TIGER: Testing and Improving Generated Code with LLMs

BY

LORENZO GALLONE
B.S., Politecnico di Torino, Turin, Italy, 2023

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2025

Chicago, Illinois

ACKNOWLEDGMENTS

I wish to express my heartfelt gratitude to my thesis committee for their guidance and support throughout my research. Your insights have been instrumental to both my academic and personal growth.

A special thank you goes to my parents and my sister for their unwavering love and constant encouragement, which have been fundamental to my achievements. My sincere and heartfelt gratitude goes to Martina, whose constant support, endless patience, unshakable encouragement, and sincere belief in me have been crucial to every step of this journey. Her love and strength have been my greatest motivation

I am sincerely grateful to all the wonderful people I've met here in Chicago, especially those from the "palazzina." Your friendship and warmth made my experience memorable and enjoyable. A warm and special thank you to Francino, with whom I've shared countless hours of studying and beyond, making my American experience even more meaningful.

A heartfelt thanks also goes to Ste, Fede, Andre, Totta, Pitta, and Deco who, despite the distance during this last year, have continued to play an important role in my life. They have always been there whenever I needed to talk or reach out, regardless of the reason.

Finally, I am genuinely thankful to everyone I've encountered on this journey for their invaluable contributions, inspiration, and support.

LG

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF TABLES (continued)

LIST OF FIGURES

LIST OF ABBREVIATIONS

UIC	University of Illinois at Chicago
SR	Succes Rate
LOC	Line of Code

SUMMARY

This thesis presents a test-driven framework for enhancing the reliability of code generated by Large Language Models (LLMs), focusing on real-world applicability and minimal developer assistance. The system is designed to simulate a realistic development environment where no ground-truth implementations are available to the model, relying exclusively on textual artifacts such as documentation, docstrings, and test outcomes. This constraint ensures that every generated function is derived from semantic understanding rather than replication or pattern-matching.

A core innovation of this work is the integration of an *iterative refinement loop*, which introduces structured feedback into the code generation process. After producing an initial function from a natural language prompt, the model's output is immediately tested. If failures occur, relevant error signals are extracted and used to update the prompt, allowing the model to revise its solution. This loop continues until the implementation passes all associated tests or a retry limit is reached. The system thus mirrors a human-like workflow of test-driven development and debugging.

To assess the contribution of this iterative process, the same framework is also evaluated in a non-iterative configuration, where each function is generated only once based on its prompt and tested without revision.

The evaluation is conducted on entire Python repositories—not isolated functions—making the task significantly more complex. Functions are embedded in larger software structures, depend on shared state or class behavior, and are often indirectly tested through multi-layered scenarios. The system parses these repositories to extract structural metadata, resolve function-to-test mappings, and build context-aware prompts that support both initial generation and iterative correction.

The results demonstrate that embedding LLMs into a feedback-rich environment substantially increases their capacity to produce robust, test-passing code. Despite added computational cost, the iterative approach leads to higher success rates across a diverse range of codebases, showing that language models, when guided by empirical signals and properly contextualized, can evolve from static generators into adaptive agents capable of producing functionally correct and maintainable code.

CHAPTER 1

INTRODUCTION

The advent of Large Language Models (LLMs) impacted the world of software development in various ways, mainly by enabling automatic code generation through natural language descriptions [1; 2]. The models have the ability to reduce the time required for hand-coded programming and improve development practices. Studies by [3] revealed that programmers using AI-assisted coding tools saw an average improvement in code quality by 18% compared to standard practices, thus validating the productivity benefits offered by LLMs. However, in spite of such benefits, ongoing code quality and code produced by LLMs' reliability problems persist, despite their extensive codebases [2].

1.1 Context and Motivation

The creation of standard code for LLM tends to use a one-shot approach [2]. Here, the model outputs code in response to a description or docstring in text. The approach is computationally inexpensive, but no system for giving and receiving feedback is included. Therefore, any logical, syntax, or semantic errors in code produced by such models must be discovered and corrected by the user on their own.

A good alternative is an iterative refinement approach. With this method, the LLM receives feedback from unit tests with caution and uses that feedback to gradually refine the function it constructs [4]. This approach is designed to make the code more understandable and improved by incorporating test feedback into the construction process. This decreases the amount of assistance humans require to correct and improve issues.

Current investigations have shown that using LLM-assisted development has the ability to significantly reduce coding and debugging time, leading to increased developer productivity [5; 3].

1.2 Research Objectives

The goal of this thesis is to demonstrate that Large Language Models (LLMs) can produce high-quality, test-passing Python code when placed in a structured generation loop that closely resembles a developer's test-driven workflow. The focus is on generating code without ever exposing the LLM to existing implementations. Instead, the model receives only high-level natural language artifacts—such as README files, reStructuredText documentation, and docstrings—along with pass/fail signals from the test suite.

The system is designed to replicate a realistic development setting, where a developer writes a function based solely on documentation and relies on tests to debug and refine their solution. In this scenario, the LLM is tasked with generating an initial function based on a prompt constructed from the available textual context. The generated code is then inserted into the codebase and executed against its corresponding unit tests. If the function fails, the system analyzes the test output, summarizes the errors, and constructs a refined prompt containing this feedback. The LLM is then invoked again to generate a revised implementation. This process continues iteratively until the function passes all relevant tests or a retry threshold is reached.

A simplified single-pass version of the system, where the LLM is invoked only once without iterative refinement, is used solely for comparison purposes in the evaluation chapter. It serves to quantify the benefits of the proposed iterative approach but is not a central focus of the study. The thesis instead concentrates on showing how test-driven, documentation-guided iteration enables LLMs to produce reliable, functional code without requiring access to existing implementations or manual corrections.

Evaluation Criteria

After defining the two code generation approaches, it is crucial to establish how their effectiveness will be measured. This study aims to determine whether iterative refinement provides a tangible benefit over single-pass generation, focusing on the actual functionality of the generated code rather than just its syntactic correctness.

To facilitate a meaningful comparison, the evaluation will utilize functional testing tools such as PyTest, JUnit, and a set of unit testing tools. These tests move beyond syntax checks and review whether the output of the function actually complies with the expected performance standards.

Key Research Questions

The key aspects under investigation include:

- **Functional correctness**: Is the output function effectively passing all functional tests, ensuring that it behaves exactly as expected?

- **Effectiveness of iterative refinement**: Is the probability of finding a correct solution substantially higher if the first try fails and the operation is redone with the function?
- **Efficiency**: What iterative methods are required in the generation of a fully functional item, and do such methods place a heavy overhead compared with generation with a single pass?

Through a careful analysis of such variables, the study seeks to determine if repeated application of a function justifies the effort required or if a properly designed single-pass generation technique is sufficient in providing stable and working code.

1.3 Thesis Structure

This thesis is divided into six chapters, each addressing a different facet of Large Language Model (LLM)–driven code generation and validation:

1. **Chapter 2: State of the Art** reviews current developments in LLM-based code generation. It explores fundamental architectures, training paradigms, and applications of LLMs, highlighting ongoing challenges such as erroneous outputs, integration complexities, and scalability. The chapter also examines prevailing evaluation strategies using frameworks like PyTest.
2. **Chapter 3: Proposed System Architecture** describes the architectural framework developed for this research, detailing the iterative refinement process. It clarifies how both approaches fit into a test-driven pipeline that prioritizes correctness over raw generation speed.

3. **Chapter 4: Implementation** provides a technical account of the system's components. It discusses how LLMs are prompted, how source code is analyzed, and how repository information is extracted. It also explains how the system integrates and evaluates newly generated code against existing test suites.

4. **Chapter 5: Experimental Methodology and Self-Reflection** outlines the design of the experiments, including selection criteria for repositories, testing workflows, and the metrics used to evaluate performance. Additionally, the chapter contains a reflective section wherein the author examines the learning outcomes and improvements gained while conducting this study.

5. **Chapter 6: Conclusions and Future Work** presents a consolidated summary of the research findings, highlighting key benefits and limitations of the iterative refinement approach. The chapter concludes by proposing potential research directions, including advanced error-handling strategies and the extension of the current system to broader, more complex contexts in code generation.

This structure facilitates an in-depth and quantitative comparison of different LLM-based code generation techniques, focusing particularly on iterative refinement and its impact within practical software engineering scenarios.

CHAPTER 2

STATE OF THE ART

2.1 Large Language Models for Code Generation

Large Language Models (LLMs) have significantly advanced natural language processing (NLP) and, in the last time, software development by not only teaching the machines how to understand and produce human language-based text but also by producing alternatively, automatically written code in several programming languages. These models that are trained on large datasets to understand and generate text in a natural language and implement the code in many programming languages, they can read and write code that a normal coder can write without any exaggeration in his or her language [1; 2]. Their capability of understanding and implementing human-readable instructions into functional code has been a very good tool for developers. It has made computer-assisted functions such as code completion, bug fixing, and refactorization easier.

Architecture and Training of LLMs

Large Language Models (LLMs) are a class of models developed using deep learning techniques, primarily based on transformer architectures [6]. These models employ self-attention mechanisms to capture relationships between elements in the input, enabling them to produce coherent and contextually appropriate language outputs. Their training involves processing vast volumes of textual data, allowing them to learn complex syntactic patterns and semantic structures.

For instance, OpenAI's GPT-3 was trained on a dataset comprising approximately 410 billion tokens sourced from the web, books, and Wikipedia [1]. Subsequent advancements have led to the development of models specifically fine-tuned for code generation. Examples include OpenAI Codex and DeepMind's AlphaCode, which were trained on extensive corpora of programming-related content [2; 4]. These specialized models demonstrate enhanced capabilities in generating software solutions tailored to specific tasks.

Nevertheless, despite these improvements, significant challenges remain. Issues such as hallucinated outputs, incorrect logic in generated functions, and security vulnerabilities persist [7]. These limitations raise important concerns regarding the reliability and robustness of LLMs in real-world software development scenarios.

Fine-Tuning and Domain Adaptation for Code Generation

Fine-tuning is an approach that adapts a pre-trained Large Language Model (LLM) to a specific domain or task by continuing its training on a targeted dataset. This method enables the model to internalize domain-specific coding practices, adhere to particular performance constraints, and follow established architectural or stylistic conventions. As a result, fine-tuned LLMs demonstrate improved effectiveness in generating contextually appropriate and syntactically correct code when compared to general-purpose models.

Fine-tuning is particularly beneficial in:

- **Domain-Specific Code Generation:** Specialist models trained on databases such as those related to cybersecurity, blockchain, or scientific computing can utilize domain-specific syntax, security best practices, and optimizations [8].
- **Enterprise Codebases:** Companies fine-tune LLMs using internal software repositories to ensure uniformity. This process enforces company coding standards, architectural constraints, and security policies [9].
- **Performance Optimization:** Fine-tuned models trained on high-performance computing benchmarks have shown significant improvements. A study by [4] demonstrated that fine-tuned AlphaCode models achieved performance comparable to mid-level human programmers in competitive programming challenges, significantly surpassing baseline models trained without domain-specific adaptation.

Applications in Code Generation

The use of large language models in software engineering is a novel approach to the programming process, which has significantly altered the relationship between developers and code. Such models are able to perform the translation of natural language explanations into executable code, thereby reducing the cognitive effort required from developers and accelerating the software development process [2]. A prominent example is GitHub Copilot, which integrates OpenAI Codex to provide real-time code suggestions directly within IDEs. This feature has demonstrated its effectiveness in improving coding efficiency, although security concerns remain a topic of discussion [7].

Auto-completion has become an increasingly significant application of LLMs. Beyond simple code correction or enhancement, these models allow developers to create new software components by providing high-level specifications in natural language. Studies have shown that auto-completion and AI-assisted bug fixing are most effective when trained on high-quality datasets and when users provide clear prompts. Furthermore, domain-restricted models improve the reliability and security of generated code, as demonstrated in recent evaluations of OpenAI Codex and similar tools [3; 10].

Unlike single-shot code generation, our study explores an iterative approach, where the model refines its output based on structured test feedback. This aligns with recent research emphasizing the importance of execution-based evaluation and adaptive refinement.

2.2 Challenges in Generating Structurally Complex Code

While Large Language Models (LLMs) have demonstrated notable proficiency in generating accurate standalone functions, their effectiveness significantly diminishes when applied to more structurally complex codebases. Such contexts involve multiple interdependent functions, shared variables, class hierarchies, and intricate control flows, all of which demand a coherent understanding of the broader software architecture.

As the size and complexity of the application grow, generating modular, well-integrated components becomes increasingly challenging. Current models often struggle to maintain consistency across related code units, properly manage persistent state, or ensure compatibility with existing API structures. These limitations underscore a fundamental weakness of LLMs when faced with the demands of real-world software engineering beyond isolated function generation.

Inconsistency in Multi-Function Codebases

LLMs are by nature stateless, meaning they do not have a full or global context at a certain moment. When client modules like binary-to-text decoders or image converters are auto-generated, verbose and confusing module names can be a problem. Another issue is missing or unknown variable names or syntax errors in the final code. Additionally, some components may be logically inconsistent, making debugging more challenging. As highlighted by Sobania et al. [10], Codex-generated modules often fail to maintain internal coherence when generating multi-function programs, requiring manual intervention to correct structural inconsistencies.

API Integration and Handling Concerns

Error management and authentication are becoming crucial components of system stability as software development depends more and more on external APIs. LLMs, however, have trouble with organized error handling and version management. This problem was found in Codex-generated API interactions by Sobania et al. [10], who found that more than 60% of RESTful API requests had improper authentication procedures or handled replies incorrectly, causing security issues.

State Management Challenges

Databases, global variables, or object-oriented programming paradigms are used in many applications' software architectures to track permanent state. Managing stateful components presents different issues depending on the programming language. However, memory allocation and resource management become inefficient due to LLMs' inherent inability to track communicated information across successive function generations [11].

