consisted of simple but essential guidelines that we considered necessary for an LLM to produce meaningful and executable unit tests from a given source code. However, we recognized the limitations of relying solely on our own perspective and sought to validate and strengthen these rules against industry standards.

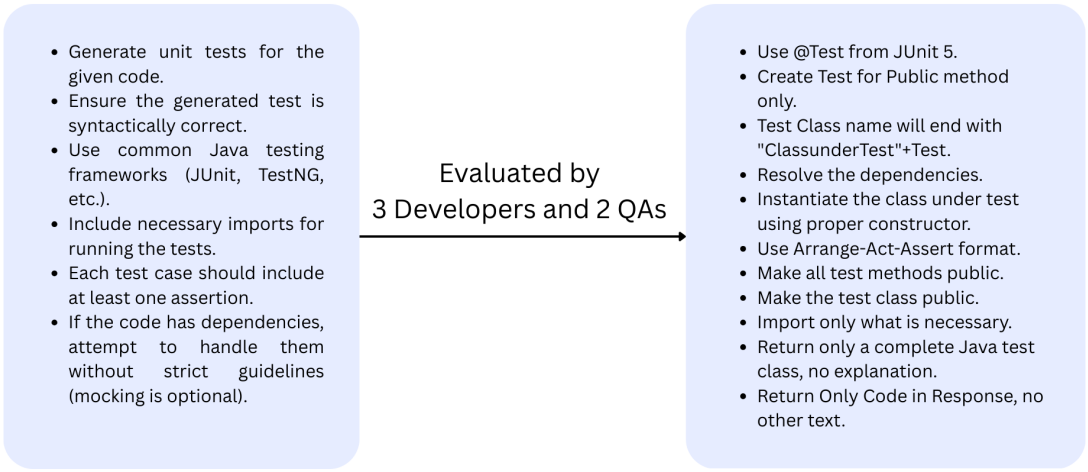

Figure 3.1: Previous Vs Refined Rules

To achieve this, we conducted an expert evaluation of our human-made rules. We collaborated with professionals from a reputed software company, engaging three experienced developers and two quality assurance (QA) engineers to review our instruction set. Their expertise and practical experience in software development and testing allowed them to critically assess the effectiveness, completeness, and clarity of the rules we had formulated. Based on their evaluation, they provided refinements and improvements that transformed our initial rules into a set of industry-standard guidelines.

This collaborative refinement process ensured that the instruction set not only aligned with academic insights but also reflected real-world testing practices. As a result, the rules became more robust, practical, and capable of consistently guiding the LLM in generating unit tests that are both syntactically correct and effective in uncovering potential defects in production code.

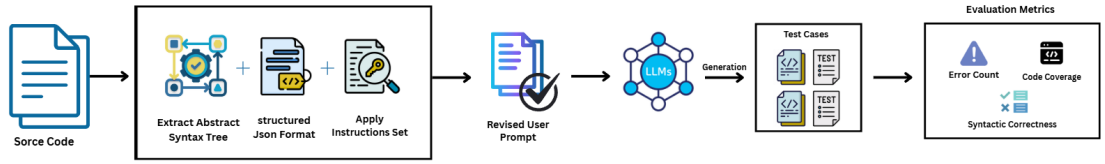

Figure 3.2: Our Proposed Piepline

3.1.2 Prompt Processing

Given the observed variability and inconsistency in unit test generation outcomes, a dedicated prompt preprocessing stage was introduced into the methodology to ensure that the input provided to the Large Language Model (LLM) is well-structured, context-aware, and optimized for producing high-quality results.

Instead of submitting raw user prompts directly to the model, this stage refines and augments the input through a series of transformations. The process begins with the detection of the programming language and its associated unit testing framework, such as JUnit for Java, xUnit for C#, or Pytest and Unittest for Python, which ensures that the generated test cases are aligned with the correct syntax and conventions.
Once the language is identified, the provided code snippet is parsed into an Abstract Syntax Tree (AST), which captures the syntactic and semantic structure of the program. For Java programs, this is achieved using the javac library in Python, which allows the code to be transformed into a machine-interpretable representation.

To convert raw Java code into a structured and machine-readable representation, the Abstract Syntax Tree (AST) was parsed using the javalang library and then serialized into JSON format. During this process, several important fields were extracted, each of which captures structural or behavioral details of the code. These fields are essential because they form the foundation upon which the LLM can generate reliable and context-aware unit tests.

The first layer of extraction involves capturing the package name and import statements. The package name situates the class within its namespace and ensures that

the generated tests remain consistent with the project's organizational structure. Similarly, the import statements highlight all external classes and libraries referenced by the program. These imports are particularly important for identifying dependencies, as they indicate external services or APIs that may require mocking or stubbing when writing unit tests.

The next focus is on class declarations. From each class, the name and access modifiers (such as public, private, or abstract) are extracted. This information provides structural context and determines whether methods are accessible for testing. In addition, class-level fields are recorded, including their names and types. These fields often represent critical dependencies injected into the class, such as services or repositories. Detecting these fields early is crucial, as they guide the choice of mocking frameworks or strategies during test generation.

Constructors form another key component of the AST. For each constructor, the parameters along with their types and names are extracted, as well as the modifiers that define its accessibility. Since constructors often establish dependency injection or initialize important class state, documenting them allows the LLM to generate tests that properly instantiate objects under realistic conditions.

The methods of a class are the central elements of the extraction process. For each method, the name, access modifiers, return type, and parameter details are captured. These attributes define the behavior to be tested and form the primary input for unit test generation. Beyond this surface-level information, the method body is analyzed for deeper insights. All method invocations are extracted, including the object on which the call is made (qualifier), the method name being called (member), and the arguments passed during the invocation. These invocations reveal inter-class interactions and dependencies, enabling the prompt to instruct the LLM whether mocking or verification is required in the tests.

Finally, conditional statements such as if statements are also extracted from the AST. For each conditional, the logical expression governing the branch is recorded, along with whether an accompanying else branch exists. These conditional structures expose the internal decision-making logic of the method, and recognizing them is vital for generating tests that cover not only the main execution flow but also boundary conditions and exceptional cases.

Overall, the extracted fields include package and import information, class declarations, fields, constructors, methods, method invocations, and conditionals. Together, they provide a complete and structured view of the program that is far more infor-

mative than raw source code. This structured representation ensures that the LLM receives explicit context about dependencies, execution flow, and behavioral logic, enabling it to generate unit tests that are accurate, comprehensive, and aligned with the intended functionality of the code.
This structured JSON representation reduces ambiguity and provides the LLM with a distilled view of the code, making it easier to focus on essential logic rather than parsing unstructured syntax. In parallel, the instructional component of the original prompt is analyzed and enriched with additional directives to improve clarity and coverage, including explicit expectations around edge cases, exception handling, boundary testing, and parameterized test design. Finally, the structured code representation and the refined instructional segment are recombined into a coherent and unified prompt, ensuring that the LLM receives both the semantic understanding of the code and precise guidelines for unit test generation.
Through experimentation, it became evident that no single fixed prompting strategy consistently produced reliable results across different coding contexts and LLMs. Even advanced models specifically fine-tuned for code generation were highly sensitive to the clarity, structure, and contextual relevance of the input prompt. As a result, the integration of this multi-step preprocessing mechanism emerged as a critical innovation to enhance the reliability, consistency, and overall quality of automated unit test generation.

3.1.3 Evaluation

For the evaluation, we focus on a set of well-defined metrics that allow us to measure both the correctness and effectiveness of the generated unit tests. First, we analyze syntactic correctness by checking whether the generated tests contain any syntax errors that would prevent them from being parsed correctly. This is followed by compilation correctness, where we dynamically compile the generated test cases using Maven to verify that they can be successfully built and executed without issues related to missing variables, incorrect references, or unresolved dependencies.

Beyond compilation, we evaluate the execution correctness of the tests by running them to see whether they produce meaningful results with valid assertions and without misclassifying correct logic as failures. To further assess quality, we measure branch coverage and statement coverage using JaCoCo, which provides insights into how well the generated tests exercise different parts of the source code and whether they address potential edge cases. Finally, we record the error count from test execu-

tion, which helps in quantifying the robustness of the generated tests. Together, these metrics provide a comprehensive picture of the reliability, coverage, and overall effectiveness of the LLM-generated unit tests, enabling us to objectively compare different approaches and configurations.

3.2 Chronological Breakdown of Components

3.2.1 Experiment Component

The primary objective of this component is to systematically evaluate the performance of LLMs in generating unit tests across real-world Java projects. This phase investigates how the use of structured preprocessing and refined prompting techniques influences the quality, correctness, and coverage of the resulting test cases.

