Code Generation and Testing in the Era of AI-Native Software Engineering

by

Noble Saji Mathews

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Noble Saji Mathews 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large Language Models (LLMs) like GPT-4 and Llama 3 are transforming software development by automating code generation and test case creation. This thesis investigates two pivotal aspects of LLM-assisted development: the integration of Test-Driven Development (TDD) principles into code generation workflows and the limitations of LLM-based test-generation tools in detecting bugs.

LLMs have demonstrated significant capabilities in generating code snippets directly from problem statements. This increasingly automated process mirrors traditional human-led software development, where code is often written in response to a requirement. Historically, Test-Driven Development (TDD) has proven its merit, requiring developers to write tests before the functional code, ensuring alignment with the initial problem statements. Applying TDD principles to LLM-based code generation offers one distinct benefit: it enables developers to verify the correctness of generated code against predefined tests. This paper investigates if and how TDD can be incorporated into AI-assisted code-generation processes. We experimentally evaluate our hypothesis that providing LLMs like *GPT-4* and *Llama 3* with tests in addition to the problem statements enhances code generation outcomes. We experimented with established function-level code generation benchmarks such as MBPP and HumanEval. Our results consistently demonstrate that including test cases leads to higher success in solving programming challenges. We assert that TDD is a promising paradigm for helping ensure that the code generated by LLMs effectively captures the requirements.

As we progress toward AI-native software engineering, a logical follow-up question arises: Why not allow LLMs to generate these tests as well? An increasing amount of research and commercial tools now focus on automated test case generation using LLMs. However, a concerning trend is that these tools often generate tests by inferring requirements from code, which is counterintuitive to the principles of TDD and raises questions about their behaviour when the flawed assumption of the code under test being correct is violated. Thus we set out to critically examine whether recent LLM-based test generation tools, such as Codium CoverAgent and CoverUp, can effectively find bugs or unintentionally validate faulty code. Considering bugs are only exposed by failing test cases, we explore the question: can these tools truly achieve the intended objectives of software testing when their test oracles are designed to pass? Using real human-written buggy code as input, we evaluate these tools, showing how LLM-generated tests can fail to detect bugs and, more alarmingly, how their design can worsen the situation by validating bugs in the generated test suite and rejecting bug-revealing tests. These findings raise important

questions about the validity of the design behind LLM-based test generation tools and their impact on software quality and test suite reliability.

Together, these studies provide critical insights into the promise and pitfalls of integrating LLMs into software development processes, offering guidelines for improving their reliability and impact on software quality.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Mei Nagappan, for his guidance and support throughout my academic journey. From accepting me as a Mitacs intern during my undergraduate studies to mentoring me as a Master's student, he has always been a constant source of encouragement. His support during tough times and his belief in my potential have meant a lot to me. He has provided me with incredible opportunities to grow as a researcher and has always pushed me to do my best.

I am grateful to all the professors at the University of Waterloo whom I have had the chance to work with or learn from. They have helped me discover my love for research, and I deeply appreciate the opportunities I had to explore areas beyond my thesis work. Special thanks to Professor Jo Atlee and Professor Pengyu Nie for serving as the readers for this thesis and for their valuable feedback.

I also want to thank the members of the SWAG Lab for making the lab such a fun and welcoming place that I looked forward to coming to every morning. You've made it a home away from home, and I have truly enjoyed every moment spent with all of you.

I am also grateful to my friends who have shared this research journey with me and to my undergraduate supervisor, Dr. Sridhar Chimalakonda, for inspiring me to pursue Software Engineering research and for showing me what an exciting field it is. I am ever thankful for your continued support and encouragement.

Finally, I wish to thank my family. To my parents, thank you for your endless support and for always having my back and enabling me to chase my dreams. To my brother, thank you for being my role model and for always being there when I needed guidance. I wouldn't be here without your love, encouragement, and prayers.

Dedication

Dedicated to my family and friends, whose unwavering support and encouragement have brought me this far.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements v

Dedication vi

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Problem Statement 2

1.2 Thesis Overview 3

1.3 Contributions 3

1.3.1 Empirical Contributions 3

1.3.2 Technical Contributions 4

2 Test-Driven Development and LLM-based Code Generation 5

2.1 Introduction 5

2.2 Research Questions 7

2.3 Methodology . . . 9
2.3.1 Datasets Used . . . 10
2.3.2 TGen Framework . . . 13
2.4 Impact of TDD in Code Generation . . . 15
2.4.1 How good is basic code generation? . . . 16
2.4.2 Does providing test cases improve code generation? . . . 16
2.4.3 Can failed tests help LLMs fix their mistakes? . . . 17
2.4.4 Do the generated solutions only satisfy the supplied tests or even unseen tests? . . . 19
2.5 Parameters affecting TDD . . . 21
2.5.1 How does problem difficulty affect these results? . . . 22
2.5.2 How many tests is good enough? . . . 25
2.5.3 Do these results hold with an open model like Llama 3? . . . 26
2.6 Related Work . . . 28
2.7 Threats to Validity . . . 29
2.8 Conclusion . . . 30