Approaches to Address These Challenges

To address these challenges, researchers are exploring hybrid approaches that integrate LLM-generated code with:

- **Static Analysis Tools:** These tools automate the detection of syntactic and semantic errors based on predefined rules, improving the reliability of generated code [11; 2].
- **Property-Based Testing:** This technique ensures that function outputs align with expected behaviors, supplementing traditional unit testing methods [12].
- **Memory-Augmented LLMs:** Recent advancements introduce memory-enhanced architectures that allow models to retain contextual information across multiple function generations, improving coherence in long-term code synthesis [11; 4].

These methods aim to reduce manual debugging efforts and enhance LLM applicability in enterprise software development.

2.3 Automated Testing and Validation

While LLMs have shown remarkable promise in code generation, ensuring the correctness and reliability of generated code remains a significant challenge. Without rigorous validation mechanisms, LLM-generated code can introduce bugs, security vulnerabilities, and logic errors. To mitigate these risks, automated testing techniques have become a critical component of evaluating generated code [2].

Test-Driven Development

Test-Driven Development (TDD) is a widely known software engineering technique in which test cases are written even before the actual implementation of a function [13]. The development process here becomes more strict, and its essence is to ascertain whether the created code is meaningful or not with respect to predefined targets. According to recent studies, integrating TDD into LLM-supported code generation might have a positive effect on the correctness and stability of the model regarding the generated functions since the model is predetermined to implement well-defined behaviors [2].

Evaluation Metrics

Evaluating LLM-generated code presents unique challenges, as traditional NLP metrics such as BLEU and ROUGE [14] fail to capture functional correctness. Several studies indicate that high BLEU scores do not necessarily correlate with successful execution. For instance, Hendrycks et al. [15] found that models achieving top BLEU scores still generated code that was either syntactically incorrect or failed execution in over 40% of cases. Similarly, Chen et al. [2] highlighted that BLEU and similar metrics fail to assess logical correctness, reinforcing the need for execution-based evaluation.

To address these limitations, execution-based evaluation metrics have been proposed as a more reliable alternative. In this study, we assess model performance using success rate (equivalent to Test Accuracy) and failure rate, along with additional factors related to efficiency and refinement process. These metrics provide a more comprehensive view of both correctness and the overall effectiveness of the generation process.

This execution-focused methodology ensures a more practical assessment of LLM-generated code in real-world development scenarios.

Challenges in Code Validation

Despite advancements in automated testing, several shortcomings remain. The code produced by LLMs might pass basic syntactic checks but still contain logical faults, leading to faulty program behavior. In addition, testing coverage and robustness are subject to the quality of test case generation. There is evidence to suggest that while testing-by-example practices improve accuracy, they do not ensure the elimination of all potential faults, necessitating subsequent alterations and human checks [2; 4].

Error Handling in Code Generation

A persistent challenge in LLM-based code generation is handling errors effectively. Unlike traditional software development, where debugging is performed manually, LLM-generated code requires systematic strategies to detect, diagnose, and correct errors. Errors can manifest in various forms, including syntax errors, runtime exceptions, and logical inconsistencies [12; 7].

Error Detection and Analysis

To guarantee that the output code meets functional standards, researchers have examined automatic error detection tools. Unit testing tools and code analysis tools help identify failure instances with useful information about the positions and causes of the failures in the output code [2]. Classifying errors into categories such as syntax errors and logical errors helps scholars maximize the feedback loop required for the improvement of large language models' output.

Feedback Mechanisms and Iterative Refinement

Recent studies suggest that structured feedback mechanisms can significantly improve the quality of LLM-generated code. Rather than discarding faulty outputs, an iterative process can be employed where models receive failure reports and attempt to refine their responses accordingly [12]. This method has been explored in research on adaptive code refinement, where LLMs adjust their outputs based on structured test feedback to progressively converge toward functionally correct implementations [10].

Notwithstanding, the ongoing improvement of code poses computational and practical issues. While certain errors will be correctable through iterative improvement, others will necessitate external help in the manner of additional context or modifications to trigger design. This study seeks to identify the optimal ways of merging automation and human oversight in a bid to maximize the preferred results.

2.4 Related Works

The area of automatic code generation has seen remarkable advancements over the past few years, mainly due to the rapid evolution of large language models (LLMs). Current research focuses on two major research directions: testing methodology-guided code generation and generation guided by natural language specifications.

Generating Code from Tests

One of the more significant topics of debate is related to the generation of code entirely from the given test specifications [2; 13]. In this context, the code is generated by the model that is then validated by an exhaustive test suite. Although this approach ensures a level of functional correctness, it is also associated with significant limitations. More specifically, in cases where the tests are poorly defined or not robust enough, the code generated can pass the tests and yet be buggy. In addition, without clear semantic guidance, models often generate code that is syntactically correct but differs from the intended functionality.

Generating Code from Program Specifications

One major area of research is on code generation directly from specifications stated in natural language [2; 9]. The approach mirrors how developers usually describe software requirements and has picked up popularity with the development in large language models' ability to understand text descriptions. The general idea is to extract relevant information from specification documents, functional mappings, or test cases to create detailed prompts that guide the code generation process.

Many studies have tried to improve code generation at the repository level using this approach. For instance, RepoCoder [16] leverages a cyclical retrieval-generation framework that combines a retrieval model with an LLM in order to generate code from relevant segments that are already present within the repository. On the other hand, CatCoder [17] uses type-context information from statically typed languages like Java and Rust, applying static analysis for extending prompts with pertinent method signatures and data structures.

Advancing Repository-Level Code Generation: Our Approach

While previous methods have shown promising results, our method includes a number of important innovations that distinguish it from RepoCoder, CatCoder, and evaluation systems like HumanEval [2] and APPS [15]:

1. Our design is carefully crafted to address the challenges posed by structurally complex repositories, where functions, classes, and shared variables are distributed across multiple files and are tightly coupled through inter-component dependencies. In these settings, generating code requires not only syntactic correctness, but also compatibility with existing architectural elements and the ability to integrate seamlessly into the broader software system. This stands in contrast to popular benchmarks like APPS and HumanEval, which operate on isolated, self-contained functions with clearly defined problem statements and no need for contextual integration. While such benchmarks are valuable for assessing

isolated code generation capabilities, they do not reflect the complexities of real-world software engineering, where correctness also depends on structural coherence and contextual awareness.

2. As a contrast with approaches like RepoCoder and CatCoder, which draw upon existing code snippets within repositories and present them for the model's aid, our method produces code entirely from documentation. The model draws upon README files ('.md'), reStructuredText files ('.rst'), function signatures, and docstrings but not upon pre-existing implementation code segments. This makes the resulting code inherently informed by high-level descriptions and not directly derived from comparable code, thus better mimicking the process by which a function would have to be built from scratch according to the documentation provided.

3. Our evaluation methodology relies on a combination of static analysis and test execution. While static analysis—using Python's Abstract Syntax Tree (AST) module—supports the extraction of structural elements such as function signatures, argument types, and docstrings, our primary metric for assessing model performance is the success rate, defined as the percentage of generated functions that pass all associated tests. This metric directly captures the functional correctness of the output, offering a more meaningful evaluation than similarity-based scores. In contrast, many prior studies rely on BLEU, edit distance, or token-level overlap with reference implementations, which may yield high scores even when the generated code is syntactically correct but functionally invalid [2; 15]. By

grounding evaluation in execution results, our approach ensures that code is not only well-formed, but also operationally sound within its target context.

4. We follow the iterative refinement practice where the process involves the reassessment of data gathered from failed tests and the subsequent use for improving the prompt. The practice is critical in driving the model towards the creation of better outputs. The feedback process is a core component of our process, setting our approach apart from others where the practice depends on single generations or single designs without the inherent process for improvement.

A major drawback with benchmarks like HumanEval and APPS is their focus on decontextualized, well-delineated issues likely not representative of real-world project environments. Our approach, by contrast, directly involves existing codebases with the need for the generated code to be merged into the overall codebase while also addressing inter-function and inter-module dependencies. Our end-to-end validation process using a testing-based approach also provides a better guarantee concerning the quality of the code.

The features outlined represent a significant advancement within the automatic code generation space with significantly increased generalization and robustness relative to existing approaches. By avoiding reliance upon the assumption of the presence of pre-existing code and focusing solely on testing-induced validation, our method functions without issues across different repositories irrespective of specific repository configurations or similarity metrics not necessarily corresponding with functional correctness.

CHAPTER 3

PROPOSED SYSTEM ARCHITECTURE

3.1 Problem Definition

Large Language Models (LLMs) have shown impressive results in generating executable code from natural language prompts [1; 2; 4]. However, these models often struggle when deployed in realistic software development scenarios, especially where correctness depends on subtle functional requirements, edge cases, or adherence to project-specific conventions [7; 10]. In such contexts, generating code that merely appears syntactically correct is insufficient. Without precise understanding of intended behaviour, the generated functions frequently fail validation when executed against real-world test suites.

This work addresses the challenge of guiding LLMs to generate functionally correct Python code using only a limited set of natural language artifacts. Specifically, the model is exposed exclusively to high-level documentation (such as `.md` and `.rst` files), function docstrings, and unit test results—never to developer-written implementations. This strict separation avoids injecting solution-specific patterns or leakage of intended logic, and better replicates the experience of a human developer working from external documentation and test-driven feedback.

Unlike approaches that operate on isolated function definitions or synthetic benchmarks—such as HumanEval [2] and APPS [15]—this study targets entire open-source repositories. These repositories present a more realistic and complex environment: functions are deeply embedded within architectural layers, interact with shared state, and often depend on behaviour defined across multiple files. This design choice significantly increases the difficulty of the generation task, but also enhances ecological validity by aligning with how code is developed and maintained in practice.

To improve the quality of generated functions under these constraints, this work proposes an iterative refinement framework. The model generates an initial candidate function based on documentation and class context. If the generated function fails one or more tests, the system extracts a structured error summary and integrates it into a revised prompt. This feedback loop continues until all associated tests pass or a fixed number of iterations is reached. This process reflects the natural workflow of debugging and test-driven development, where errors guide progressive refinements until correctness is achieved [13; 2].

A single-pass version of the system is also implemented, where the model generates each function only once. This non-iterative configuration is used exclusively for comparative purposes, enabling a clear assessment of how test-driven feedback contributes to functional correctness. Results presented in later chapters demonstrate that the iterative refinement strategy consistently increases the success rate across a variety of repositories, without requiring model retraining or fine-tuning.

By operating solely on documentation, test outcomes, and human-readable descriptions, and by explicitly avoiding the use of existing implementation code, the proposed system sets a realistic and generalizable foundation for automatic code synthesis. It combines natural language understanding with empirical validation through testing, enabling a principled investigation of LLM capabilities in software development tasks grounded in real-world constraints.

3.2 Prior Attempt: Fine-Tuning and Its Limitations

Before arriving at the current design, an initial strategy was explored based on fine-tuning a language model to specialize in Python code generation. The hypothesis was that exposing the model to curated pairs of prompts—consisting of function signatures, documentation, and class context—together with their corresponding implementations would allow it to internalize patterns specific to the task. A fine-tuned model, it was hoped, would then generalize more effectively to new, similarly structured codebases.

The experimental setup involved training on one repository and testing on another, with the goal of assessing cross-project generalization. While the model demonstrated acceptable performance when generating functions structurally similar to those in the training repository, its ability to handle unfamiliar coding styles and documentation formats was limited. A second experiment attempted to mitigate this by training on a collection of multiple repositories and testing on a completely unseen one. However, this broader exposure did not improve performance. In fact, accuracy decreased further when the test repository diverged significantly in structure or style from the training set.

These results indicate that fine-tuning, while potentially useful in narrow or domain-specific contexts, struggles to scale in open-ended, heterogeneous environments such as those considered in this work. The diversity of documentation styles, naming conventions, and structural patterns across projects introduces too much variance for a static, pretrained model to handle reliably. Consequently, the fine-tuning approach was abandoned in favor of a more adaptive strategy: an iterative framework that leverages local feedback from test outcomes to guide generation without altering the model's internal parameters.

3.3 General Overview

This work introduces a modular and test-driven framework designed to evaluate and enhance the effectiveness of large language models in generating functionally correct Python code in realistic development settings. A key design constraint is the exclusive reliance on natural language artifacts—such as documentation files, docstrings, and test cases—without access to developer-written implementations. This separation ensures that generated functions are grounded in the model's interpretive capabilities rather than derived from memorized patterns or code replication.

To address the limitations discussed in Section 2.2—such as the difficulty of generating code that integrates into structurally complex repositories, the presence of cross-file dependencies, and the need for strict functional correctness—this work adopts a test-driven, iterative approach to code generation. Rather than relying on single-shot generation or leveraging existing implementation code as context, the proposed method is grounded in previous research on documentation-based synthesis [2; 9], while deliberately avoiding any exposure to developer-written functions. The model receives as input only natural language artifacts—such as `README` files, docstrings, reStructuredText documents, and test descriptions—ensuring that generation is guided purely by specification and structural analysis.

At the core of the approach is an iterative refinement loop in which each function is generated, validated through test execution, and then revised based on structured error feedback. This design mimics real-world development workflows where developers iteratively improve their code based on test results. Unlike methods such as RepoCoder [16] or CatCoder [17], which incorporate partial code retrieval or static typing constraints, this framework aims to simulate the end-to-end behavior of a developer tasked with writing new functionality from scratch based solely on textual descriptions. As discussed in Section 2.4, this separation from code reuse enables more robust generalization across diverse repositories and reduces the risk of overfitting to repository-specific patterns.

By integrating prompt engineering with a structured test-based feedback mechanism, this methodology supports the incremental improvement of generated code without requiring model fine-tuning or reliance on code similarity metrics. The result is a flexible, repository-agnostic system capable of producing implementations that are not only syntactically valid, but functionally sound within their respective software contexts.