The primary objective of this component is to systematically evaluate the impact of different prompt techniques on the generation of unit tests by an LLM. This phase investigates how variations in prompt structure affect code quality and test efficacy.

Preparation

- **Project Selection:** A total of 20 open-source Java projects were selected, containing 442 classes in total. These projects were chosen to provide diversity in complexity, coding style, and dependencies, ensuring a realistic and challenging evaluation environment.
- **Code Parsing**: Each project was parsed using the javalang library to extract abstract syntax trees (ASTs). From the AST, essential metadata such as class names, method signatures, parameters, return types, and inter-class dependencies were captured for further analysis.
- **Environment Setup**: The experimental pipeline was designed to ensure reproducibility. Maven was used for compiling generated test cases, while JaCoCo was integrated to measure branch and statement coverage. All intermediate results (ASTs, refined prompts, generated test cases, and execution logs) were stored for evaluation.

Prompt Technique Application

- **Baseline Prompts**: Direct prompts without preprocessing were applied to measure the raw performance of LLMs in unit test generation.

- **AST-driven Prompts**: Refined prompts enriched with AST-derived metadata and simplified context (declarations only, without method bodies) were applied to provide additional structural clarity to the LLM.
- **Instruction-Enhanced Prompts**: Prompts were augmented with rule-based instructions, initially designed by the researchers and later refined through industry expert feedback, to evaluate the effect of structured guidance on test quality.

Thus, this experiment component establishes the foundation for evaluating how preprocessing, structured context, and instruction refinement collectively impact the quality and reliability of LLM-generated unit tests.

3.2.2 Prompt Processing

In this phase, the user-provided prompt is systematically processed and refined through a multi-step pipeline to ensure that the input provided to the LLM is structured, unambiguous, and optimized for high-quality unit test generation. Each step is carefully designed to reduce ambiguity, capture essential metadata, and enrich the testing requirements. The following breakdown illustrates the complete preprocessing workflow:

Step 1: Language and Test Framework Detection The first step is to identify the programming language of the given code snippet and determine the corresponding unit testing framework.

- **Programming Language Detection:** Ensures that the model interprets the code correctly, whether it is Java, Python, C#, or another language.
- **Test Framework Mapping:** Links the detected language to the appropriate testing framework (e.g., JUnit for Java, xUnit for C#, Pytest/Unittest for Python).
- **Purpose:** This step guarantees that the generated unit tests follow the correct syntax, conventions, and framework-specific features.

Step 2: Abstract Syntax Tree (AST) Extraction After detecting the language, the code is parsed into its Abstract Syntax Tree (AST).

- **Parsing:** For Java, the AST is generated using the javac library of Python.
- **Metadata Extraction:**The AST captures essential structural information such as:
  - Class definitions and method signatures

  - Parameter types and return values
  - Control flow constructs (loops, conditionals)
  - External dependencies and imported libraries
- **Purpose:** By transforming the code into AST form, the system eliminates ambiguity and creates a machine-readable representation of the program's logic.

Step 3: Transformation into Structured JSON Format The AST is then converted into a standardized JSON structure for easier integration with the LLM.

- **JSON Conversion:** The AST nodes are serialized into key-value pairs.
- **Content:** The JSON includes:
  - Class definitions and method signatures
  - Parameter types and return values
  - Control flow constructs (loops, conditionals)
  - External dependencies and imported libraries
- **Purpose:** JSON provides a concise yet rich representation of the code, allowing the LLM to consume structured data rather than parsing raw syntax.

Step 4: Instruction Extraction and Refinement The instructional part of the prompt is analyzed and improved to ensure completeness and clarity.

- **Instruction Classification:** User instructions are categorized into:
  - Action Requests like "Generate unit tests," "Mock dependencies"
  - Constraints like "Use JUnit," "Handle null pointer exceptions"
  - Style/Format Requirements like "Provide output in markdown"
  - Examples/Templates (few-shot demonstrations within the prompt)
- **Reference Resolution:** Ambiguous references (e.g., "this method," "these cases") are resolved by linking back to the extracted code metadata
- **Constraint Handling:** Specific requirements for libraries, frameworks, or test strategies are incorporated.
- **Handling Missing Information:** Incomplete or ambiguous prompts are enriched by inferring missing details from the code (e.g., automatically identifying edge cases or exceptions to test).

- **Instruction Enhancement:** Additional guidelines are added for:
  - Edge case coverage
  - Exception handling strategies
  - Boundary value testing
  - Parameterized test generation
- **Purpose:** Ensures that the LLM receives explicit and actionable instructions rather than vague or incomplete requests.

Step 5: Final Prompt Recombination The last step combines the processed code (in JSON format) and the refined instructions into a unified prompt.

- **Integration:** The JSON representation of the code and the enhanced instruction set are recombined coherently.
- **Controlled Randomization:** Slight variations can be introduced to avoid repetitive outputs and encourage diversity in test generation.
- **Purpose:** This ensures the LLM receives both the structured semantic representation of the code and precise testing directives, leading to high-quality, reliable, and context-aware unit tests.

By following this structured five-step process starting with language and framework detection, moving through AST parsing and JSON transformation, and ending with instruction refinement and prompt recombination, the methodology ensures that the LLM is not left to infer or guess critical details. Instead, it operates on a well-prepared, context-rich prompt that minimizes hallucinations, improves coverage of edge cases, and enhances the overall consistency and quality of unit test generation.

3.2.3 Evaluation Metrics and Tools

We measure the effectiveness of the generated test cases based on the following aspects:

- **Error Count:** We assess the number of errors raised during the execution of the generated test cases. This metric provides a direct measure of the stability and reliability of the generated tests by capturing runtime anomalies, unexpected behaviors, and assertion failures. A higher error count indicates lower test reliability, while fewer errors suggest that the generated cases are structurally and logically sound.

- **Branch Coverage:** To evaluate the thoroughness of the generated tests, we calculate branch coverage, which measures whether all possible decision paths in the source code are executed at least once. This ensures that the generated tests are not limited to simple paths but also address conditional logic, improving defect detection capabilities. The measurement of branch coverage is carried out using code instrumentation tools, allowing precise assessment of decision coverage.
- **Statement Coverage:** Beyond branches, statement coverage is measured to determine the proportion of source code statements executed during test case execution. This metric evaluates the extent to which the generated tests exercise the production code. [2] High statement coverage indicates that the test suite effectively validates the majority of the program logic, while low coverage highlights gaps where the code is left untested.
- **Compilation Correctness:** Static analysis alone cannot capture errors related to identifiers, class references, or library dependencies. Therefore, we verify compilation correctness by dynamically compiling the generated test cases. Only test cases that compile without errors are considered valid for execution and further evaluation. This step ensures that the generated tests are both syntactically accurate and contextually compatible with the production codebase.

3.3 Integration and Interconnections

3.3.1 User Prompt Preparation

In this initial stage, a diverse set of prompts is constructed to simulate real-world variations and complexities.

The process begins with the user prompt, which typically contains a mixture of source code and natural language instructions. At this stage, the prompt is prepared by detecting the programming language and identifying the associated testing framework. This ensures that the subsequent steps can handle the input in a standardized manner. The user prompt serves as the raw input to the system and acts as the foundation for all downstream components. However, in its unprocessed form, the prompt often lacks the clarity and structure needed by an LLM to generate high-quality test cases. Therefore, the prepared prompt is immediately passed into the prompt processing component, where it undergoes structured analysis and refinement.

3.3.2 Prompt Processing

The prepared prompt is then systematically transformed through the prompt processing component, which acts as the bridge between raw user input and optimized LLM input. First, the source code portion of the prompt is parsed using the javalang library, where the abstract syntax tree (AST) is extracted. The AST is then converted into a structured JSON representation that retains critical metadata such as class names, method signatures, parameters, return types, and inter-class dependencies. This structured context significantly reduces ambiguity and provides the LLM with a precise understanding of what needs to be tested.

In parallel, the instruction part of the prompt undergoes classification and refinement. Here, rules and guidelines are applied initially designed through researcher experience and later refined with feedback from industry experts to ensure that the LLM receives consistent, unambiguous instructions. Once processed, the refined code context and enhanced instructions are recombined into a single coherent prompt. This refined prompt is then fed into the test generation phase of the pipeline, ensuring that all subsequent outputs benefit from the structured preprocessing.

3.3.3 LLM Execution

The processed prompt is then submitted to a selected LLM for response generation. This stage represents the core interaction between the system and the language model, where the structured code context and enhanced instruction set guide the model to generate unit tests. The activities in this component include feeding the optimized prompt into the LLM, retrieving the generated outputs, and filtering out incomplete or irrelevant responses. Since multiple test cases may be produced, the execution stage also ensures that only syntactically valid Java test classes are forwarded to the next stage. The effectiveness of this step heavily depends on the clarity achieved during prompt processing, which demonstrates the interdependence between components.