3 Evaluating LLM-based test generators on buggy code 32
3.1 Introduction . . . 32
3.2 Related Work . . . 33
3.3 Methodology . . . 34
3.3.1 Dataset . . . 34
3.3.2 Test Generation Tools . . . 35
3.3.3 Evaluation Pipeline . . . 38
3.4 Results . . . 39
3.5 Discussion . . . 41
3.6 Threats to Validity . . . 43
3.7 Conclusion . . . 43

4 Conclusion **44**

4.1 Opportunities for Future Research 45

4.1.1 Investigating the Tradeoff between Effort and Accuracy in TDD Workflows 45

4.1.2 Improved Test Generation Frameworks 45

References **46**

List of Figures

2.1 Overview of the evaluation pipeline 9

2.2 Configuration of the Remediation Agent 11

2.3 Prompt used by the Coder Agent 12

2.4 GPT4: Problems resolved and strategies used for Function-level code generation benchmarks 15

2.5 Problems resolved and strategies used compared across difficulty levels using GPT4 21

2.6 Distribution of difficulty levels associated with problems solved by each strategy 24

2.7 Correctness with increasing number of tests supplied 25

2.8 Llama 3: Problems resolved and strategies used for Function-level code generation benchmarks 27

3.1 Overview of the evaluation pipeline 34

4.1 General vision and opportunities for future work 45

List of Tables

2.1 Impact of private tests on GPT-4 based TGen results with improvement offered by tests after validation highlighted 19

2.2 Difficulty Levels of Selected CodeChef Problems 22

2.3 Overview of the number of public tests in each benchmark 24

2.4 Impact of private tests on Llama 3 based TGen results with improvement offered by tests after validation highlighted 28

3.1 Evaluation Results of Generated Test Suites 40

Chapter 1

Introduction

Large Language Models (LLMs) over the past year have shown to be effective tools for tasks involving multistep reasoning, such as generating code and addressing various stages of the software development lifecycle [38, 52, 54]. Models have demonstrated strong performance in not just code generation, but also issue resolution tasks, which involve localization and patching [31]. This progress points to a shift towards an intent-driven programming style, referred to as "AI-native Software Engineering" [24]. However, while these models frequently produce syntactically correct code, logical errors or misinterpretations of the underlying requirements remain common [41]. This issue is particularly concerning, as automated code generation gains traction in production environments.

Outsourcing code development has long faced challenges with misunderstood requirements and quality concerns [29]. Testing remains a critical component of software quality assurance. Test-Driven Development (TDD) [7] is a well-established paradigm designed to ensure that the code meets predefined requirements. By writing tests first and then producing code to pass them, TDD enforces a cycle in which correctness is continuously validated. As LLMs integrate into software development workflows, leveraging human-written tests to guide code generation becomes increasingly important. However, the question of whether LLMs understand tests and if providing LLMs with test cases can improve code correctness remains largely unexplored.

Meanwhile, research and commercial efforts are investigating the automatic generation of test cases using LLMs [14, 18, 46, 47]. These tools infer requirements from the code itself, assuming the code under test is correct. This underlying assumption contradicts the core purpose of testing: to detect, not confirm, incorrect behaviour. Tests that simply validate existing code may overlook bugs, creating a false sense of security.

1.1 Problem Statement

Large Language Models (LLMs) are increasingly being used for code and test generation. Two challenges that limit their adoption in production are:

- **Correctness of Generated Code:** LLMs often produce syntactically valid code that fails to satisfy logical and functional requirements. This creates risks in production, as undetected errors may lead to faulty implementations.
- **Reliability of LLM-Based Test Generation:** Existing tools for test generation using LLMs commonly infer requirements from the code under test, assuming its correctness. This approach can lead to engrained bugs within the test suite.

To address these challenges, this thesis emphasizes the role of Test-Driven Development (TDD) as a design paradigm. TDD requires developers to write tests before generating code, thereby establishing a clear "contract" that LLMs can follow during code generation. This shift in workflow has the potential to improve the correctness of generated code and enable developers to focus their efforts towards creating more effective test suites.

Thesis Statement: TDD, as a design paradigm, can address key challenges of LLM-based code generation by using pre-written tests as requirements, thus aligning automated workflows with established software engineering principles. In contrast, generating tests from code risks creating ambiguous or incomplete test suites, potentially embedding errors into the testing process.