At the core of this framework is a cycle in which the model repeatedly generates and refines a candidate function. The process begins with the construction of an initial prompt that encodes relevant context, including the function's name, signature, docstring, and surrounding class structure. The generated implementation is dynamically inserted into the codebase and tested using the function's associated test suite. If test failures occur, they are programmatically extracted and summarized, and this diagnostic information is incorporated into a revised prompt for the next iteration. This loop continues until the function passes all tests or a predefined iteration limit is reached.

The system architecture, shown in Figure 1, is designed to scale across repositories with varying structures and documentation formats. It parses project artifacts to extract metadata and establish precise function-to-test mappings. These mappings enable accurate prompt generation and ensure that test results are correctly attributed to the relevant function. Each iteration of the generation cycle is independently validated through function-specific test execution, ensuring that progress toward correctness is measured concretely and automatically.

By replacing static fine-tuning with dynamic prompt refinement, the framework remains agnostic to domain-specific implementation patterns and adapts to the local context of each repository. It mirrors common software engineering practices such as debugging, test-driven development, and regression analysis. Consequently, it offers both a robust benchmark for evaluating LLM-based code synthesis and a practical method for integrating such models into automated development pipelines.

3.4 Repository Acquisition and Analysis

The procedure begins with the retrieval of a repository from GitHub, which is a resource for pulling in core elements, consisting of Python code, functional requirements documentation, and unit tests to ensure correctness. An initial validation phase checks integrity of the codebase before adding in any alterations. In this phase, it infers function signatures, structural dependencies, exact mappings of functions to their respective test cases. Yet, to maintain the

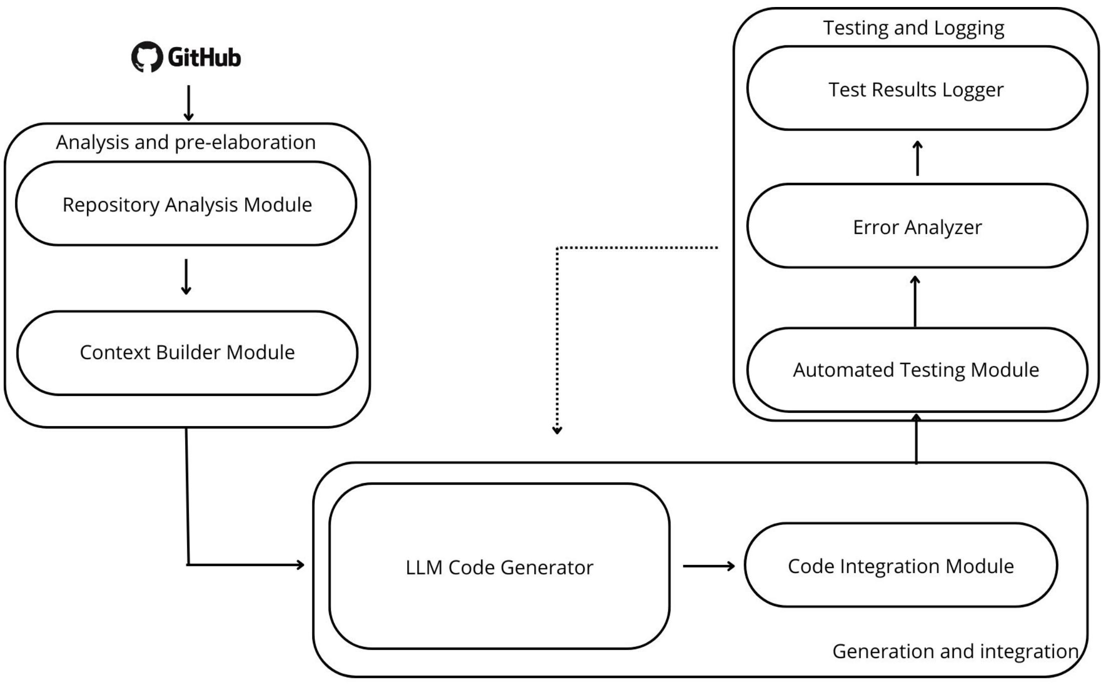

Figure 1: High-level system architecture.

challenge of code generation solely from natural language, the LLM is deliberately not granted direct access to existing implementations in the repository.

The methodology of repository acquiring is regulated by the `clone_or_update_repository` function, which guarantees that access to the latest version of the repository is ensured. In case the repository is not locally available, it is put under cloning; otherwise, if locally available, it is updated. The phase of evaluation is done through the `analyze_repository` function, which either loads a pre-computed summary or computes one in light of repository content. The process is important in achieving structural data and producing important prompts for the

LLM, while following the requirement that no code written by developers is integrated into the generation.

3.5 Context Extraction for the LLM

Large language model (LLM) context extraction is a foundational step for achieving context-aware and accurate automated code generation. In our approach, the pipeline systematically explores the repository's directory structure and extracts textual content from Markdown (.md) and reStructuredText (.rst) files, which typically contain the main body of human-written documentation. These documents are aggregated into a unified textual corpus, with filenames preserved for traceability and easier reference.

This corpus is then provided to a pre-trained language model via a carefully designed prompt that instructs the model to generate a structured summary of the system. The goal of this summary is to capture key technical aspects — such as system architecture, functionalities, APIs, data structures, and performance constraints — that are relevant for guiding code generation tasks.

By avoiding direct access to implementation files, this process ensures that the LLM generates code based solely on high-level documentation and specifications, thus adhering to the design constraint of abstraction from developer-written code.

3.6 Iterative Refinement with Error Feedback

The iterative refinement method enhances accuracy by integrating structured feedback loops throughout the code generation process. Initially, the LLM generates a function based on a carefully constructed prompt, which is then integrated into the codebase via the Replacer Module. Once incorporated, the function undergoes execution against a predefined suite of unit tests to assess its correctness. In the event of test failures, the system analyzes the errors through the Error Analyzer and Summarizer, which extracts relevant insights and refines the prompt accordingly. The LLM then utilizes this revised prompt to generate an improved version of the function. This cycle of evaluation and refinement continues iteratively until the function successfully passes all tests or reaches a predefined retry limit. Throughout this process, all results are systematically logged, ensuring transparency and enabling further analysis. By leveraging test feedback as a guiding mechanism, this approach significantly improves the likelihood of generating functionally correct implementations while minimizing the need for human intervention. The architecture for this approach is shown in Figure 2.

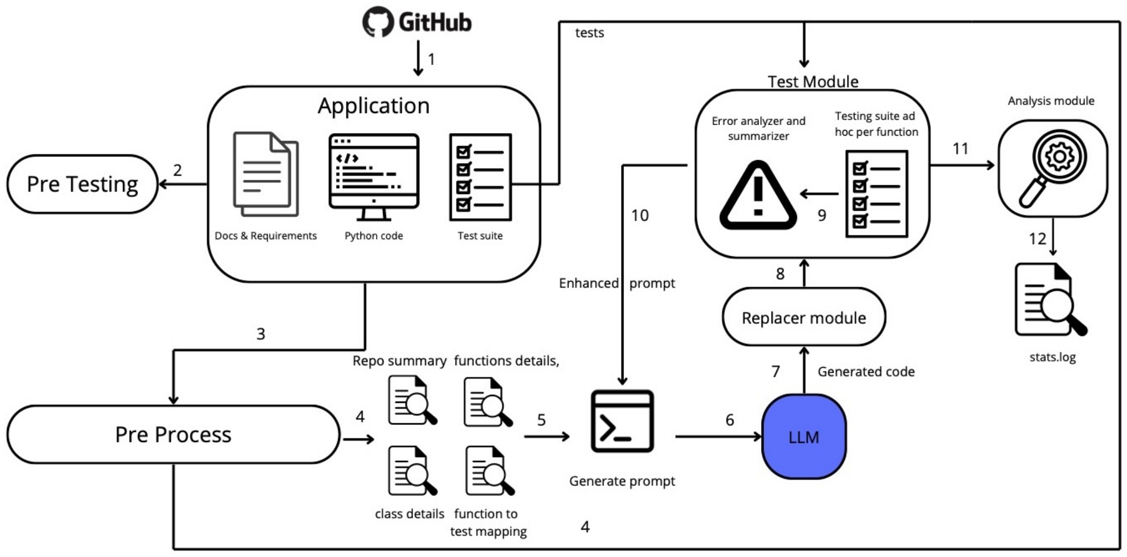

Figure 2: System architecture for iterative refinement approach.

3.7 Accuracy Validation through Testing

Evaluation via testing is a basic part of the assessment framework, as it ensures that functions developed meet set performance and accuracy levels. Validation involves use of tests that have been carefully crafted for functions derived from a suite repository, documenting instances of success and failure for empirical examination, and analyzing test execution performance using time complexity and error classification. Each function is tested exclusively with the set of pre-mapped tests that directly or indirectly depend on it, ensuring that the evaluation remains relevant to its specific role within the repository.

As illustrated in Figure 2, the iterative refinement process systematically incorporates error analysis to enhance the accuracy of the functions produced. Throughout this process, the LLM is deliberately prevented from accessing developer-written implementation code. This design choice reflects a realistic scenario where a function must be generated from documentation and test outcomes alone, without relying on call-site clues or existing function bodies, which could inadvertently guide the model. By isolating the generation process from prior implementations, we ensure that the resulting code is produced independently, thereby preserving the validity of the evaluation and its applicability to the creation of new functions in real-world settings.

This architectural framework enables fair assessment of both single-pass and iterative code generation approaches and provides an empirical measure for evaluating their effectiveness in real-world software development environments.

3.8 Error Analysis

Ensuring the correct identification of faults and extensive examination of faults is critical to improving functionality obtained from language models. The system utilizes two different methodologies for assessing functions under test for faults.

Regex-Based Error Extraction

Regex-based analysis involves analyzing test output using pattern-based methods. This technique efficiently locates faults in assertions, groups equivalent faults, and removes unnecessary information. By focusing on structured error messages, it provides a compact overview of faults without requiring additional computational resources. This technique is most useful for handling well-structured test outputs in which faults have predetermined patterns.

LLM-Driven Error Summarization

Upon activation, raw test outputs are processed by the LLM in order to generate a structured fault summary. This process differs from regex-based inspection in that it utilizes the model's ability to parse sophisticated test failure and eliminate extraneous information, thus providing a compact fault representation. The LLM is instructed to extract only the most relevant failure messages and thus exclude stack traces and unnecessary information.

3.9 Evaluation Metrics

In contrast to many prior works that evaluate code generation through similarity-based metrics such as BLEU scores or edit distance, this study adopts a functional, execution-based perspective. The primary objective is not to replicate the original implementation verbatim, but to generate functionally correct and executable code that integrates properly within a real-world software repository. Accordingly, the evaluation framework is centered on metrics that capture actual runtime behavior and development-oriented performance, rather than syntactic resemblance. [2; 15]

The most critical metric employed is the *success rate*, defined as the percentage of generated functions that pass all their associated test cases. This directly reflects the functional viability of the generated code within its operational environment. Unlike similarity metrics, which may score highly even when the code is syntactically correct but semantically flawed, test-based evaluation ensures that the model's output meets the intended behavior as defined by the test suite. This is particularly relevant in practical scenarios where correctness, rather than resemblance to a reference, is the ultimate requirement.

To complement success rate, the framework includes analysis of *effective lines of code* (LOC), comparing the size of the generated implementation to that of the original. This provides insight into the verbosity and structural efficiency of the generated code. While brevity is not always synonymous with quality, excessive or unnecessarily complex implementations may indicate misunderstanding of the task or overfitting to prompt artifacts.

Another critical dimension is *execution time*, which captures the duration needed to complete the generation, integration, and validation of each function. In iterative scenarios, this includes all regeneration cycles prompted by test failures. This metric offers an estimate of the system's responsiveness and suitability for real-world development pipelines, where developers expect prompt feedback and efficient iteration cycles.

The evaluation framework is composed of both *quantitative* and *qualitative* dimensions. Success rate and execution time are inherently quantitative, offering measurable and comparable results across experiments. LOC, while numeric, also has qualitative implications—it helps assess the readability, maintainability, and structural adequacy of generated functions. Together, these dimensions form a robust and pragmatic evaluation strategy that supports empirical comparison between approaches while remaining grounded in practical software engineering concerns.

Most importantly, this methodology ensures that models are evaluated based on their ability to produce code that is not only syntactically valid, but also executable, testable, and maintainable—closely aligning the evaluation process with the demands of real-world programming tasks.

3.10 Structured Prompting for Code Generation

A central feature of the system is in the formalized prompt design, which guarantees that the model generates functionally correct and semantically consistent code without explicit exposure to programmers' implementations. Since the model receives only textual explanations, documentation, and test cases, prompt design has to include all contextual information required to successfully direct the generation process. Each prompt is built dynamically from data acquired via parsing of the repository. This involves function names, parameter specifications, and documentation or inline comment-derived descriptions (docstring). When a function is inside a class, contextual information such as class attributes and methods is included to provide logical consistency within the codebase. This structured method ensures that the generated function complies with conventions in place within the repository and interacts with surrounding components as required.

The iterative improvement process refines the prompting methodology through structured feedback obtained from test failure instances. If a function does not validate, then the error analysis module captures relevant information about the failure and recasts it in a better prompting form. This process of improvement provides correct feedback to the large language model so that it can fix previously detected errors and avoid repeating them.

The prompting process is carried out via specialized modules that enable the generation of prompts throughout different levels. Prompting is utilized in the generation stage to maximize the chance for a correct function on the initial attempt. Where tests fail, structured error feedback is added to the prompt to enable a specific revision in implementation.

The structured prompting technique is crucial to the system's ability to improve code quality incrementally, within its main constraint: all implementations have to be derived solely from textual explanations and validation feedback. This ensures that generated functions are written independently of prior code.