3.3.4 Evaluation and Analysis

Once the output is generated, it undergoes a comprehensive evaluation using both automated quality assessment tools and manual response evaluation strategies. The evaluation and analysis component is responsible for systematically assessing the quality of the generated unit tests. Once the candidate test cases are produced, they are subjected to multiple layers of evaluation. First, syntactic correctness is verified using static analysis tools to detect parsing errors. Next, compilation correctness is checked

using Maven to ensure that the test cases integrate properly with the project code. Execution correctness is then assessed by running the test cases and recording whether they produce valid results with meaningful assertions. To measure coverage, JaCoCo is used to calculate both statement and branch coverage, providing insights into how thoroughly the test cases explore the code under test. Finally, error counts and execution logs are analyzed to capture failure rates and robustness. This component closes the loop by producing detailed evaluation metrics that can be compared across different prompt strategies and models.

3.3.5 Multi-Model Evaluation Loop

To strengthen the generalizability of the findings, the system incorporates a multi-model evaluation loop. Instead of relying on a single LLM, the refined prompts are executed across multiple models with varied architectures and training objectives. This loop allows us to compare how different LLMs respond to the same refined prompt structure and whether prompt processing strategies generalize beyond a single model. After each round of generation and evaluation, results are aggregated and compared to identify patterns in correctness, coverage, and overall effectiveness. The loop also introduces controlled variation by applying both basic and refined instruction sets, allowing us to assess the impact of instruction quality across models.

3.4 Clarity and Comprehensiveness in Visual Aids

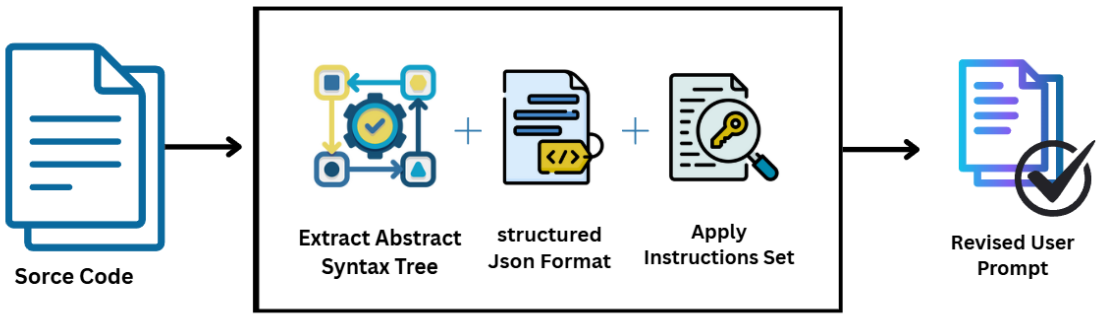

Figure 3.3: Prompt Pre-Processing

3.5 Conclude with a Summary of the Methodology

Rather than being independent modules, preprocessing, execution, and evaluation continuously interact, reinforcing the system's goal of generating higher-quality, more insightful outputs.

- Prompts are consistently structured for better model handling.
- Models are fairly and uniformly tested under controlled settings.
- Evaluations yield quantitative, comparable data across models.

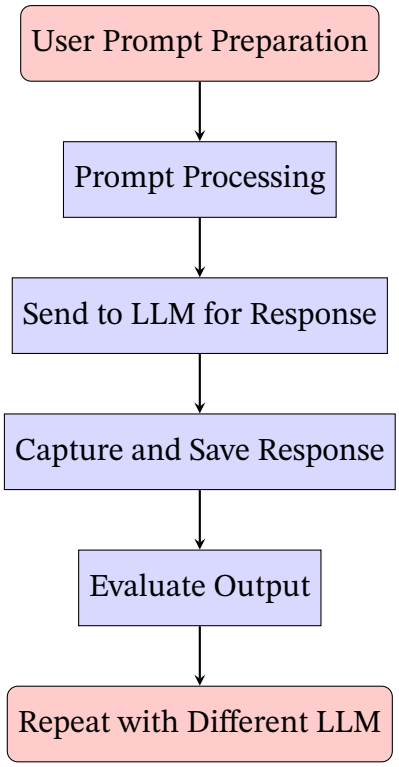

Figure 3.4: Flowchart for Processing User Prompts and Evaluating LLM Outputs

Chapter 4

Results and Discussion

4.1 Datasets and Experimental Setup

4.1.1 Database Setup

- **Project Selection:** A total of 20 open-source Java projects were selected, containing 442 classes in total. These projects were chosen to provide diversity in complexity, coding style, and dependencies, ensuring a realistic and challenging evaluation environment.
- **Code Parsing**: Each project was parsed using the javalang library to extract abstract syntax trees (ASTs). From the AST, essential metadata such as class names, method signatures, parameters, return types, and inter-class dependencies were captured for further analysis.
- **Source Code Format**: The provided methods were stripped of comments, descriptions, and any extraneous context. This minimalist approach ensured that the models were not influenced by any additional information, focusing purely on the method's structure and functionality. The code was provided in its raw form to prevent any bias or external input from affecting the test generation process.
- **Prompt Strategies**: To evaluate the impact of different input formats on test case generation, two prompt strategies were selected:
  - **AST+SRC**: This strategy combines the abstract syntax tree (AST) with the raw source code (SRC) to provide the model with both structural and textual context. It was chosen to investigate whether richer, structured input

improves test coverage and reduces generation errors.

- **SRC only**: This strategy provides only the raw source code as input, serving as a baseline to assess the effect of adding AST information. It helps determine the advantage of structural guidance over plain source code.

4.1.2 Experimental Setup

- **Model Used**: We selected these three models to cover a diverse spectrum of large language models in terms of size, specialization, and accessibility:
  GPT-OSS-120B: This open-source, large-scale LLM (120B parameters) was included to evaluate how an open-access, general-purpose model performs in generating unit tests for Java projects. Its large parameter count allows us to observe capabilities typical of state-of-the-art models available to the research community.
  Gemini-2.5-Pro: A general-purpose proprietary LLM, Gemini-2.5-Pro represents commercially optimized models. Including it allows for comparison between open-source and commercial systems in terms of code understanding, coverage, and error generation.
  Qwen-2.5-Coder-32B-Instruct: As a code-specialized LLM, Qwen-2.5-Coder-32B-Instruct is designed specifically for programming tasks. Evaluating this model highlights the potential advantages of domain-specialized models for automated unit test generation, especially in handling code structure, method coverage, and error minimization.

- **Human Intervention**: Due to the lack of API access for GPT and Google Gemini, human intervention was required to manually input the methods and prompts into their respective web interfaces. Extra caution was exercised not to provide any additional data or alter the input in any way. This ensured that the prompts remained as close as possible to the intended test setup, mimicking the way they were fed to the Python script.

- **Test Evaluation**: The generated unit tests were saved as Java files and systematically evaluated using tools such as JaCoCo and compiler checks. This allowed us to measure both the correctness and the coverage of the test cases. The evaluation was applied consistently across all LLMs and prompt strategies, ensuring that results reflect the true impact of AST+SRC versus SRC prompts without influence from external or unintended factors.

4.2 Presenting the Results

4.2.1 Metric: Coverage

Table 4.1: Code Coverage Comparison Across LLMs for AST+SRC and SRC Prompts

LLM	Prompt Type	Branches (%)	Lines (%)	Methods (%)
GEMINI	AST+SRC	46.04	56.79	68.38
	SRC	35.84	49.06	65.33
GPT	AST+SRC	47.80	67.13	80.26
	SRC	38.05	58.18	70.53
QWEN	AST+SRC	55.59	62.34	73.28
	SRC	37.93	45.23	67.80

Results on Prompt Comparison (LLM Gemini)

To examine the impact of AST-based prompts(AST) versus raw source (SRC) prompts, we analyzed LLM Gemini's coverage across branches, lines, and methods. As shown in Figure 4.1, AST prompts consistently outperform SRC prompts. Branch coverage increases from 35.8% to 46.0%, line coverage from 49.1% to 56.8%, and method coverage from 65.3% to 68.4%. The bar and line charts clearly show that AST prompts improve overall coverage, with the largest gains in branch coverage, indicating better handling of decision logic. Overall, using AST context helps the LLM generate more comprehensive tests than using the raw source alone.