We hypothesize this approach may offer several advantages:

1. Help establish a "contract" for the LLM, ensuring that generated code aligns with predefined requirements.
2. Redefine the developer's role, focusing on creating robust tests and specifications, while delegating coding tasks to the AI agent.
3. Reduce reliance on post-hoc test generation tools, which may fail to capture underlying defects.
4. Eases integration of AI-generated code with the test suite as a set of guarantees that are clear to the developer.

1.2 Thesis Overview

This thesis is structured around two main empirical studies that address the above research questions, followed by a discussion of their implications for practice and future research.

- **Chapter 2: TDD and LLM-Assisted Code Generation.** This chapter examines the role of TDD-inspired workflows in guiding LLM-based code generation. Can TDD principles improve the correctness of code generated by LLMs? While writing tests before code is a well-established practice in traditional software engineering, its effectiveness in LLM-driven workflows has not been fully examined. We evaluate if and how pre-written tests guide LLMs to produce logically correct implementations.
- **Chapter 3: Evaluating LLM-Based Test Generation Tools.** Existing LLM-based test generation tools start from code instead of requirements and seem to prioritize coverage and discard failing tests, which might inadvertently validate incorrect implementations. This chapter investigates how these practices affect the quality and reliability of generated test suites, once again making a case for TDD.

1.3 Contributions

The contributions of this thesis are both empirical and technical:

1.3.1 Empirical Contributions

This thesis provides the following empirical insights:

1. **Impact of Tests on Code Generation:** Introducing tests as input improves code generation correctness across function-level benchmarks (MBPP and HumanEval) by 12.0% and 8.5%, respectively. Tests help LLMs resolve issues such as incorrect function signatures, logical errors in mathematical computations, and edge-case handling (Chapter 2).
2. **Effectiveness of Remediation Loops:** Iterative remediation loops further enhance problem-solving rates by an additional 5.26% (MBPP) and 5.49% (HumanEval). This iterative approach addresses issues such as type mismatches, logical flaws, and edge-case failures (Chapter 2).

3. **Private Test Validation:** Code solutions developed using public tests do seem to generalize to private test suites, still showing net gains of 18.04% (MBPP), 14.64% (HumanEval), and 7.27% (CodeChef)(Chapter 2).

4. **Parameters that affect TDD:** . We further manually inspect the data to highlight the parameters that seem to affect TDD including the number of tests, coverage, models used and difficulty (Chapter 2).

5. **Critical Gaps in LLM-Based Test Generation Tools:** Current tools for automated test generation, such as GitHub Copilot, CoverAgent, and CoverUp, fail to effectively detect bugs, with up to 68.1% of their test suites validating incorrect implementations (Chapter 3).

6. **Key Challenges and Design Limitations:** Filtering mechanisms in test generation tools often discard valuable bug-revealing tests and the prioritization of coverage metrics as a cutoff leads to suites that aren't as comprehensive as their manually written counterparts (Chapter 3).

1.3.2 Technical Contributions

In addition to the empirical insights gained from this work, the thesis also introduces a framework and an evaluation pipeline developed as a direct outcome of the empirical studies:

1. **TDD Framework for LLM-Based Code Generation:** We propose a TDD-inspired framework *TGen*, demonstrating that pre-written tests do improve LLM-based code generation. This framework also incorporates remediation loops, enabling iterative refinement of generated code until it meets predefined requirements (Chapter 2).

2. **Evaluation Pipeline for Test Generation Tools:** A systematic evaluation pipeline is introduced to assess the effectiveness of test generation tools and we outline key limitations and design flaws in existing tools (Chapter 3).

Chapter 2

Test-Driven Development and LLM-based Code Generation

An earlier version of the work in this chapter was presented at ASE 2024.
DOI: 10.1145/3691620.3695527

2.1 Introduction

Large Language Models (LLMs) are transforming software development. However, the importance of correctness in this domain remains a challenge. Despite their advanced capabilities, LLMs often exude deceptive confidence in their outputs, which can be misleadingly erroneous [38, 52, 54]. This issue becomes especially critical when the cost of failure is high. The potential pitfalls of LLMs, such as generating syntactically correct, but logically flawed code or failing to adhere to specific requirements, underscore the need for stringent validation mechanisms. Ensuring correctness is not just about preventing errors. It is about building trust in machine-generated code and ensuring it meets the rigorous standards required in real-world applications. Programmers will need to act as quality assurance specialists, spending more time checking the validity of autogenerated code, utilizing both traditional testing tools and advanced automated program repair techniques to ensure code quality [41]

Generating tests has gained additional momentum with the emergence of LLMs, and recent techniques boast testing performance similar to human-written tests. Past work,

however, has questioned how coverage as a goal may not be an effective target to set [22]. If LLMs fail to handle edge cases regarding code, generated tests derived from the implementation can be as wrong as the generated code itself. While LLMs might be useful in helping developers formalize intent, this work does not seek to answer the question, "When and how should AI write tests?". This work rather seeks to explore whether there might be a better paradigm for utilizing tests in LLM-based code generation.