CHAPTER 4

IMPLEMENTATION

4.1 Repository Selection and Management

Criteria for Repository Selection

The selection of repositories plays a critical role in shaping the scope, generality, and realism of the evaluation framework. For this study, repositories were selected based on their capacity to support meaningful testing of automatically generated code in a real-world development context. Specifically, repositories were prioritized if they met several key conditions: they had to be implemented in Python, include a sufficiently rich set of test cases, and offer meaningful natural language documentation in formats such as Markdown (.md) or reStructuredText (.rst). The presence of informative docstrings and clearly documented interfaces further reinforced their eligibility.

Importantly, the chosen repositories span a broad spectrum of domains and sizes. Some are lightweight libraries offering utility functionality, while others are larger-scale projects with more complex structures and dependencies. This heterogeneity was intentional, as it allows the system to be evaluated across a variety of software development scenarios, ranging from cryptographic tools to debugging utilities, from scheduling libraries to database engines. Such diversity is essential to assess the generalizability and robustness of the code generation approach.

Each repository is treated as a self-contained software artifact. The system must analyze its documentation, extract function-level metadata, and generate or refine code in such a way that the resulting implementations integrate seamlessly within their local context. This diversity in size, design, and functionality ensures that the evaluation is not confined to contrived or overly simplified scenarios, but instead reflects the variability encountered in practical software engineering.

TABLE I: OVERVIEW OF SELECTED REPOSITORIES AND ASSOCIATED METADATA.

Index	**Owner**	**Repository**	**Description**	**# Functions**
1	msiemens	tinydb	A lightweight document-oriented database designed for small-scale applications. It provides a flexible query system and is often used in scenarios where traditional SQL-based solutions are excessive.	46
2	cdgriffith	Box	A Python library that extends dictionary objects to allow attribute-style access. It simplifies manipulation of nested data and is often used in configuration management and data serialization.	18
3	pyeve	cerberus	A rule-based validation engine for Python dictionaries. It enables precise validation of user input or configuration files and is frequently used in REST API backends.	44

Index	Owner	Repository	Description	# Functions
4	cool-RR	PySnooper	A lightweight debugging utility that automatically logs variable changes and function calls to aid in understanding and troubleshooting code execution.	13
5	mpdavis	python-jose	A cryptographic library for handling JSON Web Tokens (JWTs), signature algorithms, and key management in Python applications, often used in authentication systems.	54
6	delgan	loguru	A modern logging library that simplifies logging configuration while offering advanced features such as sink management, colorized output, and contextual information injection.	24
7	arrow-py	arrow	A replacement for the standard Python datetime module, providing a more intuitive interface for date and time manipulation. It is frequently used in scheduling and temporal data processing.	92
8	jquast	blessed	A library for terminal interaction that provides advanced formatting, keyboard input handling, and color management, useful for building command-line interfaces.	81

Index	Owner	Repository	Description	# Functions
9	mahmoud	boltons	A large collection of standalone utility functions and classes covering areas such as iteration, time management, string manipulation, and networking. It serves as a general-purpose standard library extension.	237
10	stochastic-technologies	shortuuid	A utility for generating concise, unambiguous, and URL-safe UUIDs. It is typically used in web applications where identifier brevity and clarity are desirable.	9

Selecting appropriate repositories for evaluation involved several trade-offs and practical considerations. One of the foremost challenges was achieving a balance between diversity and manageability. Repositories varied widely in terms of codebase size, complexity, and architectural design. While smaller projects provided a controlled environment for debugging and quick iteration, larger ones introduced real-world intricacies such as inter-module dependencies, abstract base classes, or indirect test coverage. Ensuring that the system could adapt to both ends of this spectrum required the development of flexible parsing strategies and modular processing pipelines.

In some cases, test execution also depended on specific environmental constraints, such as the presence of native libraries, third-party services, or operating system features. Repositories that posed insurmountable setup or compatibility issues were excluded from final experimentation, though special attention was given to ensure the remaining selection retained broad functional diversity.

GitHub Integration and Repository Management

To maintain consistency and reproducibility throughout the experimental pipeline, repositories are integrated directly from GitHub using an automated acquisition strategy. Upon initial execution, each repository is cloned locally from its public remote. If the repository is already available on disk, the system attempts to update it using a standard pull operation, thereby ensuring that the most recent version of the source code and associated test suite is used.

This approach avoids unnecessary duplication while also ensuring that any recent changes, such as added test cases or updated documentation, are captured. It also facilitates large-scale experimentation across multiple repositories without manual intervention, a necessary feature for iterative evaluation loops. All repository data is stored in a standardized directory structure that allows downstream modules to access code, documentation, and tests efficiently.

At present, all evaluated repositories are publicly accessible and do not require authentication. However, the infrastructure can easily accommodate authenticated requests if private repositories are to be included in future expansions. This would involve integrating GitHub API tokens to handle rate limiting and secure access control.

By combining a diverse and well-structured set of repositories with automated integration and update mechanisms, the system is equipped to support rigorous, repeatable, and generalizable evaluation of LLM-based code generation capabilities.

4.2 Code Extraction: Functions, Classes, and Tests

AST Parsing and Code Analysis

To extract relevant information from Python repositories, the system employs the Abstract Syntax Tree (AST) module, a built-in Python library that enables static code analysis. This approach ensures a structured and reliable method for identifying function definitions, class definitions, and their associated metadata.

The extraction process begins by parsing each Python file into an AST representation, allowing the system to traverse the syntax tree and detect class and function nodes. Each class definition is analyzed to extract its name, docstring, and attributes. Instance attributes are collected by inspecting assignments to the `self` object within the `__init__` method, while class attributes are identified as variables assigned directly in the class body. This information is crucial in constructing a clear class structure that can later be leveraged for generating coherent and context-aware function implementations.

Function extraction follows a similar methodology. Each function is analyzed to retrieve its name, arguments, and docstring. Additionally, the system captures the precise function signature, ensuring that decorators and return type annotations are preserved. This is particularly valuable for maintaining function integrity and ensuring that generated code matches expected specifications. The ability to extract multiline function signatures allows the system to generate highly informative prompts when invoking the language model for code generation.

To improve robustness, the system is designed to ignore irrelevant files, such as those located in example directories or those that contain syntax errors. By systematically filtering out nonrelevant or malformed files, the system minimizes noise and enhances the accuracy of the extraction process.

The information collected is stored in an internal representation that reconstructs the repository's structural hierarchy. This code tree enables the system to efficiently navigate the codebase, establishing links between classes, functions, and eventually their test coverage. The entire process is governed by a modular pipeline, summarized in the following high-level pseudocode:

Algorithm 1 Class and Function Extraction via AST

```
Input: Repository path $R$
for each file $f$ in $R$ do
    if $f$ is not a valid Python file or contains syntax errors then
        continue
    end if
    Parse $f$ into AST tree $T$
    for each node $n$ in $T$ do
        if $n$ is a class then
            Extract class name, docstring, attributes
        else if $n$ is a function (global or method) then
            Extract function name, arguments, docstring, signature
        end if
    end for
end for
return Structured list of functions and classes with metadata
```

This structured parsing and filtering process ensures that only meaningful and analyzable code is passed to downstream modules. The extracted metadata serves as a foundation for generating consistent and context-aware prompts, aligning the code generation process with the original implementation logic of the repository.

Function-Test Mapping

After identifying all functions and classes within the repository, the system proceeds to establish a precise correspondence between these elements and the test cases that validate them. This step is critical to ensure that any generated function can be reliably evaluated by leveraging the repository's existing test suite.

The process begins by scanning all files recognized as test scripts. These are typically located in conventional directories such as `tests/`, or named according to common testing patterns like `test_*.py`. Within these files, the system identifies test functions based on naming conventions (e.g., functions starting with `test_`) and parses their source code using Abstract Syntax Tree (AST) analysis. Through this parsing, the system extracts the names of application-level functions invoked within each test. This forms the foundation for the initial test-to-function mapping.

However, this direct association often captures only part of the picture. Many real-world test functions verify behaviors that rely on chains of internal function calls, not just the ones explicitly invoked within the test itself. To address this, the system extends its analysis through test propagation, a process that recursively expands test coverage by analyzing the function call graph.

In particular, if a test case invokes a function A, and A in turn calls another function B, then the test case is considered to cover both A and B. This transitive reasoning is generalized across the entire call graph: any function f_i that calls f_j transfers its test coverage to f_j, recursively, until the propagation stabilizes. This approach ensures that even deeply nested helper functions—those not directly invoked by test code—are still associated with the appropriate test cases, provided they are part of a tested execution path.

Algorithm 2 Function-to-Test Mapping with Propagation

```
1: Input: Repository path $R$, function set $\mathcal{F}$
2: Extract test functions and build call graph $G$ over $\mathcal{F}$
3: Initialize mapping function_to_tests = {}
4: for each test function $t$ do
5:     Associate $t$ to directly called functions
6: end for
7: repeat
8:     changed ← False
9:     for each function $f_i$ in $G$ do
10:        for each callee $f_j$ of $f_i$ do
11:            Propagate tests from $f_i$ to $f_j$
12:            if new tests added then
13:                changed ← True
14:            end if
15:        end for
16:    end for
17: until changed = False
18: return function_to_tests
```

This recursive propagation is implemented efficiently, ensuring convergence by halting once no new associations are added in a full pass over the graph. The final result is a comprehensive map from functions to the set of test cases that either directly or indirectly exercise them.

The resulting function-test mapping is stored in a structured format and includes metadata such as the file and line number where each function is defined. This output not only provides transparency into the system's internal mapping logic but also serves as a valuable debugging tool. For instance, it allows users to easily verify which functions are well-covered and which might lack sufficient testing support.

In practice, this enriched mapping significantly improves the robustness of the code generation pipeline. It ensures that the evaluation of generated functions reflects the full behavioral coverage offered by the test suite, capturing edge cases and helper logic that might otherwise be overlooked. Moreover, by relying on a structural analysis of the call graph rather than superficial string matching, the system achieves high accuracy in associating tests to the actual functionality under examination.

In summary, the function-to-test mapping with propagation plays a central role in validating the correctness of generated code. It allows the system to take advantage of the test suite's implicit structure and logic, ensuring that each function is evaluated not only based on explicit references in test files, but also through the complete chain of functional dependencies involved in tested workflows.

4.3 Contextual Summarization and Documentation Extraction

A foundational component of the proposed system is the generation of a high-level summary that captures the structure, purpose, and internal mechanics of the target repository. This summary is derived by aggregating and analyzing all textual documentation within the repository, primarily sourced from Markdown (`.md`) and reStructuredText (`.rst`) files. The extraction process systematically traverses the directory tree to locate these documents and compiles them into a unified textual corpus.

This corpus is subsequently passed to a dedicated summarization module, which prompts a large language model to generate a detailed and structured overview of the repository. The prompt is designed to elicit specific technical details relevant to code generation, including the system's architectural layout, functional modules, data flow patterns, API specifications, and domain-specific algorithms. The result is a structured summary that serves as the foundational context for subsequent code synthesis tasks.

The importance of this process is twofold. First, it allows the model to build a mental representation of the repository's logical design and software purpose without accessing any existing implementation code. Second, it enhances the semantic coherence of generated functions by exposing the model to prior function names, expected behaviors, and component responsibilities described within the documentation. In many cases, the prompt includes the names of core functions and classes, which enables the LLM to invoke them appropriately during generation. This capability is especially critical when the generated function is meant to interact with existing components, such as performing operations on user-defined objects or invoking internal utility methods.

Furthermore, the quality of this initial context directly influences the success of the generation process. A well-summarized system facilitates more accurate and structurally aligned function generation, reducing the likelihood of semantic drift or redundant code. By contrast, poor or incomplete documentation leads to vague prompts, which may produce syntactically valid but semantically inappropriate implementations.

Therefore, the contextual summarization phase is not merely a preparatory step but a critical enabler of functionality-aware code generation. It ensures that all downstream modules operate with a shared and accurate understanding of the software system, laying the groundwork for functionally correct, maintainable, and contextually integrated code outputs.

4.4 LLM API Integration

Provider Selection and Integration Strategy

The integration of Large Language Models (LLMs) is a central component of the proposed system, as it enables both the generation and iterative refinement of Python functions based solely on natural language documentation and structured prompt inputs. In order to support high-volume, iterative experimentation while balancing factors such as latency, cost per token, API stability, and rate limits, several LLM providers were evaluated during the design phase.

Ultimately, the platform selected for integration was OpenRouter, which offers a unified API interface for accessing multiple models. This choice facilitated seamless experimentation without the need to modify the underlying codebase for each model. Among the models available through OpenRouter, two were chosen for detailed analysis. Gemini 2.0 was selected as a general-purpose model due to its competent performance across a variety of tasks and its favorable pricing structure, while QwenCoder 2.5 Instruct was chosen for its specific optimization toward code generation. By employing these two models, the system is able to compare a generic LLM (Gemini 2.0) with a code-centric model (QwenCoder 2.5 Instruct), thereby providing valuable insights into their relative performance in generating and refining code based solely on natural language prompts.

Notably, models with relatively small parameter counts and limited context windows (typically below 7 billion parameters) were excluded from the final round of testing. These models struggled to process long, structured prompts containing detailed class hierarchies and documentation, often truncating or ignoring critical input details and leading to incomplete or incoherent outputs. Consequently, the focus was maintained on models that could effectively handle the complexity inherent in real-world software development scenarios.

In summary, by leveraging OpenRouter to access both Gemini 2.0 and QwenCoder 2.5 Instruct, the system benefits from a comparative framework in which one model provides a general-purpose approach while the other is specifically tailored for code generation. The subsequent analysis of their results enables a deeper understanding of the trade-offs between generality and specialization in LLM-based code synthesis.