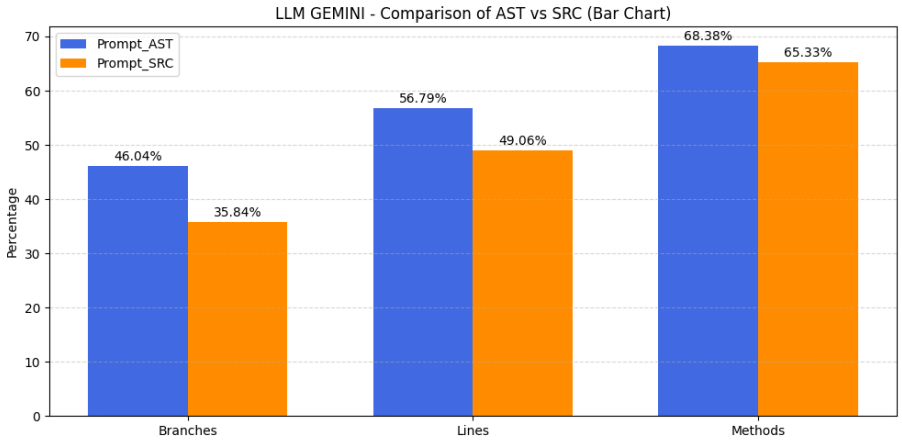

Figure 4.1: Prompt Coverage Comparison: Gemini

Results on Prompt Comparison (LLM GPT)

We compared AST-based prompts(AST) with raw source code (SRC) prompts for LLM GPT. Figure 4.2 shows the results as a bar chart for branch, line, and method coverage. The bar chart shows that Prompt_AST gives higher coverage than Prompt_SRC.

Branch coverage rises from 38.05% (SRC) to 47.8% (AST), meaning AST prompts help with conditional logic. Line coverage increases from 58.18% to 67.13%, showing AST prompts allow tests to cover more code. Method coverage improves the most, from 70.53% to 80.26%, reflecting better testing of methods. The largest gap appears in method coverage, highlighting the benefit of AST context. In summary, AST-based prompts significantly improve coverage in LLM GPT, especially for methods, making tests more thorough and complete.

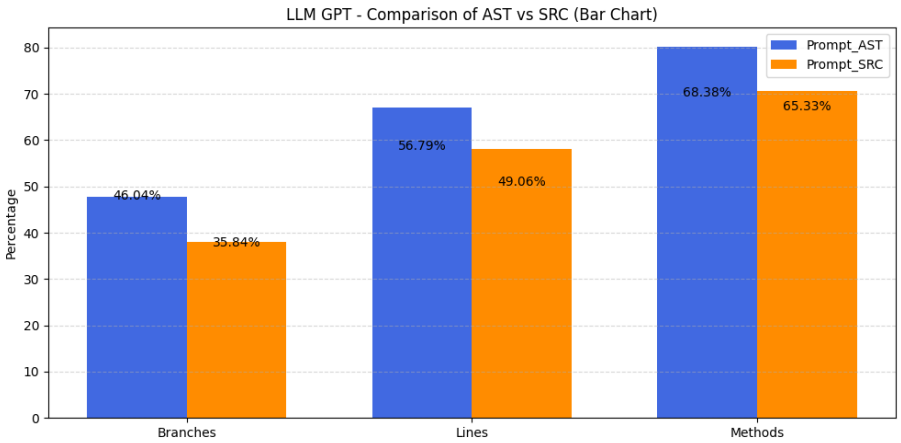

Figure 4.2: Prompt Coverage Comparison: GPT

Results on Prompt Comparison (LLM Qwen)

For LLM Qwen, AST-based prompts outperform raw source (SRC) prompts across all coverage metrics. In figure 4.3, Branch coverage improves from 37.93% (SRC) to 55.59% (AST), line coverage rises from 45.23% to 62.34%, and method coverage increases from 67.8% to 73.28%. These results show that AST prompts provide better structural context, helping the model handle conditional logic and exercise more code and methods. The bar chart clearly shows the AST curve above the SRC curve in all metrics, with the largest improvement in branch coverage, highlighting the effectiveness of AST-based prompts for generating deeper and more comprehensive test cases.

Summary

In this analysis, we computed the average code coverage across all three LLMs (Gemini, GPT, and Qwen) for two types of prompts: AST-based prompts and raw source code (SRC) prompts. We considered three metrics branch coverage, line coverage, and method coverage and calculated the mean percentage for each metric across the models.

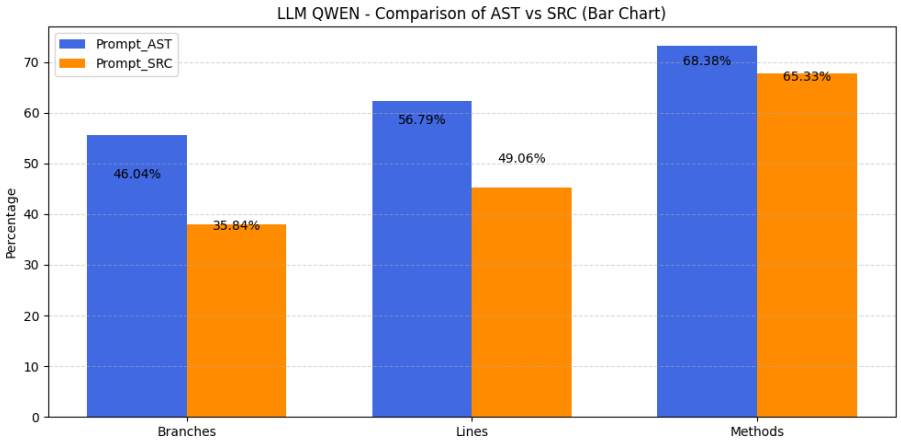

Figure 4.3: Prompt Coverage Comparison: Qwen

Table 4.2: Comparison of Metrics Across LLMs for AST+SRC and SRC Prompts

LLM	Prompt Type	Branches (%)	Lines (%)	Compilation Success (%)	Average Errors per Class
GEMINI	AST+SRC	46.04	56.79	61.90	5.64
	SRC	35.84	49.06	63.16	3.96
GPT	AST+SRC	47.80	67.13	72.22	2.17
	SRC	38.05	58.18	54.68	9.25
QWEN	AST+SRC	55.59	62.34	51.02	3.57
	SRC+SRC	37.93	45.23	38.57	12.94

The bar graph figure 4.4 shows the average code coverage across all LLMs for AST-based and SRC prompts. For each metric, branch coverage, line coverage, and method coverage AST-based prompts consistently achieve higher percentages than SRC prompts. Branch and line coverage show the largest improvements, indicating that AST prompts help the models better handle conditional logic and exercise more executable code. Method coverage also improves, though slightly, suggesting a broader reach in testing methods. Overall, the bar chart clearly highlights that AST-based prompts lead to more comprehensive coverage across all metrics.

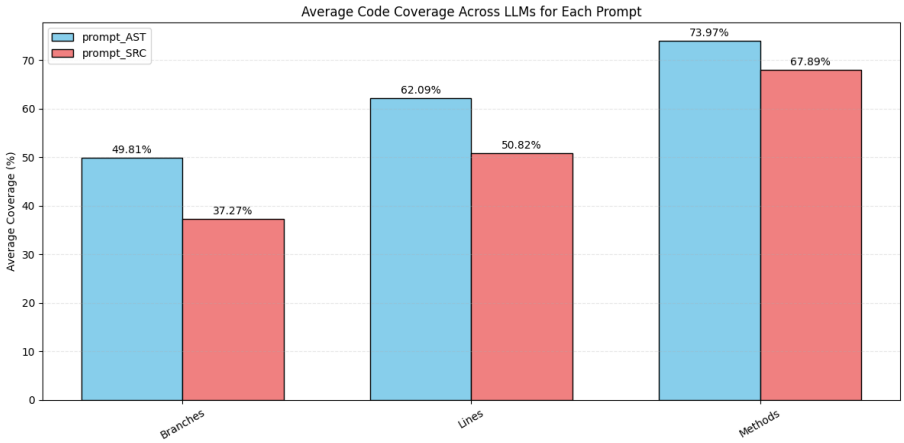

Figure 4.4: Prompt Coverage Comparison Among all The Models

4.2.2 Metric: Error

Table 4.3: Error Type Percentages Across LLMs for AST and SRC Prompts

LLM	Prompt	Cannot Find Symbol (%)	Syntax Error (%)	Incompatible Types (%)	Execution Error (%)
GEMINI	AST	87.31	22.64	11.09	12.24
	SRC	37.04	37.04	22.22	8.47
GPT	AST	39.27	17.42	15.45	5.52
	SRC	28.57	39.29	30.36	12.43
QWEN	AST	47.28	7.61	7.61	7.76
	SRC	34.21	9.97	12.97	14.08

Table 4.4: Error Type Percentages Across LLMs for AST and SRC Prompts After Normalizing

LLM	Prompt	Cannot Find Symbol (%)	Syntax Error (%)	Incompatible Types (%)	Execution Error (%)
GEMINI	AST	65.51	16.98	8.32	9.18
	SRC	35.35	35.35	21.21	8.08
GPT	AST	50.57	22.43	19.89	7.11
	SRC	25.82	35.51	27.44	11.23
QWEN	AST	67.30	10.83	10.83	11.04
	SRC	48.02	14.00	18.21	19.77

The first step is to ensure that the error percentages for each prompt type sum to 100%. This is done so that the pie charts later accurately represent relative proportions of different error types. The code creates a deep copy of the original dataset to avoid altering the raw data. Then, it iterates over each LLM (GEMINI, GPT, QWEN) and each prompt type (prompt_AST, prompt_SRC).For each error type within a prompt (e.g., cannot_find_symbol, syntax_error), the values are divided by the total sum of errors and multiplied by 100. This normalization makes the error distributions comparable across prompts and LLMs. Purpose: This step ensures consistency in visualization and prevents misinterpretation due to differing scales of raw error counts.