Test-driven development (TDD) [7] offers a structured approach to mitigate the risks associated with code generation. This software development process revolves around the repetitive cycle of writing a test for a specific function or feature, developing code to pass that test, and subsequently refactoring the code to meet the desired standards of quality and efficiency. Hence, TDD could serve as a critical framework for validating the correctness and functionality of LLM-generated code.

There is extensive research into generating test cases from code, whether human or machine-authored [34, 49]. However, the reciprocal process has not been systematically evaluated, particularly regarding how human-written tests (created as part of the TDD process) can improve LLM-generated code. By examining how LLMs respond to the additional context and constraints of test cases, we can gain insights into their potential to enhance code correctness through a more informed code generation process. While it may seem obvious to assume that including tests alongside problem statements will inherently enhance the correctness of machine-generated code, this assumption requires careful examination. The actual impact of human-written tests on improving the quality of LLM-generated code remains underexplored. Specifically, the extent to which LLMs can interpret and utilize these tests to refine their code outputs has not been thoroughly investigated. This gap highlights a critical area for empirical study: measuring the effectiveness of test cases in guiding LLMs toward generating more accurate code that captures requirements which may not be explicitly stated (much like the goals of TDD).

In our empirical study, we create a simple framework *TGen* to evaluate our research questions. *TGen* is rooted in TDD principles, requiring developers to precisely articulate their coding intents through tests. *TGen* operates within a continuous self-improving loop. Developers begin by detailing their requirements and intended functionalities through a combination of problem statements and test cases. These test cases then serve as a definitive guide for the expected behaviour of the code, allowing *TGen* to generate code that adheres to these requirements. Integral to the framework is an LLM-based feedback mechanism designed to iteratively refine and correct the generated code, ensuring it passes all provided unit tests. By leveraging test cases as a benchmark for validation, *TGen* attempts to generate code satisfying these requirements. In cases where *TGen* cannot create correct code, by design, developers know that the code generated does not pass all

the test cases and, therefore, can correct the code before including it in their projects.

We investigate whether incorporating tests in the code generation process improves the model's ability to generate correct code and also discuss the impacts and challenges of this approach. While we use *TGen* in our study for evaluation, it's important to note that our contribution is not the framework itself, but the empirical study we conduct. We systematically explore whether the TDD paradigm works in AI-driven code generation, under what conditions it is effective, and how tests can be useful. Our findings can support or challenge the use of TDD in this context.

By integrating TDD, developers can continuously test the outputs of LLMs against predefined specifications and use cases. This helps identify and correct errors early in the development process and aligns the model's outputs with the specific requirements of the task at hand. Thus, the adoption of TDD for code generation could stand as a promising strategy to enhance the practical utility of these advanced AI systems in software development.

2.2 Research Questions

Our goal is to examine the baseline capabilities of neural code generation and the incremental benefits introduced by incorporating tests. Specifically, we explore if and how tests are helpful through the following four research questions:

RQ1: How good is basic code generation? Basic neural code generation, particularly with LLMs, has shown promising results in various tasks. However, its efficacy in complex coding scenarios without any test information remains a subject of exploration. We aim to evaluate the fundamental capabilities of these models in generating syntactically and logically correct code, especially in standard programming tasks. This investigation will provide insights into the baseline performance of LLMs in code generation, setting a benchmark for further discussions.

RQ2: Does providing test cases improve code generation? The foundational principle of TDD is that software should be developed primarily by writing tests before any functional code is written. This method has historically led to robust, well-documented software that is closely aligned with user requirements. By requiring developers to define explicit tests upfront, TDD ensures that all new code functions as intended and adheres strictly to the specified requirements. This rigorous approach to software development could be transformative when applied to the code-generation

capabilities of LLMs. We hypothesize that by providing LLMs like GPT-4 with both problem statements and corresponding tests, the generated code will achieve the required functionality and exhibit an enhanced adherence to the intended specifications and requirements.