Prompt Generation and Structuring

A defining feature of this system is its modular and hierarchical approach to prompt construction. All prompts are generated dynamically through specialized functions that encode contextual information about the target function, its class (if applicable), and the repository in which it resides. This approach ensures that the model operates exclusively on textual inputs derived from documentation and structural analysis, in full compliance with the constraint of avoiding any exposure to developer-written implementations.

The core prompting functions are `generate_prompt_for_function` for the first prompt, and `generate_improved_prompt` for the subsequent prompts. These functions accept as input the function metadata, a repository-level summary, and, when applicable, class-level details such as attributes and related methods.

The resulting prompt includes a complete function signature, a detailed docstring, and a textual description of the function's expected behavior. When the target function belongs to a class, the prompt also includes the class name, its attributes (both class-level and instance-level), and the names and signatures of other methods defined within the same class. This provides the LLM with the contextual information necessary to generate a function that integrates coherently within its surrounding code structure.

The prompt construction process enforces strict formatting and output constraints. The model is explicitly instructed to return only the function definition and its body, with no accompanying commentary, markdown formatting, or usage examples. These instructions are critical for ensuring that the generated code can be parsed, tested, and integrated into the codebase without additional post-processing.

Iterative Refinement with Error Feedback

When a generated function fails validation, the system engages in an iterative refinement process aimed at incrementally improving the implementation. This is achieved by invoking the `generate_improved_prompt` function, which constructs a new prompt incorporating information about the previous generation attempt and the associated test failures.

The improved prompt retains the function signature and docstring but appends two new sections: the exact content of the previously generated code, and a structured summary of the errors encountered during testing. This feedback may originate from regex-based parsing of the test output or from a separate LLM module dedicated to error summarization. The prompt then includes specific instructions to revise the function, addressing the identified errors while preserving the intended behavior and structural integrity of the original specification.

To illustrate, a typical refinement prompt might include the function's prior definition:

```
def parse_data(file_path):
    ...
```

followed by an error summary:

```
TypeError: expected string or bytes-like object, got 'NoneType'
```

The model is then instructed to generate a revised implementation that is not identical to the previous attempt and that explicitly addresses the listed error. By repeating this cycle—prompt generation, function synthesis, test execution, and feedback integration—the system guides the model toward a correct and stable implementation.

Each iteration is logged for future analysis, and the loop terminates either when all tests pass or when a predefined retry threshold is reached. This mechanism ensures that function generation is not a one-shot operation, but rather an adaptive process that mimics real-world development workflows, where errors are systematically diagnosed and resolved.

4.5 Multi-Function vs. Single-Function Approaches

The proposed system is designed to operate across entire repositories or on individual functions, using an iterative refinement strategy guided by unit test feedback. This flexibility enables both broad evaluations of model performance and targeted debugging of specific functions. Central to this approach is the integration of automatic feedback loops that allow the model to iteratively revise its output in response to empirical errors.

Iterative Refinement Across Repository

In this method, the system begins by generating a candidate implementation for each function using a prompt that includes its docstring, parameters, and surrounding class context. This initial version is inserted into the codebase and tested using the function's associated test suite. If the implementation fails any test cases, the system activates a refinement cycle that incorporates error feedback into a new prompt. This process repeats until the function passes all tests or a predefined iteration limit is reached.

This iterative strategy mimics real-world debugging workflows, where repeated testing and correction cycles are essential for achieving robust functionality. Each failed attempt provides actionable insights that guide subsequent corrections, transforming test feedback into a generative signal. The refinement loop is supported by an Error Analyzer and Summarizer module, which distills raw error outputs into concise descriptions that can be interpreted by the model.

As demonstrated in Chapter 5, this approach yields significant improvements in function correctness across diverse codebases. The system adapts to varying documentation styles and

Algorithm 3 Iterative Refinement Across Repository

```
1: Input: list of functions with docstrings and test coverage, maximum iteration limit $N$
2: for each function $f$ in the repository do
3:     Prompt the LLM with $f$'s docstring and context to generate an initial solution
4:     for $i = 1$ to $N$ do
5:         Inject the generated code into the repository
6:         Run all relevant tests for $f$
7:         if tests pass then
8:             Record success and stop refining $f$
9:             break out of the loop
10:        else
11:            Extract error details from the failing tests
12:            Append error summaries to the prompt for the next iteration
13:            Query the LLM again using the refined prompt
14:        end if
15:    end for
16:    if none of the $N$ attempts succeed then
17:        Record failure for $f$
18:    end if
19: end for
20: Summarize results for all functions
```

logic complexity without requiring changes to the model's internal parameters. By relying solely on prompt-level modifications and error-driven iteration, the method achieves high reusability and generalization in open-ended software environments.

This flexible generation pipeline can operate at scale, processing entire repositories, or can be deployed in a focused manner to improve specific functions. Regardless of scope, the underlying principle remains the same: use structured testing feedback to guide the model toward solutions that satisfy predefined correctness conditions in a test-driven and documentation-aware manner.

4.6 Automated Code Replacement and Integration

A crucial phase in the system's workflow involves integrating the newly generated function directly into the existing codebase, by replacing the current implementation of the function under test. Unlike a stub-completion setting, the goal here is not to fill in a blank method or create code from scratch, but to evaluate whether the function generated by the language model can effectively replace a real, working function from an active repository. This is essential for assessing the quality of LLM-based code generation in real-world development environments.

This task is managed by the `function_replacer.py` module, which carries out a precise sequence of steps: extracting the original function and its indentation, reformatting the new code to match the repository's structure, replacing the original implementation in-place and executing relevant tests.

The replacement procedure begins with the creation of a backup of the original file. This ensures that even if an error occurs during parsing, editing, or testing, the source code remains intact. The system then uses AST traversal to locate the exact bounds of the original function, preserving decorators and indentation. The generated function is then parsed, cleaned, and re-indented to match the formatting of the surrounding code, using a normalization strategy based on the function's original indentation and structural depth.

Once the new implementation is inserted into the file, the system immediately executes a pre-selected subset of tests that are known to invoke the target function, either directly or indirectly. These test mappings are established earlier during the repository analysis. The outcome of the test execution determines whether the generated function is accepted or discarded. Crucially, this evaluation happens in a realistic setting: the new function is tested on site, within its full application context.

Regardless of the test outcome, the module concludes by restoring the original version of the file from the backup. This design ensures that the repository remains unmodified between runs, which is essential for isolating each test iteration and preventing cascading effects. The replaced function and the result of the test execution are logged in detail, including the generated code, success or failure status, and any relevant error summaries.

The internal logic responsible for replacing and validating generated code is governed by a structured routine. This procedure ensures that each newly produced function is integrated into the codebase, executed in a controlled environment, and evaluated solely through its associated tests. The original implementation is first backed up, then replaced with the generated version, and specific tests targeting the modified function are run. Depending on the outcome, the system records either a success or failure and restores the original code to ensure repository integrity for future runs.

This conservative and isolated strategy enables precise assessment of functional correctness without introducing unintended side effects. It reflects a realistic scenario in which a developer proposes a new implementation, tests it in place, and only accepts changes once correctness has been empirically demonstrated.

Algorithm 4 High-Level Procedure for Function Replacement and Validation

```
1: Input: target_function, new_implementation
2: Preserve the original source code for safety
3: Integrate the new implementation into the codebase
4: Identify and execute relevant tests for the modified function
5: if tests are successful then
6:     Record the implementation as valid
7:     return success
8: else
9:     Capture and log test failure information
10:    return failure
11: end if
12: Restore the original function to maintain repository integrity
```

4.7 Error Handling and Debugging

A key component of the system is its ability to interpret test failures and use them to guide successive code refinements. This error-handling capability is essential when evaluating the correctness of generated functions, especially in the iterative mode where test outcomes directly influence future generations. To achieve this, the system implements two complementary mechanisms for extracting and summarizing test errors: one based on regular expression parsing, and the other powered by a dedicated LLM. These approaches are designed to accommodate different levels of complexity in test outputs, and can be selected or combined depending on the nature of the repository and the user's configuration.

Regex-Based Error Extraction

The first strategy employs pattern matching to identify and extract useful information from the raw test output. As soon as a test fails, the system records its full textual trace. The function `extract_error_summary` then processes this log line by line, discarding unrelated content—such as platform metadata or general summaries—and isolating segments that indicate assertion failures or exceptions.

The method focuses primarily on lines that begin with the error marker `E`, as well as those that contain explicit `assert` statements. These lines are grouped into compact entries based on their main error messages, such as type mismatches, missing attributes, or value discrepancies. Redundant lines and repeated stack traces are collapsed to reduce noise, and line numbers are ignored to prevent superficial variations from inflating the count of unique issues. When the same error occurs across multiple files or test cases, the system merges them into a single, canonical entry, ensuring that the resulting summary remains focused and easy to interpret.

Regex-based extraction is efficient and lightweight. It does not rely on any external services and functions reliably in offline environments. However, its effectiveness depends on the consistency of the test output format and the specificity of the matching patterns. In repositories with unconventional test logging or less structured output, this method may struggle to produce clear summaries. For these cases, an alternative mechanism—based on LLM summarization—is also available.

LLM-Assisted Error Summarization

To handle more verbose or ambiguous error logs, the system offers an optional error summarization mode that leverages a Large Language Model. When this mode is enabled, the system invokes the function `extract_error_summary_with_LLM`, which constructs a custom prompt containing the raw test output. The model is asked to produce a concise and readable summary of the errors, omitting file names, line numbers, and stack trace details. The emphasis is placed on identifying the underlying causes of failure and organizing them in a way that facilitates correction.

This approach can provide valuable advantages, especially in large or heterogeneous repositories where test failures may span multiple layers of logic or involve nested exceptions. By summarizing conceptually similar issues—even if their wording differs slightly—LLM-based summaries offer a more semantic interpretation of the test outcomes. They are particularly helpful when multiple errors stem from a single underlying misunderstanding, as the model can consolidate feedback and highlight patterns that might otherwise go unnoticed.

To ensure robustness, the system includes a fallback mechanism. If the LLM request fails due to network issues, API errors, or empty responses, the system reverts to regex-based parsing automatically. This guarantees that error information is never lost and that the iterative process can proceed without interruption.

In both modes, error analysis is triggered only if a test suite fails and if the iterative refinement option is active. The summarized errors—whether generated by regex or the LLM—are then included in the prompt for the next generation attempt. This loop continues until the function passes all relevant tests or the retry limit is reached. By combining syntactic parsing with semantic analysis, the system maintains a flexible and resilient error-handling pipeline capable of supporting both straightforward and complex debugging scenarios.

4.8 Performance Considerations

Scalability and Large-Scale Processing

The framework is built to analyse repositories ranging from small utility libraries to large projects that expose hundreds of public callables. Although no hard ceiling is imposed on input size, growth in repository scale magnifies three practical pressures: total wall-clock time, aggregate compute consumption, and sensitivity to external language-model services.

When a repository expands, every function triggers a fixed pipeline—documentation parsing, prompt construction, model inference, code insertion, and test execution. Under iterative refinement this pipeline may repeat several times before a function satisfies all of its tests. In a codebase such as `mahmoud/boltons` (more than 230 functions), thousands of pipeline stages can accumulate in a single evaluation run. While an individual stage completes quickly, their collective cost scales super-linearly with both the number of functions and the average number of retries per function.

Test-suite heterogeneity adds further variability. Lightweight unit tests often finish in milliseconds, whereas file-system or network-bound integration cases can run orders of magnitude longer. Because the framework delegates execution to each repository's own tests without modification, it inherits this latency profile wholesale. A single slow test, if invoked repeatedly during refinement, can dominate the global timeline.

Dependence on remote LLM endpoints introduces an additional scaling factor. Every prompt consumes tokens and counts against provider quotas. During long experiments, transient throttling, network congestion, or short service interruptions can delay generations or force retries. These exogenous effects must be considered when budgeting time and resources for large experimental batches.

Several engineering measures keep the workload tractable. Parsed documentation, ASTs, and contextual summaries are cached so that reruns avoid redundant computation. Related functions are processed in batches to amortise interpreter start-up and to reuse test fixtures. Configurable retry ceilings prevent unbounded iterations on especially stubborn functions. Future versions may incorporate real-time quota telemetry, adaptive concurrency, and checkpoint recovery, but even in its present form the system can run unattended on extensive repositories while honouring its documentation-driven, test-guided philosophy.

Computational Overhead and Execution Time

Iterative refinement mirrors a developer's inner loop: generate a candidate implementation, run the tests, inspect failures, and revise until all tests pass or a stop rule fires. Each iteration introduces a fresh model call, a code rewrite, and another test cycle, so total runtime grows with both the number of functions and the average iterations per function.

Runtime also depends on external factors. Network latency can stretch model response times, while provider rate limits may throttle bursty request patterns. Variation within test suites—especially when integration tests are involved—widens the gap between best-case and worst-case completion times, complicating accurate scheduling.

Not every revision advances the function toward correctness; some changes fix one issue while revealing another. To reduce waste, each cycle ends with an automatic error-summarisation step that condenses failing traces into concise, actionable feedback. Focused feedback raises the likelihood that the next generation will address the underlying defect, shortening the expected path to convergence.

Although the loop increases computational expense, it turns unit tests into a live supervisory signal that steers generation toward functional soundness. Results in Chapter 5 show a clear trend: additional iterations significantly raise the success rate, particularly in security-critical or data-validation code where reliability cannot be compromised. Early-stopping thresholds, adaptive prompt budgets, and granular logs give practitioners fine-grained control over the trade-off between runtime and correctness.

In sum, the framework invests extra computation to achieve higher assurance. For scenarios where accuracy outweighs speed, this overhead is a rational cost: iterative refinement consistently delivers implementations that satisfy existing tests and align with real-world constraints, something one-shot generation rarely attains.