The error analysis shows clear differences between AST and SRC prompts across all three LLMs. For GEMINI in figure 4.5, AST prompts cause a high number of symbol-related errors (around 87%), while syntax and type errors remain low. SRC prompts reduce symbol errors but increase syntax and typing mistakes. For GPT in figure 4.6, AST prompts lower symbol errors to 39% with moderate syntax and type errors, whereas SRC prompts increase syntax and type issues. QWEN in figure 4.7 shows a more balanced pattern: AST prompts still have the highest symbol errors (47%), and SRC prompts reduce symbol mistakes but increase execution errors to 14%. Overall, AST prompts tend to produce more symbol resolution errors due to abstraction, while SRC prompts reduce these but can introduce syntax, type, or runtime errors. This highlights the trade-off in prompt design between symbol handling and overall correctness.

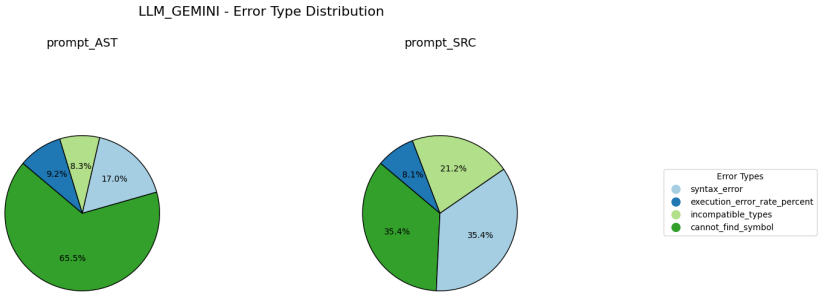

Figure 4.5: Error Percentage in Gemini

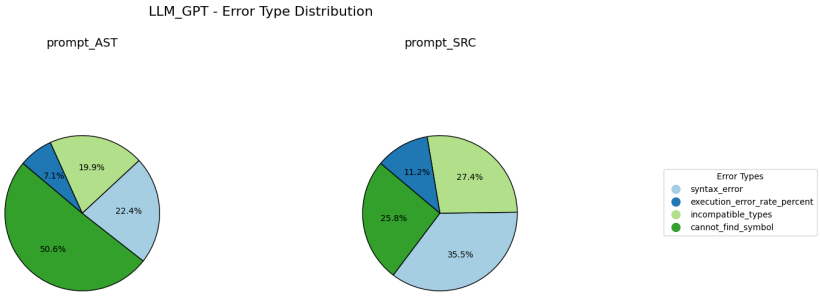

Figure 4.6: Error Percentage in GPT

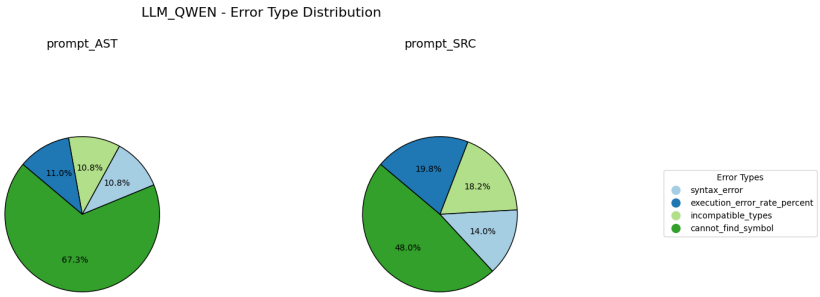

Figure 4.7: Error Percentage in Qwen

Results on Prompt Comparison (LLM Gemini)

The bar charts figure 4.8 show the error distribution for LLM GEMINI with AST and SRC prompts. The AST prompt produces a very high number of cannot_find_symbol errors (about 87%), while syntax and type errors remain low. The SRC prompt reduces symbol errors to around 37%, but syntax and type errors increase, with incompatible types around 22%. The line chart confirms these trends, showing that AST errors are dominated by symbol issues, while SRC errors are more balanced. Overall, these charts highlight that prompt design strongly affects the types of errors generated, with

AST favoring symbol resolution and SRC introducing more syntax and type mistakes.

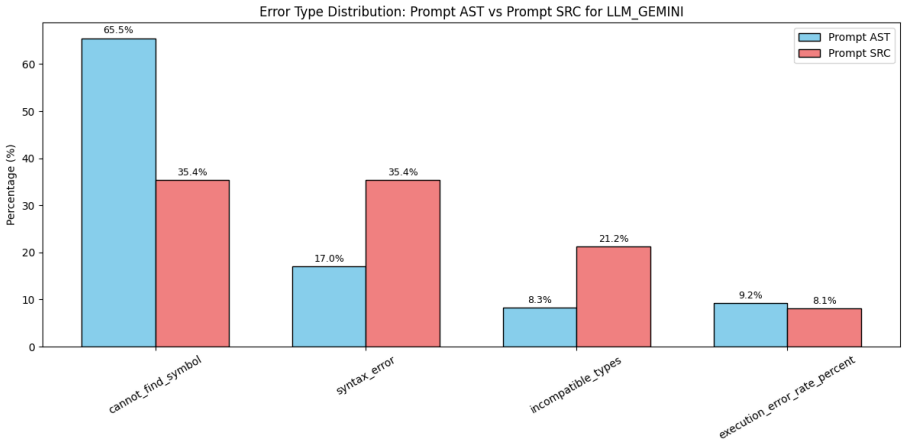

Figure 4.8: Prompt Error Comparison Gemini

Results on Prompt Comparison (LLM GPT)

The bar chart figure 4.9 show error trends for LLM GPT with AST and SRC prompts. The AST prompt has the highest proportion of cannot_find_symbol errors (39%), while syntax and type errors are lower (17% and 15%), and execution errors are minimal (5.5%). The SRC prompt reduces symbol errors (28.%) but increases syntax (39%) and type errors (30%), with execution errors rising to 12%. The line chart confirms these trends, showing AST errors dominated by symbols, while SRC errors are more balanced. Overall, AST prompts help with syntax and type correctness but struggle with symbols, whereas SRC prompts improve symbol handling at the cost of more syntax and type issues.

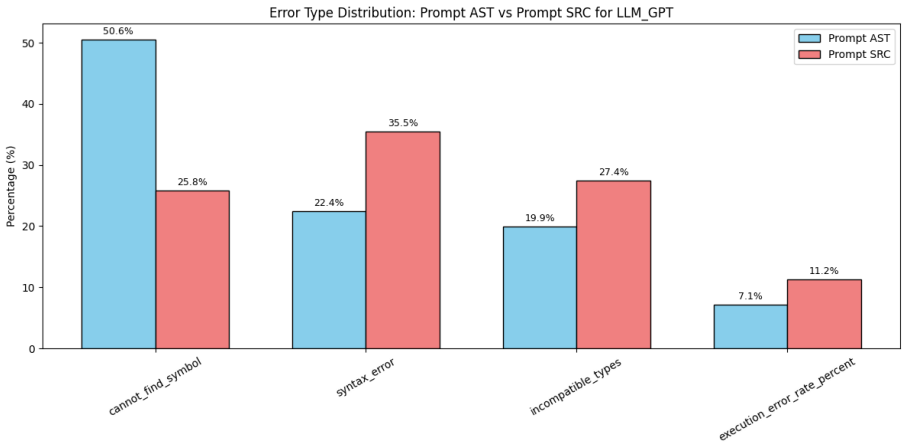

Figure 4.9: Prompt Error Comparison GPT

Results on Prompt Comparison (LLM GPT)

The chart figure 4.10 show error trends for LLM QWEN with AST and SRC prompts. With the AST prompt, cannot_find_symbol errors dominate (47%), while syntax and type errors are low (7.6% each) and execution errors are 7.8%. The SRC prompt reduces symbol errors (34%) but slightly increases type (13%) and execution errors (14%), with syntax errors remaining low (10%). The line chart highlights that AST errors are concentrated on symbols, while SRC spreads errors more evenly. Overall, AST minimizes syntax and type issues, whereas SRC improves symbol handling at the cost of slightly higher runtime and type errors.