RQ3: Can failed tests help LLMs fix their mistakes? We investigate the ability of LLMs to learn from their errors through remediation loops. This involves assessing whether LLMs can iteratively refine and correct their code outputs based on feedback or error detection. The remediation loop in our study closely mirrors the iterative cycle at the heart of TDD. In traditional TDD, the process involves initially writing tests, which the new code then fails, leading developers to refine the function until it passes the tests. We aim to explore the potential of LLMs to self-correct as reported in past literature [58]and the ability to align better the code they generate with the provided specifications.

RQ4: Do the generated solutions only satisfy the supplied tests or even unseen tests? This research question explores a critical aspect of AI-generated code: its robustness and generalizability beyond the explicitly defined test cases. In traditional software development, the ability of code to perform well under a variety of scenarios—especially those not explicitly anticipated during testing—is a hallmark of high-quality software. TDD is grounded in the principle that tests are pivotal in improving design and functionality. However, there remain questions about the extent to which this principle applies when code is generated by LLMs.

In addition to these research questions, we delve deeper into the complexities of incorporating tests, discussing our findings further through the following additional explorations:

EXP1: How does problem difficulty affect these results? We hypothesize that a problem statement's difficulty could significantly impact an LLM's performance in code generation. We aim to explore how varying levels of problem complexity influence the model's ability to generate correct code that satisfies the requirements expressed by the statement and supplied test cases. We curate our own dataset consisting of competitive programming problems of different difficulty levels and attempt to understand how increasing difficulty impacts our approach.

EXP2: How many tests is good enough? Determining the optimal number of tests necessary for effective code generation by LLMs is a key concern. In traditional TDD, the number and quality of tests are crucial for ensuring robust software. We

aim to investigate how the quantity of provided test cases influences the correctness of the code generated by LLMs. While intuitively, adding tests should only help clarify the requirements. However, when working with LLMs, we are limited by the context length of the model, which controls how much information we can feed the model. By experimenting with varying numbers of tests, we wish to identify if there exists a threshold that balances thoroughness and efficiency, guiding best practices for using tests to enhance LLM-based code generation.

EXP3: Do these results hold with an open model like Llama 3? While our approach is LLM agnostic, the LLM itself plays a major role in our experiments, requiring the ability to write correct code and review and fix incorrect code. While there may exist an ideal combination of different LLMs specialized for each of these tasks, we wish to explore if this structure would work with other models outside OpenAI's GPT family of models as well. By evaluating Meta's LLama 3 under the same conditions, we can assess if similar patterns emerge, providing insights into the consistency and reliability of our findings across different LLMs.

2.3 Methodology

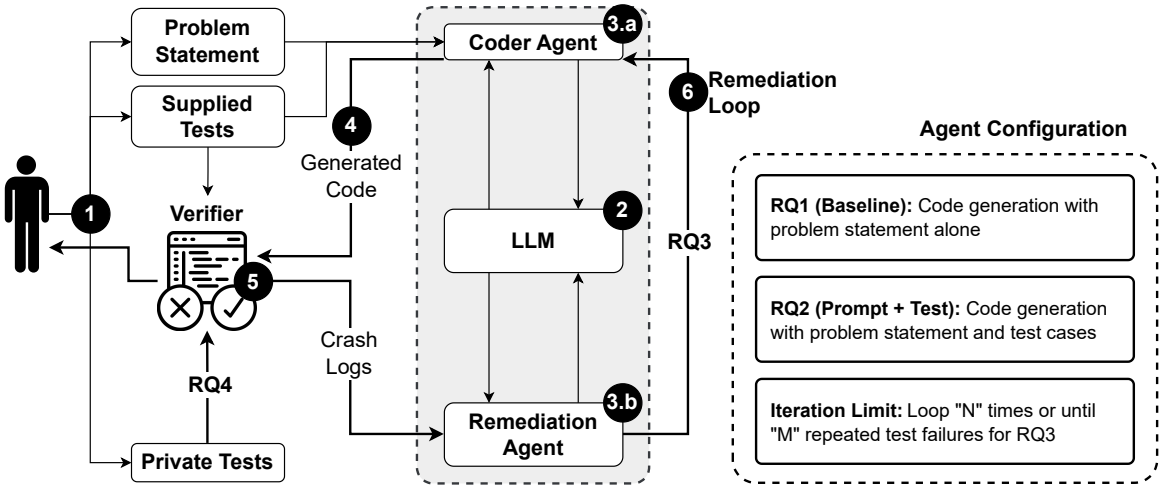


Figure 2.1: Overview of the evaluation pipeline

We design our experiments to help evaluate the effectiveness of integrating TDD with

LLM-based code generation. We aim to determine if tests affect LLMs and when they might be useful.