CHAPTER 5

EVALUATION AND COMPARISON

This chapter assesses the effectiveness of the iterative, test-driven framework introduced in the previous sections. A streamlined, *non-iterative* configuration—illustrated in Figure 3—is implemented solely to provide a reference point. In that variant the model receives the same prompt constructed from documentation and docstrings, generates a single candidate implementation, and is evaluated once against its unit tests. No regeneration takes place, and no additional feedback is supplied. All other components of the pipeline (repository analysis, prompt construction, code insertion, and test execution) remain identical, ensuring that any performance difference can be attributed to the presence or absence of the feedback loop.

The comparative analysis proceeds along three quantitative dimensions: *success rate*, defined as the proportion of functions that pass every associated test; *lines of code* (LOC), which captures verbosity and potential over-engineering; and *execution time*, measured from the first prompt emission to the final test outcome, including all retries where applicable. Section 4.7 complements these metrics with a qualitative study of the two error-handling modes—regex extraction and LLM-based summarisation—and their impact on convergence speed and accuracy.

Because the evaluation targets complete open-source repositories rather than isolated tasks, the generated functions must integrate with pre-existing modules, respect cross-file dependencies, and satisfy domain-specific constraints. This setting is markedly more demanding than

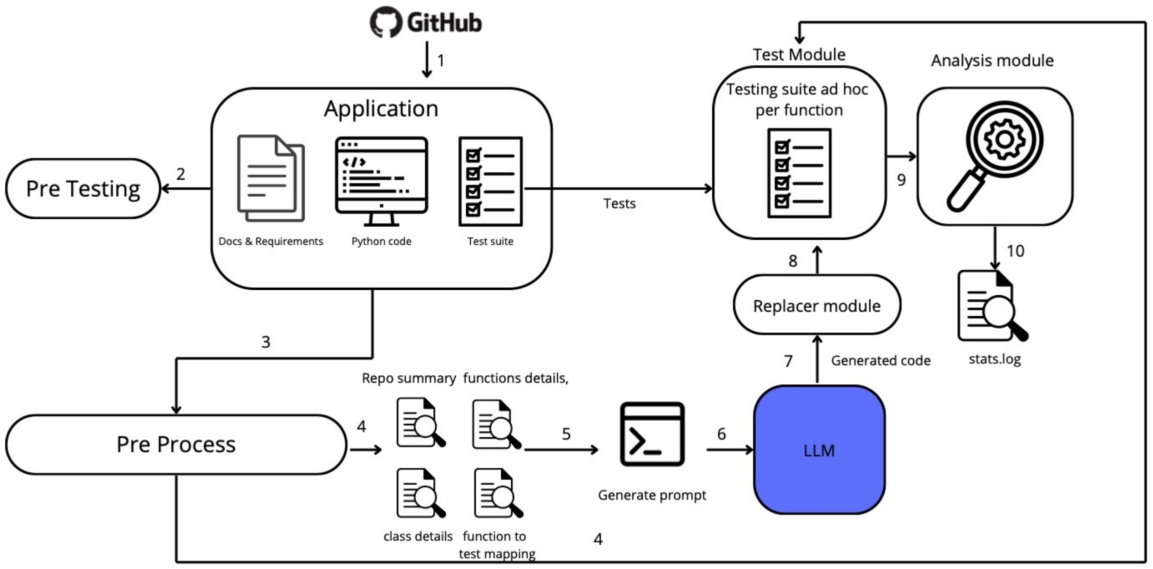

Figure 3: System architecture for single-pass code generation approach.

benchmarks based on stand-alone problems such as HumanEval or APPS, where each function is self-contained and receives an exhaustive specification in the prompt [2; 15].

The remainder of the chapter is organised as follows. Section 5.1 presents aggregate results across all ten repositories, contrasting the iterative framework with the single-prompt reference. Section 5.2 compares the behaviour of a general-purpose LLM (Gemini 2.0) and a code-specialised LLM (QwenCoder 2.5 Instruct). Section 5.3 investigates performance as a function of repository size and structural complexity, while Sections 5.4 and 5.5 discuss, respectively, the relationship between code length and correctness, and the computational overhead introduced by iterative refinement. The chapter concludes with a synthesis of key findings and an outlook on how test-guided iteration can be integrated into real-world development workflows.

Key Metrics Computation and Logging

In order to assess the performance of the proposed system, three primary metrics were computed and logged throughout the experiments. These metrics, which complement the evaluation framework discussed in Section 3.9, provide both quantitative and qualitative insights into the system's behavior under various generation strategies.

The *Success Rate (SR)* was determined by calculating the proportion of functions that, upon integration, passed all associated test cases. For each function generated, the system executed a predefined test suite—following the automated code replacement process described in Section 4.6. A function was marked as successful only if it passed every test case; the overall success rate was then computed as the ratio of successful functions to the total number of generated functions, expressed as a percentage. All test outcomes were logged systematically for subsequent analysis.

The *Lines of Code (LOC)* metric was used to assess the verbosity and structural efficiency of the generated solutions compared to the original implementations. Beyond serving as a measure of code compactness and clarity, LOC also provided insight into how the LLM handled functions of varying length, enabling an analysis of whether the model was more effective when generating shorter or longer implementations. The computed LOC values for each function were recorded in the experimental log files for both individual and aggregate analyses.

Finally, *Execution Time* captured the total time required for a function to reach a passing solution or for the system to exhaust the predefined retry limit. Timing measurements were taken from the moment a function generation attempt was initiated until the test suite produced a final outcome. In cases where iterative refinement was applied—incorporating multiple generation attempts as detailed in Section 4.7—the execution time metric included the cumulative duration of all iterations. High-resolution timers ensured precise measurement, and all timing data were logged alongside the corresponding function identifiers and iteration counts.

By systematically logging these metrics in a structured format, the evaluation framework provides a robust basis for comparing single-pass and iterative approaches and for drawing conclusions regarding their practical applicability in real-world software development workflows.

5.1 Aggregate Results: Single-Pass vs. Iterative Methods

Table II summarizes the overall performance of both single-pass and iterative approaches across the selected repositories. For each repository, the table compares the success rate of each model in single-pass mode against the the same model using iterative refinement, with one row per model. The results confirm that iterative strategies consistently outperform the single-pass method in terms of functional correctness.

In particular, repositories such as *Box*, *PySnooper*, and *shortuuid* show marked improvements. For example, the Gemini 2.0 model achieves a success rate of only 38.9% in single-pass mode for *Box*, which rises to 66.7% when using iterative refinement. Similarly, QwenCoder 2.5 improves from 69.2% to 92.3% in *PySnooper*, and from 25.0% to 77.8% in *shortuuid*. These patterns suggest that many failures in single-pass mode are recoverable through successive regeneration attempts informed by test feedback.

The benefit of iteration is also evident in larger repositories such as *boltons* and *python-jose*, where the initial success rate is relatively low (around 40% or less), but iterative execution lifts it by more than 10 percentage points for both models. While the absolute gain varies, the relative improvement remains consistent, underscoring the importance of test-driven retries for handling complex code structures and external dependencies.

It is important to note that the gains from iteration come at the cost of increased computational effort. As discussed in Section 5.5, iterative methods may require significantly longer runtimes due to multiple generation-validation cycles. However, given the substantial gains in correctness observed across nearly all repositories, the trade-off is often justifiable, especially in settings where reliability takes precedence over latency.

In sum, these results reinforce the value of integrating feedback-driven iteration into LLM-based code generation pipelines. By overcoming limitations of the single-pass approach, iterative refinement enhances the model's ability to produce functionally valid, test-passing code across diverse software repositories.

Comparison of Error Summaries

Within the iterative refinement process, the type of error feedback provided to the LLM plays a crucial role in guiding successful regeneration. This study investigates two complementary strategies for error summarization: a lightweight, rule-based approach using regex-based

TABLE II: SINGLE-PASS (ITERATION = 1.0) VS. BEST ITERATIVE (ITERATION = 10.0) SUCCESS RATES FOR EACH REPOSITORY, WITH ONE ROW FOR GEMINI 2.0 AND ONE FOR QWENCODER 2.5 INSTRUCT.

Repository	**Model**	**Single-pass SR (%)**	**Iterative (10.0) SR (%)**
Box	Gemini 2.0	38.9	66.7
Box	QwenCoder 2.5	38.9	61.1
PySnooper	Gemini 2.0	76.9	84.6
PySnooper	QwenCoder 2.5	69.2	92.3
arrow	Gemini 2.0	27.2	35.9
arrow	QwenCoder 2.5	27.2	34.8
blessed	Gemini 2.0	37.0	46.9
blessed	QwenCoder 2.5	37.0	46.9
boltons	Gemini 2.0	43.9	54.9
boltons	QwenCoder 2.5	42.2	59.1
cerberus	Gemini 2.0	29.6	47.7
cerberus	QwenCoder 2.5	27.3	36.4
loguru	Gemini 2.0	29.2	50.0
loguru	QwenCoder 2.5	33.3	50.0
python-jose	Gemini 2.0	20.4	46.3
python-jose	QwenCoder 2.5	16.7	35.2
shortuuid	Gemini 2.0	22.2	66.7
shortuuid	QwenCoder 2.5	25.0	77.8
tinydb	Gemini 2.0	41.3	60.9
tinydb	QwenCoder 2.5	34.8	63.0

extraction, and a more semantic, model-driven strategy employing an LLM-based summary of test failures.

The regex-based mode extracts concise error messages directly from the test output logs by matching common patterns. These summaries are typically short and focused on the immediate cause of failure—such as a missing attribute, a type mismatch, or a failed assertion. For example, after generating the initial implementation:

```
def tables(self) -> Set[str]:
    return set(self._storage.keys())
```

the regex-based summary identified the recurring issue:

```
AttributeError:  'MemoryStorage' object has no attribute 'keys'
```

This kind of feedback allows the model to quickly realize the underlying misunderstanding (e.g., treating the storage object as a dictionary) and produce corrected versions. Through successive iterations and increasingly specific errors (e.g., handling of `NoneType` or checking if the storage was opened), the function was eventually refined to:

```
def tables(self) -> Set[str]:
    if not self._opened:
        raise RuntimeError("Database is not opened")
    data = self._storage.read()
    if data is None:
        return set()
    return set(data.keys())
```

which passed all test cases successfully.

In contrast, the **LLM-based error mode** operates by summarizing the full test output using another LLM, producing more verbose and semantically rich feedback. Instead of simply reporting exceptions, it contextualizes them within the structure of the test logic. For instance, for an incorrect generation such as:

```
def tables(self) -> Set[str]:
    return set(self._opened.keys())
```

the feedback might include multiple compacted assertions like:

```
assert [] == list(db.tables()) → AttributeError:  'bool' object has no
attribute 'keys'
assert "persisted" in db.tables() → AttributeError:  'bool' object has no
attribute 'keys'
assert db.tables() == {"_default", "table1", "table2"} → AttributeError:
'bool' object has no attribute 'keys'
```

This mode is more expressive and often captures richer information, such as the logical relationships between tests, side effects, or inferred expectations. However, the verbosity can introduce noise, and in some cases, less targeted guidance may lead to slower convergence or even hallucinated fixes.

Empirical results (see Section 5.1) show that both methods can be effective, but their performance varies depending on the model and context. The regex-based mode tends to be more stable and focused, making it especially effective in simpler repositories or with models that respond well to low-level feedback. LLM-based summaries, by contrast, are more useful in complex scenarios requiring higher-level understanding or when the root cause of failure is distributed across multiple test cases.

Ultimately, the choice between these strategies represents a trade-off between *precision and abstraction*, and their comparative impact is analyzed in the following sections.

Success Rate and Iteration Counts

Table III summarizes the average number of iterations required to reach a passing solution across the various repositories and models. In the single-pass configuration, the iteration count is fixed at 1.0 by definition. In contrast, the iterative refinement process—employing either LLM-based or regex-based error feedback—yields modestly higher iteration counts. For instance, in the Box repository with the Gemini 2.0 model, the LLM-based approach required an average of 2.3 iterations, whereas the regex-based method converged in 1.7 iterations. In the same repository, when using QwenCoder 2.5 Instruct, both iterative strategies converge in approximately 1.2 iterations. Similar trends are observed for the PySnooper repository, where Gemini 2.0 improves from a single-pass of 1.0 to 1.7 iterations with LLM feedback and to 1.2 iterations with regex feedback, while QwenCoder 2.5 Instruct consistently converges after about 1.2 iterations.

More complex repositories such as arrow and python-jose tend to require slightly higher iteration counts, ranging from 1.3 up to 3.0 iterations, which suggests that more sophisticated error feedback is occasionally necessary to resolve intricate issues. Nevertheless, the overall average iteration counts remain low. It is important to note that the cap of 10 iterations was set solely for experimental purposes to observe the system's behavior under a conservative upper bound. In practical applications, the vast majority of functions are corrected within just a few cycles.

These findings indicate that incorporating error feedback—whether via LLM-based summarization or regex-based extraction—substantially enhances the success rate over the single-pass approach, while only minimally increasing the average number of iterations needed. The relatively low average iteration counts support the view that iterative refinement is an effective strategy for overcoming initial generation shortcomings, and that further efficiency improvements may be achieved in practical settings.

5.2 Comparison of General-Purpose and Code-Specific LLMs

The experimental evaluation also provides an opportunity to directly compare the performance of two distinct LLM types: a general-purpose model, Gemini 2.0, and a code-specific model, QwenCoder 2.5 Instruct. Although both models are capable of generating syntactically correct and functionally viable code, they differ in several key performance aspects.

Gemini 2.0, designed for versatility across various language tasks, typically exhibits a lower success rate in single-pass mode. However, when iterative refinement is applied, its success rate improves significantly, suggesting that its initial outputs benefit considerably from the

TABLE III: AVERAGE NUMBER OF ITERATIONS (FOR SUCCESSFUL GENERATIONS) ACROSS REPOSITORIES, COMPARING SINGLE-PASS (1.0) AND ITERATIVE REFINEMENT USING LLM-BASED AND REGEX-BASED ERROR SUMMARIES.