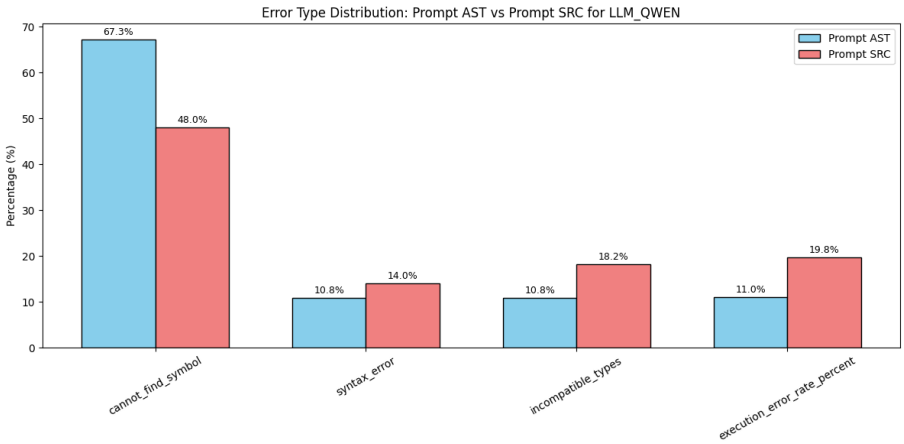

Figure 4.10: Prompt Error Comparison Qwen

4.3 Summary of Results

4.3.1 Evaluation Metrics

1. **Statement Coverage:** This metric measures the percentage of executable statements in the code that are executed by the generated test cases. We choose it because it provides a fundamental indicator of how thoroughly the LLM-generated tests exercise the code. High statement coverage ensures that most parts of the program logic are executed at least once, which helps in detecting errors that may otherwise remain hidden in untested statements. It is simple to compute, widely used in software testing, and provides a baseline for evaluating the effectiveness of automated test generation.

2. **Branch Coverage:** Branch coverage measures the percentage of decision points (like if, switch, or loop conditions) that are exercised by the generated tests. We choose this metric because it ensures that all possible execution paths in the

code are tested, not just individual statements. High branch coverage indicates that the tests can catch logic errors and edge cases that may occur along different branches of conditional statements.

3. **Compilation Success Rate:** This metric measures the proportion of generated test cases that compile successfully without errors. We use it because successful compilation is a prerequisite for executing tests. A higher compilation success rate indicates that the LLM is generating syntactically correct and semantically valid code, reflecting its ability to understand and produce correct programming constructs.

4. **Average Errors per Class:** This metric tracks the average number of errors (syntax, type, or runtime) per class in the generated tests. We include it because it provides a direct measure of the correctness and reliability of the generated tests. Fewer errors per class indicate that the LLM is producing cleaner, more maintainable, and executable test code.

4.3.2 Results

This section presents the evaluation results of the three LLMs (GEMINI, GPT, and QWEN) for automatically generated test cases using two prompt strategies: AST-based prompts (prompt_AST) and source-code-based prompts (prompt_SRC). The results are analyzed in terms of code coverage, compilation success, and error type distribution.

Results of Prompting Strategies (LLM Gemini)

The grouped bar chart 4.11 compares AST-based prompts with SRC-based prompts across four evaluation metrics for LLM GEMINI. For branch coverage, AST achieves 46.04%, while SRC lags behind at 35.84%. A similar trend is seen in line coverage, where AST reaches 56.79% compared to 49.06% for SRC. Interestingly, the compilation success rate shows the opposite pattern: SRC performs slightly better at 63.16%, while AST achieves 61.90%. Finally, when looking at the average errors per class, AST produces more errors (5.64) than SRC (3.96). The line chart reinforces these findings. The AST curve stays consistently above SRC for both branch and line coverage, showing its advantage in code coverage. However, SRC surpasses AST in compilation success rate and average errors, suggesting that while AST prompts improve coverage, they come with more compilation issues and errors.

Overall, for LLM GEMINI, AST prompts are stronger in test coverage, whereas SRC

prompts provide more stable compilations with fewer errors.

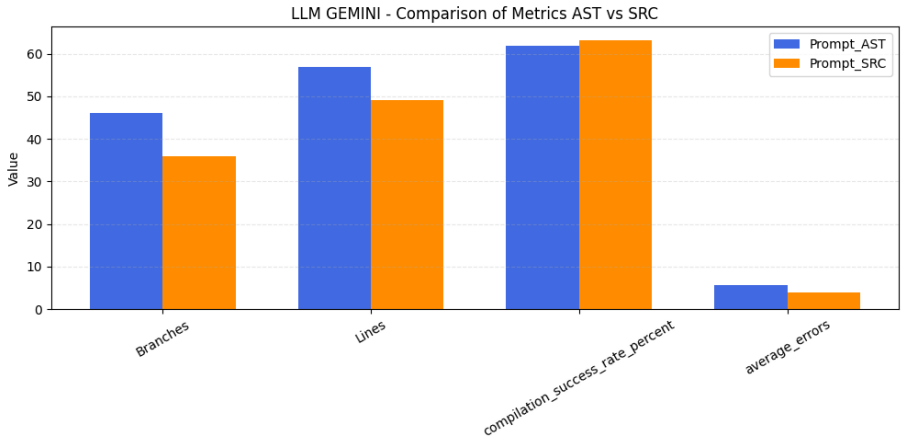

Figure 4.11: Metrics Comparison Gemini

Results of Prompting Strategies (LLM GPT)

The comparison of AST-based prompts and SRC-based prompts in LLM GPT 4.12 was evaluated across four metrics: branch coverage, line coverage, compilation success rate, and average errors. SRC prompting consistently achieved higher branch coverage and line coverage than AST prompting in LLM GPT. This shows that giving GPT direct access to raw source code enables it to generate test cases that cover a larger number of execution paths and lines.SRC prompting also outperformed AST prompting in terms of compilation success rate. The test cases generated by GPT with SRC prompts compiled more successfully, highlighting their practical advantage in real-world execution. On the other hand, AST prompting led to fewer average errors compared to SRC prompting in LLM GPT. Although coverage was lower, the structured AST representation helped GPT avoid inconsistencies and invalid outputs. In LLM GPT, SRC prompting provides broader and more executable test coverage, making it suitable when maximizing coverage is the goal. Meanwhile, AST prompting reduces generation errors, making it useful when correctness and stability are prioritized.

Results of Prompting Strategies (LLM Qwen)

AST prompting produced higher branch coverage and line coverage compared to SRC prompting in QWEN 4.13. This indicates that QWEN benefits from the structured nature of AST when it comes to exploring more execution paths and lines of code. In terms of compilation success, AST prompting again outperformed SRC prompting. This suggests that test cases generated by QWEN with AST inputs are more likely to compile successfully, pointing to better structural consistency in the outputs.

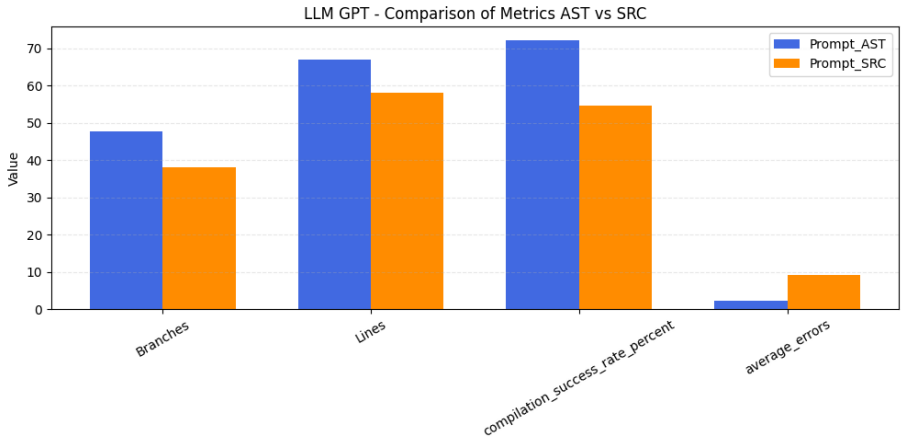

Figure 4.12: Metrics Comparison GPT

The average errors per class were significantly lower in AST prompting compared to SRC prompting for QWEN. SRC-based prompts introduced many more inconsistencies and invalid outputs, reducing their overall reliability. Unlike GEMINI and GPT, where SRC prompts dominated in coverage, QWEN performed better with AST prompts across most metrics. This shows that QWEN responds more effectively to structured AST inputs, achieving stronger coverage, higher compilation success, and fewer errors.