2.3.1 Datasets Used

Evaluating code generation typically employs benchmark datasets such as MBPP [4], HumanEval [11], CodeContests [37], or APPS [25]. Recently, newer datasets have explored class-level [17] and repository-level code completion [63], as well as automated software engineering [30]. However, we focus on function-level generation datasets like MBPP and HumanEval for two reasons.

1. MBPP and HumanEval datasets are standard benchmarks widely used to evaluate LLMs on code generation leaderboards [17, 38].

2. Repository-level code generation requires context about the codebase, making it complex and less controlled. Class-level benchmarks can be evaluated in various generation configurations, introducing many variables complicating the analysis. Function-level generation datasets like MBPP and HumanEval are relatively more straightforward. These involve generating the body of a function with a given signature, providing a clear and manageable scope for evaluation. By focusing on MBPP and HumanEval, we can systematically explore our research questions in a controlled environment where we can focus on TDD and its effects.

By establishing the effectiveness of TDD in LLM-based code generation using these datasets, we lay the groundwork for exploring more complex datasets focused on real-world TDD data in future studies. Conversely, if TDD proves ineffective with these benchmarks, it may indicate limited applicability in this context. We utilized refined versions of HumanEval and MBPP datasets curated by EvalPlus [38], which adds additional tests to allow for a more rigorous evaluation of the generated code.

- **HumanEval**: Contains 164 handwritten Python programming problems. Each problem includes a function signature, docstring, body, and several unit tests. The EvalPlus variant we use includes additional unit tests (80 times the original) for each of the 164 problems, providing more extensive coverage and thorough testing.

- **MBPP**: Comprises around 1,000 crowd-sourced Python programming problems designed for entry-level programmers. Each problem includes a task description, a

System: You are an expert Python programmer, your job is to look at code and reason why it doesn't work as intended. Once you reason why it doesn't work, you will provide a prompt that highlights the key points that need to be fixed in the code. You will not write the code yourself. IMPORTANT: Tests cannot be modified. Never make a suggestion outside the scope of modifying the solution code snippet
Assistant: *¡Code Snippet under remediation¿*
User: This code is not correct as it led to the following issues:
¡List of issues following test execution extracted by log-analyzer¿

Figure 2.2: Configuration of the Remediation Agent

code solution, and three automated test cases. The version we use includes 35 times the original number of unit tests for each problem. It consists of 399 tasks, which are a subset of the original MBPP dataset. This subset is selected by the authors of EvalPlus from the sanitized MBPP (a set of 427 manually filtered by the original MBPP authors), with further removal of low-quality and ill-formed tasks for benchmark quality control.

We never supply the LLM with any additional test cases that EvalPlus added using dynamic program testing techniques and only use them to test correctness. However, existing work in the literature has questioned the accuracy of function-level benchmarks for evaluating LLMs [9, 48]. We also look at file-level generation to address this concern and validate our findings. For this, we curate our own dataset of 1100 problems from a popular competitive programming website, CodeChef, for the following reasons:

- We also wish to explore how difficulty impacts our results, and existing datasets do not have a balanced set of problems spanning various difficulty levels
- CodeContests v2 is not public, and the older dataset is outdated and filled primarily with Python 2 solutions and problems that no longer accept solutions on the competitive programming platforms they were extracted from
- APPS dataset, which is a more recent dataset of coding challenge problems, does not give a clear difficulty measure, but rather categorizes problems into abstract buckets, which doesn't work well for our purposes

Your task is only to write a Python function to satisfy requirements specified by the users Prompt.
Only generate a single complete Python code snippet. Start generating after the [PYTHON] tag. Your python solution should end with a [/PYTHON] tag.

| *Multi-step prompting for case with tests supplied*

1. Look at the "Prompt" and "Tests" provided to understand the users requirements.
2. Generate code based on requirements in the prompt and ensure the logic is such that it would pass the corresponding tests provided.
3. Do not attempt to programmatically verify the tests. The test should not be part of the generated code in any way.
4. Do not write code in order to satisfy the tests, primarily focus on satisfying the prompt.
5. Include necessary imports in the generated code.
6. Your code is expected to read input from stdin and write output to stdout as per the requirements in the prompt.
7. Define additional helper functions if needed
8. Your output should be as shown in the sample tests. Ensure you do not return or print any additional information that can cause the tests to fail.

| *Dataset-specific guidelines for function-level case*

If starting code is provided look at the issues mentioned and attempt to fix them
Ensure that the signature of the function is the same as the one provided by the user
And format your response as follows:
[PYTHON]
def function_name(parameters_as_provided_in_signature):
{
Ensure the function signature is the same as the one provided in the signature
Infer meaning of parameters from the prompt and tests
Valid python code with necessary imports if any that satisfies the prompt
Ensure the entry method is a function with the same name as the function name in the tests
}
[/PYTHON]