Repository	**Model**	**LLM Iter.**	**Regex Iter.**
Box	Gemini 2.0	2.3	1.7
Box	QwenCoder 2.5	1.2	1.2
PySnooper	Gemini 2.0	1.7	1.2
PySnooper	QwenCoder 2.5	1.2	1.2
arrow	Gemini 2.0	1.3	2.0
arrow	QwenCoder 2.5	1.3	1.8
blessed	Gemini 2.0	2.1	1.9
blessed	QwenCoder 2.5	1.8	1.6
boltons	Gemini 2.0	1.7	1.8
boltons	QwenCoder 2.5	1.9	1.9
cerberus	Gemini 2.0	2.0	1.2
cerberus	QwenCoder 2.5	1.4	1.3
loguru	Gemini 2.0	1.9	1.8
loguru	QwenCoder 2.5	1.5	1.3
python-jose	Gemini 2.0	2.4	2.8
python-jose	QwenCoder 2.5	2.8	2.7
shortuuid	Gemini 2.0	1.4	1.5
shortuuid	QwenCoder 2.5	2.9	2.3
tinydb	Gemini 2.0	2.3	1.9
tinydb	QwenCoder 2.5	3.0	1.9

corrective feedback provided by the testing pipeline. In contrast, QwenCoder 2.5 Instruct often achieves higher success rates in single-pass mode, reflecting its specialization in code synthesis. Nonetheless, even QwenCoder 2.5 Instruct gains from iterative feedback in more complex scenarios.

A closer examination of the iteration data reveals that Gemini 2.0 generally requires more iterations to converge to a passing solution than QwenCoder 2.5 Instruct. For example, in the Box repository, Gemini 2.0 averaged 2.3 iterations with LLM-based feedback compared to approximately 1.2 iterations for QwenCoder 2.5 Instruct. This trend is consistent across several repositories, indicating that the general-purpose model benefits from additional correction cycles to adapt its output to stringent code requirements. In contrast, the lower iteration counts observed for QwenCoder 2.5 Instruct highlight its effectiveness in producing high-quality code with minimal feedback, particularly in less complex contexts.

Table IV provides an aggregated summary of key performance metrics for the two models. The table shows that the average single-pass success rate for Gemini 2.0 is 36.7%, compared to 35.2% for QwenCoder 2.5 Instruct. Under iterative refinement, the success rates improve to 56.1% and 55.7%, respectively, while the average iteration counts remain nearly identical (1.91 for Gemini 2.0 and 1.90 for QwenCoder 2.5 Instruct). In terms of execution time, however, QwenCoder 2.5 Instruct incurs a higher computational cost, with average single-pass and iterative times of 13.9 seconds and 25.6 seconds, respectively, compared to 10.7 seconds and 20.3 seconds for Gemini 2.0.

Overall, these findings underscore the importance of selecting an LLM that aligns with the specific task requirements. While general-purpose models such as Gemini 2.0 offer flexibility and lower latency, specialized models like QwenCoder 2.5 Instruct are better suited for tasks that demand precise, context-aware code generation. The trade-offs between iteration count, runtime, and success rate—as summarized in Table IV—provide valuable insights for future applications of LLM-based code generation in real-world software development.

TABLE IV: AVERAGE PERFORMANCE COMPARISON BETWEEN GEMINI 2.0 (GENERAL-PURPOSE) AND QWENCODER 2.5 INSTRUCT (CODE-SPECIFIC).

Metric	**Gemini 2.0**	**QwenCoder 2.5 Instruct**
Average Single-pass SR (%)	36.7	35.2
Average Iterative SR (%)	56.1	55.7
Average Iteration Count (LLM)	1.91	1.90
Average Single-pass Time (s)	10.7	13.9
Average Iterative Time (s)	20.3	25.6

5.3 Analysis by Repository Complexity

In this section, the performance of each code-generation approach is assessed by grouping the ten repositories according to their size and complexity. Only functions with a descriptive docstring or that are tested (directly or indirectly) are included, ensuring that each function has both conceptual guidance and meaningful test coverage. Table V shows the number of analyzed functions in each repository and the assigned tier.

TABLE V: REPOSITORY GROUPINGS BY COMPLEXITY, BASED ON NUMBER OF FUNCTIONS. SMALL REPOSITORIES HAVE UP TO 30 FUNCTIONS, MEDIUM REPOSITORIES HAVE 31–60 FUNCTIONS, AND LARGE REPOSITORIES HAVE MORE THAN 60 FUNCTIONS.

Repository	**# Functions**	**Complexity Tier**
shortuuid	9	Small
PySnooper	13	Small
Box	18	Small
loguru	24	Small
cerberus	44	Medium
tinydb	46	Medium
python-jose	54	Medium
blessed	81	Large
arrow	92	Large
boltons	237	Large

Small Repositories

In small repositories (up to 30 functions), single-pass generation can sometimes be effective for simpler logic but often fails when confronted by corner cases. Table VI summarizes success rates for shortuuid, PySnooper, Box, and loguru. Iterative refinement tends to improve correctness, particularly for utility-like functions that rely on correct handling of data structures or custom exceptions. The computational overhead remains moderate in most small repositories.

Most failures in small repositories stem from attribute errors, incorrect handling of special cases, or omissions of input validation. Regex-based error extraction is usually sufficient for direct assertion failures, while LLM-based error summaries are more helpful when multiple issues appear concurrently. The gains in success rate across all small repositories—*shortuuid*, *PySnooper*, *Box*, and *loguru*—confirm that test-driven iteration substantially reduces failures, even in relatively simple codebases.

TABLE VI: SUCCESS RATES (%) IN SMALL REPOSITORIES (UP TO 30 FUNCTIONS), COMPARING SINGLE-PASS GENERATION WITH ITERATIVE REFINEMENT UNDER LLM-BASED AND REGEX-BASED ERROR FEEDBACK.

Repository	**Model**	**Single-pass**	**Iterative (LLM)**	**Iterative (regex)**
Box (18)	Gemini 2.0	38.9	66.7	66.7
	QwenCoder 2.5	38.9	61.1	50.0
PySnooper (13)	Gemini 2.0	76.9	84.6	76.9
	QwenCoder 2.5	69.2	92.3	84.6
shortuuid (9)	Gemini 2.0	22.2	55.6	66.7
	QwenCoder 2.5	25.0	77.8	66.7
loguru (24)	Gemini 2.0	29.2	50.0	50.0
	QwenCoder 2.5	33.3	50.0	50.0

Medium Repositories

Medium repositories (31–60 functions) introduce additional complexity and domain-specific considerations. Table VII contains success rates for cerberus, tinydb, and python-jose. These codebases highlight more intricate validation logic, file or database I/O, and multi-step domain workflows. Single-pass performance rarely exceeds 40%, but iterative feedback, either via regex or LLM-based error summaries, improves outcomes in all three cases.

The iterative approach is particularly helpful in addressing non-trivial dependencies or cryptographic routines (as in python-jose). LLM-based summaries show strong utility where several assertion failures arise from interdependent causes. Meanwhile, regex-based extraction proves an efficient baseline for more direct or repeated exceptions, such as missing keys or incorrect type checks.

TABLE VII: SUCCESS RATES (%) IN MEDIUM REPOSITORIES (31-60 FUNCTIONS).

Repository	**Model**	**Single-pass**	**Iterative (LLM)**	**Iterative (regex)**
cerberus (44)	Gemini 2.0	29.6	47.7	38.6
	QwenCoder 2.5	27.3	36.4	36.4
tinydb (46)	Gemini 2.0	41.3	58.7	60.9
	QwenCoder 2.5	34.8	63.0	60.9
python-jose (54)	Gemini 2.0	20.4	38.9	46.3
	QwenCoder 2.5	16.7	35.2	33.3

Large Repositories

Large repositories (more than 60 functions) are the most challenging due to deeper dependencies, diverse function sets, and broader test coverage. Table VIII shows the success rates for blessed, arrow, and boltons. Single-pass results drop below 40% in most configurations, underscoring the complexity of replicating the original logic in a single shot. The iterative modes offer consistent improvements, although they sometimes converge more slowly, reflecting the interplay of multiple abstractions or layered modules.

Many failures in arrow and boltons relate to domain-specific edge cases, such as time-zone conversions or generic utility functions spanning multiple data types. Iterative refinement enables the models to incrementally correct these issues. The regex-based mode remains robust for relatively straightforward exceptions, while the LLM-based mode helps when interlocking functionalities trigger

compound errors.

TABLE VIII: SUCCESS RATES (%) IN LARGE REPOSITORIES (MORE THAN 60 FUNCTIONS).

Repository	**Model**	**Single-pass**	**Iterative (LLM)**	**Iterative (regex)**
blessed (81)	Gemini 2.0	37.0	46.9	42.0
	QwenCoder 2.5	37.0	46.9	43.2
arrow (92)	Gemini 2.0	27.2	30.4	35.9
	QwenCoder 2.5	27.2	31.5	34.8
boltons (237)	Gemini 2.0	43.9	54.0	54.9
	QwenCoder 2.5	42.2	59.1	58.2

Summary of Complexity Analysis

Smaller repositories can sometimes be handled with single-pass generation if functions are relatively simple, but iterative prompts deliver valuable correctness boosts in nearly all cases. In medium repositories, domain complexity grows and single-pass success rates drop, making iterative refinement increasingly important. In large repositories, single-shot code generation struggles with multi-layered interactions, and feedback-driven iteration is often necessary to reach acceptable success rates. Overall, detailed docstrings and direct or indirect test coverage allow each function to be both guided by initial context and rigorously validated, ensuring that iterative corrections can converge on robust solutions.

5.4 Lines of Code (LOC) Analysis

Table IX presents a comparative view of the original lines of code (*Orig LOC*) versus the lines of code in model-generated solutions (*Gen LOC*), along with their difference. Several trends emerge from these data. In some instances, the generated code is more concise than the reference implementation. For example, single-pass solutions for PySnooper (Gemini 2.0 and QwenCoder 2.5) are substantially shorter than the originals, often because the model omits certain checks or defensive constructs. In other repositories, such as Box with Gemini 2.0, the generated code exceeds the original by approximately two lines, reflecting additional logic inserted to handle errors or validate inputs. It is important to note that only effective lines of code were counted, excluding comments, empty lines, and formatting artifacts.

Generated vs. Original LOC

TABLE IX: AVERAGE LINES OF CODE (LOC) FOR SUCCESSFUL GENERATIONS. THE TABLE LISTS THE ORIGINAL LOC, GENERATED LOC, AND THEIR DIFFERENCE (GENERATED MINUS ORIGINAL) FOR EACH REPOSITORY AND MODEL IN SINGLE-PASS MODE.

Repository	**Model**	**Orig LOC**	**Gen LOC (Diff)**
Box	Gemini 2.0	7.3	9.4 (+2.1)
Box	QwenCoder 2.5	6.0	5.7 (-0.3)
PySnooper	Gemini 2.0	6.3	2.4 (-3.9)
PySnooper	QwenCoder 2.5	6.0	2.0 (-4.0)
arrow	Gemini 2.0	2.3	2.7 (+0.4)
arrow	QwenCoder 2.5	2.3	1.8 (-0.5)
blessed	Gemini 2.0	9.1	7.2 (-1.9)
blessed	QwenCoder 2.5	9.1	8.6 (-0.5)
boltons	Gemini 2.0	7.5	9.0 (+1.5)
boltons	QwenCoder 2.5	6.7	5.7 (-1.0)
cerberus	Gemini 2.0	2.6	2.7 (+0.1)
cerberus	QwenCoder 2.5	2.5	2.2 (-0.3)
loguru	Gemini 2.0	2.3	2.3 (0.0)
loguru	QwenCoder 2.5	2.3	2.3 (0.0)
python-jose	Gemini 2.0	2.7	3.4 (+0.6)
python-jose	QwenCoder 2.5	2.3	3.7 (+1.3)
shortuuid	Gemini 2.0	3.5	2.5 (-1.0)
shortuuid	QwenCoder 2.5	7.0	6.0 (-1.0)
tinydb	Gemini 2.0	5.1	3.6 (-1.5)
tinydb	QwenCoder 2.5	3.6	3.1 (-0.5)

The variability in generated code length does not adhere to a single pattern across repositories. In smaller projects like PySnooper, concise solutions may suffice when the underlying functionality is relatively straightforward, especially if the model omits repetitive validation checks from the original. Conversely, in repositories such as arrow or python-jose, the addition of multiple checks or error-handling statements can expand the total code. Iterative refinement further contributes to these fluctuations. In some cases, repeated attempts shorten the solution by removing superfluous lines. In others, each cycle introduces new code elements to address corner cases or align with implicit expectations revealed through the tests.

Shorter code does not invariably correlate with higher accuracy. For instance, while PySnooper shows a substantial reduction in lines, it also relies on robust testing and a clearly documented API to guide the model. In Box, the average gain of two lines for Gemini 2.0 often reflects additional logic needed to handle missed conditions in the original attempt. These observations underscore that the presence or absence of lines is less consequential than ensuring the generated code captures all required domain constraints.

Correlation with Success Rates

There is no consistent relationship between code length and success rates, as some verbose implementations fail critical tests and certain minimal solutions pass them with fewer lines. Repositories like shortuuid and PySnooper illustrate that effective outcomes can be more concise, particularly for functions whose original implementations include redundant control structures or defensive checks that the model omits while still preserving correctness. By contrast, python-jose demonstrates how higher line counts sometimes improve correctness when handling layered encryption or token verification routines. Yet even in that domain, merely adding lines does not guarantee success; the newly introduced logic must specifically address the underlying test failures.