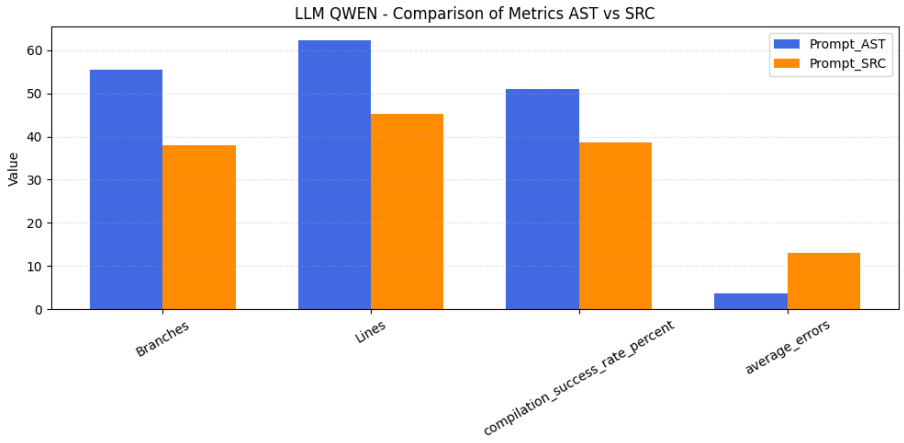

Figure 4.13: Metrics Comparison Qwen

4.4 Challenges and Limitations

Throughout this research, numerous challenges and limitations were encountered that might have influenced the results.

4.4.1 Challenges

- **Lack of Suitable Datasets:** Existing datasets primarily consist of isolated code snippets or individual methods, which are insufficient for evaluating project-level or class-level unit test generation and coverage analysis. Comprehensive datasets containing complete, real-world software projects are scarce.
- **Custom Project Collection:** One of the big challenges we faced was finding the right open-source Java projects to use from GitHub. We needed projects that were complete, had different levels of complexity, and were suitable for project-level evaluation. However, many repositories we came across were either incomplete, poorly documented, or didn't have proper test coverage, which made them difficult to work with. On top of that, each project had its own structure and setup, so we had to spend extra time preparing and validating them before they could fit into our evaluation process.
- **Dynamic Import Requirements:** One of the first challenges we faced was dealing with imports. Every generated test file needed a unique set of imports based on the source code and its path. Since no two files were the same, it became very difficult to create a uniform way of handling these imports. We often had to adjust them manually, which slowed down our workflow.
- **Running Java from Python:** Another big challenge was running Java projects directly from Python. Managing classpaths, dependencies, and test execution across two different environments wasn't straightforward. Small mistakes often caused failures, and the overall process felt unstable and complex to manage smoothly.
- **Manual Evaluation Effort:** Evaluating the test cases manually took a lot of time and energy. We had to check each generated test class, fix issues, and then run it to see if it worked. This repetitive process wasn't just inefficient but also made it easy to overlook mistakes. It quickly became one of the most exhausting parts of our experiment.
- **Automation Complexity:** We tried to solve some of these issues with automation, but building a reliable script was not simple. The script needed to handle cleaning, compiling, managing dependencies, and running test files without errors. Making sure all of this worked consistently was itself a challenge, and the automation is still a work in progress.

4.4.2 Limitations

- **Limited Project Scope:** Although we collected multiple Java projects, the number of projects and classes we tested was still limited compared to the huge variety of real-world software systems. This means our results may not fully represent how the approach would perform on much larger or more diverse projects.
- **Model Dependency:** Our evaluation relied heavily on the specific LLMs we used. Different models may behave differently, and improvements or updates in future models could change the outcomes. This dependency makes it harder to generalize our findings across all LLMs.
- **Evaluation Metrics Coverage:** While we measured important aspects like error counts, branch coverage, and compilation correctness, some qualitative factors like readability, maintainability, or long-term usefulness of the generated tests were not deeply analyzed. This leaves part of the quality picture unexplored.
- **Automation Gaps:**A fully automated workflow for handling imports, compiling, and running tests was not fully achieved. Because of this, some parts of the evaluation still required manual intervention, which could introduce small inconsistencies or human error.
- **Contextual Limitations:**Even with AST-based simplifications, providing enough context to the LLM without exceeding input limits remained challenging. As a result, some generated tests may have missed critical logic due to incomplete context.

4.5 Implications and Significance of Findings

4.5.1 Comparative Results of Prompting Strategies Across LLMs

The evaluation of AST-based prompts and SRC-based prompts across three LLMs GEMINI, GPT, and QWEN, reveals distinct patterns in how different models respond to structured versus source-based inputs. The analysis considers four key metrics: **branch coverage, line coverage, compilation success rate, and average errors per class**.

Branch and Line Coverage

For LLM GEMINI and LLM GPT, SRC prompts consistently achieved higher branch and line coverage, indicating that direct access to raw source code enables these models to generate test cases that explore more execution paths and lines of code. In contrast, LLM QWEN performed better with AST prompts, suggesting that its generation process benefits more from structured representations than from raw source code.

Compilation Success Rate

Both LLM GEMINI and LLM GPT favored SRC prompts, achieving higher compilation success rates and producing a greater proportion of compilable test cases. However, LLM QWEN displayed the opposite trend, with AST prompts yielding better compilation outcomes. This highlights model-specific sensitivities to the form of prompting.

Average Errors per Class

For LLM GEMINI and LLM GPT, AST prompts consistently reduced the number of errors, even though they limited coverage. For LLM QWEN, AST prompts again proved superior, leading to significantly fewer errors compared to SRC prompts, which introduced a much higher rate of inconsistencies and invalid outputs.

Overall Trade-off

For LLM GEMINI and LLM GPT, SRC prompting offers broader coverage and better compilability, making it more effective in scenarios where test completeness is prioritized. However, AST prompts remain useful when reducing inconsistencies is critical. For LLM QWEN, AST prompting is clearly more effective across all major metrics, emphasizing that QWEN benefits more from structured representations than raw source input.

The optimal prompting strategy depends on the LLM. While SRC prompts enhance coverage and compilation for GEMINI and GPT, AST prompts are more effective for QWEN, improving both coverage and correctness. This comparative evaluation underscores that the design of prompting strategies should be tailored to the characteristics of each LLM rather than assuming a one-size-fits-all solution.

Chapter 5

Conclusion

5.1 Restating Research Objectives and Questions

The general purpose of this research was to examine the impact that different prompts how engineering methods impact the quality and effectiveness of test cases automatically by large language models (LLMs). Specifically, the project was seeking to identify which prompting strategies result in greater code cov-verage, reduced error rates, and more stable outputs, hence making LLMs a better utility in automatic soft-ware testing. To guide this research, the following key research questions were formulated:

- How does the prompt structure and complexity affect the statement and branch coverage of test cases produced by LLM?
- Which of the prompting methods, Source Code and Source Code + AST, generates the most complete and accurate test cases?
- Are there prompting methods more robust across different LLMs like GPT and Gemini, or model-specific calibration is necessary?

These questions and objectives guided the design of the experiments, the analysis criteria, and the comparative analysis undertaken in the course of research. By revisiting them here, the findings and implications discussed subsequently can be set clearly in the context of the study's scope and objectives.

5.2 Summary of Key Findings

1. **Importance of Prompt Processing:**
   It was discovered that how it is set, as well as the quality, of a prompt signifi-

cantly determines its effectiveness with which language models are able to perform unit testing. No single technique of prompting exceeded in all contexts. Extremely composed prompts like Chain-of-Thought (CoT) and Tree-of-Thought (ToT) added the most amount of hallucinations, wrong imports, and semantic errors, and far too elementary prompts like zero-shot gave incomplete guidance, causing them to render incompletely correct answers. This points to the necessity to craft and process prompts in a way that achieves a balance between simplicity, guidance, and clarity.

2. **Findings for LLM GEMINI:**
   For LLM GEMINI, the SRC prompting strategy demonstrated higher effectiveness in terms of branch and line coverage, as well as compilation success rate, compared to AST prompts. However, AST prompts resulted in fewer average errors per class, suggesting a trade-off between maximizing coverage and maintaining stability. Error analysis revealed that AST prompts struggled with symbol resolution, whereas SRC prompts introduced more syntax and type inconsistencies.

3. **Findings for LLM GPT:**
   For LLM GPT, the AST prompting strategy provided the best overall performance. It achieved higher line and branch coverage, the highest compilation success rate among all models, and significantly fewer average errors per class compared to SRC prompts. In contrast, the SRC strategy showed lower compilation success and produced more syntax and type errors, indicating that GPT benefits more from structured AST input rather than raw source code.