Figure 2.3: Prompt used by the Coder Agent

2.3.2 TGen Framework

Figure 2.1 provides an overview of the proposed *TGen* framework's pipeline, which we use for evaluation. Each part of the pipeline is described below:

1. **Input Phase**: The evaluation pipeline starts by accepting two main inputs: the *problem statement* describing the task and the *tests* containing the public unit tests and the required signature, if any, to be enforced in the desired solution. These tests serve both as specifications for the code to meet and as a verification tool for the output. These tests serve as both specifications and verification tools. Tests provided to the LLM during generation are labelled as supplied tests, while the rest are private tests. For competitive programming problems, the challenge description is the problem statement, and available public tests are supplied tests.

2. **LLM Engine**: The core reasoning engine, powered by state-of-the-art LLMs (e.g., Meta Llama 3, GPT-3.5 & GPT-4 Turbo), processes the structured input. The foundational model leverages its own knowledge and any supplied context to generate an output response. This study uses GPT-4 Turbo v1106 for all experiments, with results validated using Meta Llama 3.

3. **LLM Agents**: In the context of Large Language Models (LLMs), "agents" are systems that utilize LLMs as their underlying reasoning engine [8]. These agents are designed with predefined prompts and thought patterns, enabling them to interact intelligently with users or other systems. Agents in our pipeline utilize the intent and available context from the input phase to guide the LLM engine in generating code, reviewing snippets, and suggesting fixes.

   (a) **Coder Agent**: This agent takes the supplied information and generates a code snippet from it. The system prompt used when tests are supplied for function-level generation cases is shown in Figure 2.3. The first part provides a rough thought process to instruct the LLM on how to utilize the tests, and the second part contains instructions specific to the structure of tasks in a dataset. Though the overall structure of the prompt remains the same, variants of this are used in different cases, namely, when no tests are supplied or when a file-level generation task is performed.

   (b) **Remediation Agent**: This agent looks at failures reported by the verifier and attempts to suggest how to fix the code, which is then taken up by the "Coder Agent" in the next iteration. The structure of the prompts is shown in Figure 2.2.

4. **Output Phase**: The *Generated Code* is extracted from the LLMs output, which is then subject to validation. The output is expected to satisfy both the functional requirements and the constraints imposed by the tests, if any.

5. **Verification**: The output of the coder agent is always run through the verifier that uses PyTest, a Python unit-testing framework, to verify the generated code against all available tests. The verifier collects crashes or test failures, and creates a comprehensive list of failures for the LLM to inspect and address issues. The verifier identifies any discrepancies and ensures only correct solutions are accepted and returned to the user.

6. **Remediation Loop**: The system enters a remediation loop if the generated code fails the tests. The remediation agent uses failure information (e.g., stack trace, detected issues, executed lines) to refine feedback for the coder agent. This loop runs for a fixed number of iterations or until the same tests fail repeatedly, showing no progress. We limit the loop to five iterations and three repeated failures for our experiments. We observed that the remediation advice often becomes repetitive beyond three to four iterations, yielding diminishing returns on further attempts to fix the code.

To conduct our experiments in a cost and compute-efficient manner, our pipeline executes in the following fashion:

- **Solved without tests:** The first solution is attempted with the problem statement alone and verified for correctness
- **Needs tests:** If the problem statement alone fails to generate a valid solution, the pipeline is rerun, supplying all public tests, along with the problem statement. The newly generated code is then verified against all tests
- **Needs remediation:** If the previous step results in an invalid solution, the failure data is collected from the verifier and the remediation loop is executed until a correct solution is generated
- **Unsolved:** If the remediation loop exits without generating a valid solution, the problem is considered unsolvable by *TGen*

We provide a replication package [1], which includes output and runtime details from *TGen* for all cases we experimented with and code scripts for the experiments. Output

[1] https://osf.io/e3jy6/?view_only=bc67e33bebd3435abf5537d56767401d

from LLMs is often unpredictable. We use a seed value of "1106" and a temperature of 0 in our experiments to improve the replicability of our results.