Overall, these patterns indicate that the decisive factor for correctness lies less in code length and more in how thoroughly each solution satisfies the demands of the tests and domain constraints. The role of comprehensive docstrings and well-targeted test cases proves critical: by clarifying functional requirements, they enable the model to focus on the essential logic, be it in fewer lines or more.

5.5 Time Efficiency and Overhead

Table X compares the average execution time for successful generations using single-pass generation versus iterative refinement under two different error handling modes. The data clearly illustrate that single-pass attempts are systematically faster but often fail more frequently (see Section 5.1), whereas iterative methods require multiple prompts, resulting in higher overall runtime.

Single-Pass vs. Iterative Timing

Single-pass generation produces code in one step and runs the test suite only once. As shown in Table X, this leads to lower average times (e.g., 3.0 s for Gemini 2.0 in *shortuuid* or 4.0 s for QwenCoder 2.5 in *PySnooper*), but with limited chances to correct errors. Consequently, the success rate in single-pass mode remains lower, particularly in medium-to-large repositories with more complex dependencies or extensive tests.

Iterative refinement, on the other hand, retries generation after test failures and integrates error feedback into subsequent prompts. While this approach substantially improves correctness (see Section 5.1), each additional attempt incurs extra time. The iterative LLM-based mode can be especially time-consuming due to multiple calls to the large language model (e.g., 12.6 s for Gemini 2.0 in *Box*, compared to 3.8 s in single-pass). However, for challenging tasks, investing additional cycles to iteratively resolve errors may prove more efficient overall than manually diagnosing failures or relying solely on one-shot code generation that fails outright.

Impact of Error Handling Mode

Regex-based error extraction generally offers the fastest feedback cycle: it scrapes concise failure messages from the test logs, leaving little overhead for processing or summarization. As a result, its total runtime in iterative mode tends to be lower than the LLM-based approach (e.g., 5.5 s vs. 7.3 s for QwenCoder 2.5 in *Box*). However, regex extraction struggles to capture multiple, interlinked errors spanning multiple test assertions.

By contrast, the LLM-based summary mode provides more context-rich feedback, enabling the model to address intricate or compound errors more effectively. This advantage is particularly apparent in repositories with more complex testing scenarios or non-standard failure cases (see also Section 5.3). Yet the extra LLM calls and token usage lead to increased computational cost (e.g., 35.1 s vs. 25.0 s for QwenCoder 2.5 in *arrow*).

TABLE X: AVERAGE EXECUTION TIME (IN SECONDS) FOR SUCCESSFUL GENERATIONS ACROSS REPOSITORIES AND MODELS. THE TABLE COMPARES SINGLE-PASS EXECUTION WITH ITERATIVE REFINEMENT USING LLM-BASED AND REGEX-BASED ERROR FEEDBACK (ITERATION LIMIT = 10.0).

Repository	**Model**	**Single-pass**	**Iterative (LLM)**	**Iterative (regex)**
Box	Gemini 2.0	3.8	12.6	8.2
Box	QwenCoder 2.5	9.1	7.3	5.5
PySnooper	Gemini 2.0	3.7	6.4	4.3
PySnooper	QwenCoder 2.5	4.0	5.1	4.2
arrow	Gemini 2.0	17.0	33.8	37.6
arrow	QwenCoder 2.5	25.0	35.1	36.2
blessed	Gemini 2.0	4.3	13.1	14.8
blessed	QwenCoder 2.5	5.4	14.8	9.8
boltons	Gemini 2.0	3.9	8.0	8.7
boltons	QwenCoder 2.5	4.4	12.4	8.7
cerberus	Gemini 2.0	3.0	8.9	6.1
cerberus	QwenCoder 2.5	2.8	9.3	7.5
loguru	Gemini 2.0	14.4	26.0	23.2
loguru	QwenCoder 2.5	16.4	24.0	21.2
python-jose	Gemini 2.0	51.0	78.0	84.2
python-jose	QwenCoder 2.5	61.5	106.1	109.4
shortuuid	Gemini 2.0	3.0	6.4	5.5
shortuuid	QwenCoder 2.5	6.0	22.4	11.3
tinydb	Gemini 2.0	3.3	10.2	6.9
tinydb	QwenCoder 2.5	4.6	19.2	7.0

In essence, there is a clear trade-off between the lightweight speed of regex-based extraction and the more robust but resource-intensive LLM-based approach. For simpler failures or smaller repositories, regex-based parsing can be sufficient and faster. When failures are more intricate—perhaps involving multiple attributes, layered assertions, or domain-specific logic—the richer semantics of LLM-based error summaries may justify the higher overhead.

5.6 Discussion of Key Findings

Significance of Iteration for Complex Functions

Iterative generation yields tangible advantages when addressing functions that involve intricate dependencies or domain-specific nuances. Multiple attempts, each guided by error feedback, enable the model to incorporate incremental corrections until the function aligns with deeper architectural demands. This iterative process is particularly apparent in repositories such as *python-jose* and *boltons*, where single-pass attempts often fail on subtle inter-module or cryptographic challenges. Successive regenerations, informed by precisely captured or summarized errors, progressively adapt the code to match the intended domain logic. In settings where correctness is a primary concern, such as security-focused libraries, the capacity to refine an initially incomplete solution becomes critical.

Trade-Offs in Cost and Time

Despite the clear gains in reliability from iterative refinement, this method incurs additional computational overhead. Each unsuccessful attempt prompts fresh calls to the large language model, requiring further replacements and repeated test runs. Although single-pass generation executes rapidly, it seldom addresses complex corner cases at the first attempt. In time-sensitive or cost-sensitive scenarios, the iterative approach risks inflating token usage and execution time. However, findings show that in complex repositories, single-pass methods rarely achieve acceptable coverage without manual post-processing or debugging. When uninterrupted operation and high assurance are priorities, iterative refinement justifies the higher resource consumption by steadily converging on verified, test-passing code. In simpler repositories or in prototyping phases, a single-pass approach might suffice, although subsequent manual corrections become indispensable.

Limitations and Possible Future Works

One significant limitation of the current system is its heavy reliance on the quality and comprehensiveness of the test suites. The evaluation framework assumes that a function passing all available tests is correct; however, if the tests are incomplete or insufficiently rigorous, critical edge cases may be overlooked. Consequently, a function might pass validation while still failing in real-world scenarios. Furthermore, when error logs and assertions lack sufficient detail, both regex-based extraction and LLM-based summarization may generate inadequate feedback, thereby impeding effective iterative refinement.

It is important to note that for this study only repositories with test suites that pass 100% of their tests at the time of acquisition were selected. This controlled selection criterion ensures that any failure in the generated code is attributable to deficiencies in the generation process itself rather than to shortcomings in the test suite. In such an environment, a single test failure unequivocally indicates that the generated function does not meet the intended specifications.

To mitigate these issues, future versions of the system could incorporate mechanisms for dynamic weighting or tagging of test cases based on their coverage depth, ensuring that the most critical behaviors are rigorously evaluated. In addition, implementing a retrieval-augmented correction mechanism to maintain a repository of recurring error patterns and their successful resolutions could streamline the refinement process. By allowing the system to reuse previously effective fixes, it would reduce redundant reasoning and accelerate convergence on new tasks.

An adaptive error-handling strategy also offers a promising improvement. Simpler, well-structured logs could continue to be processed efficiently using lightweight regex extraction, while more complex or ambiguous failure cases could dynamically trigger LLM-based summarization, balancing speed and depth of understanding depending on the complexity of the test output.

Finally, further enhancements in prompt engineering and lightweight static analysis integration could improve the system's ability to interpret vague or incomplete test feedback. Incorporating minimal static checks, such as basic type validation or undefined variable detection, would enrich the feedback without sacrificing the system's lightweight and general-purpose design. Altogether, these directions aim to reduce computational redundancy while making the generated code even more robust for real-world development scenarios.

CHAPTER 6

CONCLUSION

The findings in the previous chapter reaffirm the strength and practicality of an iterative, feedback-driven approach for LLM-based code generation. Throughout this work, the single-pass strategy was used solely as a comparative reference, to highlight how introducing structured feedback loops can dramatically enhance correctness — without relying on expensive fine-tuning, additional model training, or human intervention.

Instead, the proposed method demonstrates that meaningful improvements can be achieved with lightweight means already available to any developer: project documentation, function-level docstrings, and existing test suites. By iteratively refining the generated code based solely on test feedback, the system evolves its outputs autonomously, respecting the natural development workflow while minimizing overhead.

While iterative refinement naturally entails some additional runtime compared to a single generation attempt, the overall computational costs remain modest and are justified by the significant gains in success rate and code reliability. This balance is particularly evident across a wide range of projects, from smaller libraries like Shortuuid to more complex ecosystems like Box and TinyDB, where repeated, feedback-guided generations consistently led to functional implementations.

Moreover, the system's dual error-handling strategy — lightweight regex parsing for straightforward cases and LLM-based summarization for complex failures — enables adaptable, efficient correction cycles tailored to the nature of the encountered errors.

Ultimately, this approach demonstrates that improving LLM-generated code does not require monopolizing resources, retraining models, or building specialized infrastructures. It is a lightweight, cost-effective, and developer-friendly solution that scales naturally with the complexity of real-world software projects.

CITED LITERATURE

1. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., and Dhariwal, P.: Language models are few-shot learners. Advances in Neural Information Processing Systems, 33:1877–1901, 2020.

2. Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., and Kaplan, J.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

3. Svyatkovskiy, A., Wang, S. K., and Lee, N. S.: Intellicode compose: Code generation with transformer-based model. arXiv preprint arXiv:2005.08025, 2021.

4. Li, Y., Lakshminarayanan, B., Ritchie, D., and Singh, S.: Competition-level code generation with alphacode. arXiv preprint arXiv:2203.07814, 2022.

5. Bavarian, M., Lee, F., Khabsa, M., Rawat, A., Hajishirzi, H., Kiela, D., and Lewis, M.: Efficient training of language models to fill in the middle. arXiv preprint arXiv:2207.14255, 2022.

6. Vaswani, A., Shazeer, N., Parmar, N., and Uszkoreit, J.: Attention is all you need. Advances in Neural Information Processing Systems, 30, 2017.

7. Pearce, H., Ahmad, W., McGill, M., Peebles, D., and Shmatikov, V.: Assessing the security of github copilot's code contributions. IEEE Symposium on Security and Privacy, 2022.

8. Wang, Y., Deng, S., Wang, W., Gao, Y., Wang, J., Ye, X., Chen, Q., Zhao, Z., and Xu, J.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922, 2023.

9. Nijkamp, E., Lee, B., Huang, Y., Zhang, S., Sun, W., Chang, K.-W., Smolensky, P., and Jiang, H.: Codegen-multi: Multi-turn program synthesis with pretrained models. arXiv preprint arXiv:2302.06817, 2023.

10. Sobania, D., Schuff, H., Neumann, G., and Hoppe, T.: Analysis of code generation with openai codex. arXiv preprint arXiv:2301.12359, 2023.

CITED LITERATURE (continued)

11. OpenAI: Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

12. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Lattimore, F., Brockman, G., Cottrell, G., and Schulman, J.: Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155, 2022.

13. Beck, K.: Test-Driven Development: By Example. Addison-Wesley, 2002.

14. Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J.: Bleu: A method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pages 311–318, 2002.

15. Hendrycks, D., Burns, C., Basart, S., Zou, A., Song, D., Steinhardt, J., and Jacob, M.: Measuring coding challenge competence with apps. arXiv preprint arXiv:2105.09938, 2021.

16. Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W.: Repocoder: Repository-level code completion through iterative retrieval and generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 2471–2484, 2023.

17. Pan, Z., Hu, X., Xia, X., and Yang, X.: Enhancing repository-level code generation with integrated contextual information. arXiv preprint arXiv:2406.03283, 2024.

VITA

NAME	Lorenzo Gallone
CONTACT	+1 (464) 204-8833 — lorenzo.gallone@gmail.com — www.linkedin.com/in/lorenzo-gallone
EDUCATION	
	Master of Science in “ Computer Science ”, University of Illinois at Chicago, May 2025, USA
	Specialization Degree in “ Computer Engineering ”, Jul 2025, Polytechnic of Turin, Italy
	Bachelor’s Degree in Computer Engineering, Jul 2023, Polytechnic of Turin, Italy -
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2023 - IELTS examination (7.0/9)
	A.Y. 2024/25 One Year of study abroad in Chicago, Illinois
	A.Y. 2023/24. Lessons and exams attended exclusively in English
SCHOLARSHIPS	
Fall 2024	Italian scholarship for final project (thesis) at UIC
Fall 2024	Italian scholarship for TOP-UIC students
TECHNICAL SKILLS	
Programming Languages	Python, Java, MATLAB, SQL, JavaScript, PHP, HTML5, C, C++, Bash, Kotlin
Frameworks	React, NodeJS, Firebase, GitHub, Linux, Docker
Skills	Software Engineering, Artificial Intelligence, Neural Networks, Databases, Data Management, Programming & Software Development, Academic Research, Web Application Development, System Design, Object Oriented Programming

VITA (continued)

ACADEMIC PROJECTS & APPLIED EXPERIENCE

2024 – Present	**LLM-Powered Code Generation System**
	Master's thesis at University of Illinois at Chicago on developing an AI-powered system for automatic Python code generation using large language models. Integrated automated testing and iterative refinement to ensure accuracy. Demonstrated strong problem-solving, adaptability, and independent research, aiming for a research publication.
2024 – 2025	**Secure Web Application**
	Joint project between Politecnico di Torino and University of Illinois at Chicago. Developed a secure, real-time multiplayer web application using React, Node.js, and Express. Combined creative UI design with robust backend security. Highlighted teamwork, cross-cultural collaboration, and practical problem-solving.
2024 – 2024	**Professional Task Manager (Android)**
	Developed a Kotlin-based Android application featuring live chat and multi-team support. Delivered a fully functional prototype under tight deadlines, demonstrating agility, time management, and a practical, hands-on approach.