4. **Findings for LLM QWEN:**
   For LLM QWEN, the AST prompting strategy again outperformed SRC prompts in most evaluation metrics. AST prompts led to higher branch and line coverage, better compilation success, and substantially fewer average errors per class. SRC prompts, although slightly improving symbol resolution, resulted in a significant increase in errors, including runtime issues. This indicates that QWEN is more sensitive to raw source code input and performs more reliably with AST-based prompting.

5. **Overall Findings:**
   Overall, the study demonstrates that prompting strategies have a decisive impact on LLM performance in unit test generation. GEMINI benefits more from SRC prompts for maximizing coverage, while GPT and QWEN perform better with AST prompts, prioritizing stability and correctness. The findings highlight a

fundamental trade-off between coverage and correctness, showing that no single prompting approach is universally optimal across all models. Instead, the choice of prompt strategy should be tailored to the specific strengths and weaknesses of the LLM being used.

5.3 Discussion of Implications

The findings of this research carry several important implications for both academia and industry. From an academic perspective, the comparative analysis between AST-based and SRC-based prompting strategies deepens the understanding of how prompt design shapes the performance of large language models in test case generation. It demonstrates that while source-based prompts maximize coverage and compilability, abstract syntax tree prompts offer the advantage of reducing inconsistencies, highlighting a trade off that future research can build upon. From an industrial perspective, the study provides actionable insights for software engineers and quality assurance teams who wish to integrate LLMs into automated testing workflows. By selecting prompting strategies that align with project needs—whether prioritizing coverage or correctness organizations can enhance the reliability and efficiency of their testing processes. More broadly, the research emphasizes the transformative potential of LLMs in software engineering, suggesting that careful prompt engineering can bridge the gap between theoretical capabilities and real-world applicability.

5.4 Contributions of the Research

This research makes several key contributions to the field of test case generation with large language models (LLMs). These contributions can be summarized as follows:

- **Empirical Insights:** By systematically evaluating three LLMs (Gemini, GPT, and Qwen) across four metrics, this study provides a clearer understanding of how prompting strategies affect test case generation quality.
- **Comparative Evaluation:** The research highlights the trade-offs between source-based (SRC) and AST-based prompting, showing that SRC excels in coverage and compilability, while AST reduces generation errors.
- **Methodological Contribution:** The study introduces a structured evaluation framework combining coverage metrics with practical success rates, offering a balanced way to assess LLM-driven test case generation.

- **Practical Relevance:** The findings can inform software engineers and researchers about the strengths and weaknesses of different prompting strategies, enabling better integration of LLMs into software testing workflows.
- **Foundational Contribution:** By documenting both strengths and limitations, the research lays the groundwork for future studies to refine prompting strategies and expand evaluation metrics in more complex, real-world settings.

5.5 Acknowledging Limitations

Like all research, this study is not without its limitations. First, the evaluation was conducted on a limited set of large language models, which may not fully capture the diversity of performance across the broader landscape of available models. As newer models continue to emerge, their behavior may differ significantly from those studied here.

Second, the scope of the work was restricted to a specific set of evaluation metrics, namely branch coverage, line coverage, compilation success rate, and average errors. While these metrics provide valuable insights, they may not represent the complete spectrum of factors relevant to test case generation, such as maintainability, scalability, or integration into real world workflows.

Third, the study relied on synthetic and controlled datasets rather than large-scale, production level software systems. This constraint may limit the external validity of the findings, as results might vary under more complex or domain specific conditions.

Finally, technical challenges, including prompt design constraints and resource limitations, shaped the experimental setup and may have influenced outcomes. Acknowledging these limitations helps contextualize the results and highlights areas where future research can extend and refine the findings.

5.6 Suggestions for Future Research

While this study has provided meaningful insights into prompt engineering for test case generation using large language models, several areas remain open for further exploration. Future research could expand the scope by experimenting with a wider range of models, including newer architectures or domain specific LLMs, to better understand how model design impacts test generation. Additionally, investigating hybrid prompting strategies that combine structural and contextual information may

yield more balanced results in terms of correctness and coverage.

Another important direction would be the creation of standardized benchmark datasets specifically tailored to evaluating LLMs in software testing scenarios. Such datasets would enable fairer comparisons across models and methodologies, ensuring that improvements are measured consistently. Researchers may also consider examining how fine-tuning or reinforcement learning techniques can further optimize LLM performance in generating reliable and maintainable test cases.

Finally, integrating human feedback loops into the test generation process presents an exciting opportunity. By combining the strengths of automated generation with expert validation, it may be possible to develop more practical, industry-ready solutions. Together, these directions point toward a richer understanding of LLM capabilities and their potential to transform software quality assurance practices.

5.7 Final Reflections

This research journey has been both challenging and rewarding. At the start, the goal was clear: to explore how different prompting strategies influence large language models in generating unit tests. Along the way, unexpected patterns emerged, such as the trade-off between broader coverage and maintaining syntactic correctness, which deepened my understanding of how LLMs process structured and unstructured input.

Beyond the technical findings, this process also shaped our perspective as a researcher. Navigating through errors, adjusting methodologies, and analyzing results required persistence and adaptability, teaching me that setbacks often lead to valuable insights.

In the broader research community, this study contributes to the growing conversation about effective prompt engineering strategies for software testing. While the results highlight specific strengths and weaknesses of AST and SRC prompts, they also underscore the importance of tailoring approaches to individual models rather than assuming one universal solution.

Looking back, the journey has been more than just an academic exercise it has been an opportunity for personal and intellectual growth. The lessons learned here extend beyond this thesis, preparing me to approach future challenges with a deeper appreciation of the balance between innovation, experimentation, and practical application.

5.8 Concluding Remarks

In this work, we have helped to shed light on one important aspect of using large language models (LLMs) for software testing the way we craft prompts can significantly impact the quality of test cases the models generate. Through careful consideration of different prompting methods, we've reached insightful information that can have practical ramifications for developers and organizations wanting to harness the potential of LLMs for automatic testing. The findings of this study provide practical guidance on how to better use these models, helping to ensure that the test cases developed are not only comprehensive but also accurate.

While we've made some important advances, there is still much to be learned here. The research indicates exactly how important it is to thoughtfully craft the prompts we provide and how they can influence the results. With LLMs continuing to evolve and becoming more deeply embedded in development workflows, the insights from this work will hopefully guide future advancements in this area.

From figuring out how to better prompt these models to understanding how to make them perform better on more tasks and codebases, there's a lot of room to grow. The potential for LLMs to change how we test and develop is vast, and We hope that this research is one step along the way to a future in which LLMs are more intelligent, efficient, and accessible to developers across the globe.

References

[1] S. Gao et al., *The prompt alchemist: Automated llm-tailored prompt optimization for test case generation*, 2025. arXiv: 2501.01329 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2501.01329

[2] R. Lingampally, A. Gupta, and P. Jalote, "A multipurpose code coverage tool for java," in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007, 261b–261b. DOI: 10.1109/HICSS.2007.24

[3] M. Mosbach, T. Pimentel, S. Ravfogel, D. Klakow, and Y. Elazar, *Few-shot fine-tuning vs. in-context learning: A fair comparison and evaluation*, 2023. arXiv: 2305.16938 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2305.16938

[4] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng, *Automatic prompt optimization with "gradient descent" and beam search*, 2023. arXiv: 2305.03495 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2305.03495

[5] A. Sabbatella, A. Ponti, I. Giordani, A. Candelieri, and F. Archetti, "Prompt optimization in large language models," *Mathematics*, vol. 12, p. 929, Mar. 2024. DOI: 10.3390/math12060929

[6] D. Soylu, C. Potts, and O. Khattab, *Fine-tuning and prompt optimization: Two great steps that work better together*, 2024. arXiv: 2407.10930 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2407.10930

[7] W. Sun et al., *Abstract syntax tree for programming language understanding and representation: How far are we?* 2023. arXiv: 2312.00413 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2312.00413

[8] B. Wang et al., *Towards understanding chain-of-thought prompting: An empirical study of what matters*, 2023. arXiv: 2212.10001 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2212.10001

[9] H. Yu and J. Liu, *Deep insights into automated optimization with large language models and evolutionary algorithms*, 2024. arXiv: 2410.20848 [cs.NE]. [Online]. Available: https://arxiv.org/abs/2410.20848

[10] Q. Zhang, Y. Shang, C. Fang, S. Gu, J. Zhou, and Z. Chen, *Testbench: Evaluating class-level test case generation capability of large language models*, 2024. arXiv: 2409.17561 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.17561

[11] T. Zheng et al., *The curse of cot: On the limitations of chain-of-thought in in-context learning*, 2025. arXiv: 2504.05081 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2504.05081