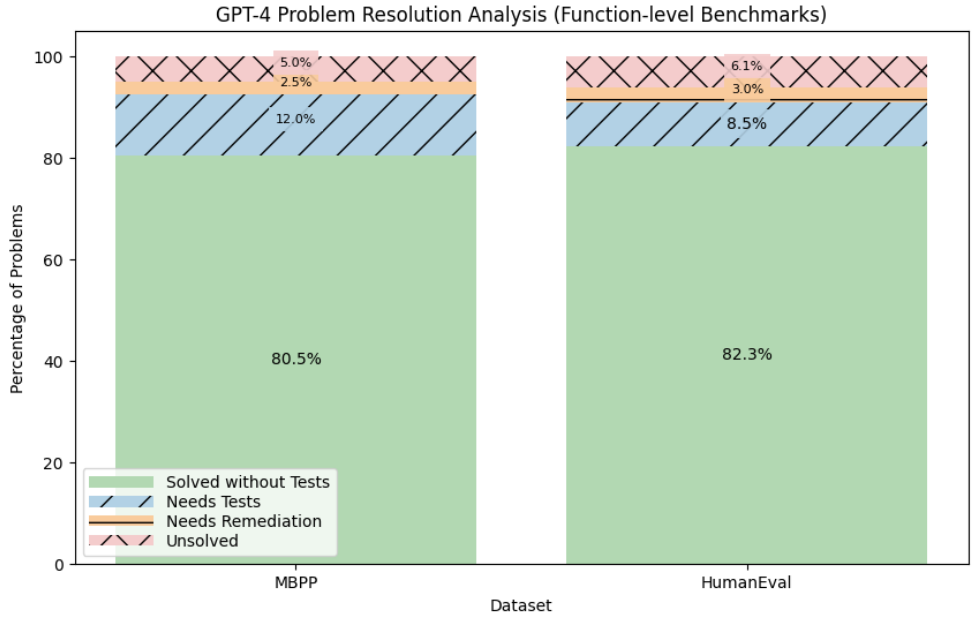


Figure 2.4: GPT4: Problems resolved and strategies used for Function-level code generation benchmarks

2.4 Impact of TDD in Code Generation

In this section, we seek to answer the "IF" question: Do tests help the code generation process? We also manually explore the data to assess the impact of tests and determine which types of problems benefit from tests and which do not. The fraction of correct solutions obtained following the aforementioned workflow with GPT-4 Turbo version 1106 as the LLM for the two function-level benchmark datasets MBPP and HumanEval can be seen in Figure 2.4. RQs 1 through 4 break down these results further. Satisfying public tests is considered the correctness criterion in RQs 1, 2 and 3. RQ4 dives into the notion of private tests and uncaptured requirements. We also manually investigate the data to highlight key details and insights. Please note that we refer to specific cases within the dataset using the following notation "Dataset/Key" where the key is a unique identifier for the problem within the corresponding dataset.

2.4.1 How good is basic code generation?

GPT-4, being one of the most powerful LLMs available today, does reasonably well solving 80.5% of MBPP problems and an even higher 82.3% of HumanEval tasks, as shown in Figure 2.4, both of which consist of function-level benchmarks. Section 2.5.1 discusses file-level generation, which is inherently more challenging, and also how using another LLM affects these results. These numbers are slightly lower than have been reported for GPT-4 Turbo v1106 in popular LLM coding benchmark leaderboards [38] by about 5%. The drop in performance might be attributed to the removal of sample test cases if present in the problem statement for this experiment and differences in the prompt and model parameters like seed and temperature.

2.4.2 Does providing test cases improve code generation?

For MBPP and HumanEval, including tests contributes to solving an additional 12.0% and 8.5% of problems, respectively, showcasing that tests as part of the input help the LLM reach the correct solution. These percentages correspond to an additional 51 problems in the MBPP dataset and 15 additional problems in the HumanEval dataset. Manually inspecting the results of the evaluation with and without tests, several key insights emerged:

- **Problems where the initial implementation had correct logic but incorrect function signatures**: benefited from tests as they helped identify mismatches and ensure adherence to the required function signatures (e.g., MBPP/6, MBPP/90, MBPP/233).
- **Problems involving mathematical formulas or geometric calculations** saw improvements as tests helped catch errors in formula application and ensured the correct implementation of mathematical logic (e.g., MBPP/14, MBPP/430).
- **Problems requiring precise string manipulations or regular expressions** benefited from tests highlighting edge cases and incorrect pattern matching, leading to more robust solutions (e.g., MBPP/16, HumanEval/48).
- **Problems involving complex data structures** such as nested dictionaries or tuples. Tests helped ensure that the functions correctly handled various input formats and edge cases. (e.g., MBPP/106, MBPP/299)

- **Edge Case Handling**: Many problems initially failed due to unhandled edge cases. Tests were instrumental in identifying these edge cases, prompting the system to refine its logic to handle all possible scenarios (e.g., MBPP/170, HumanEval/5).
- **Problems requiring specific algorithmic approaches**, such as finding the maximum difference or sorting criteria, benefited from tests that guided the system to correct logical flaws and align with the problem requirements (e.g., MBPP/63, HumanEval/127).

LLMs seem to be able to improve their understanding of the requirements presented in the problem statement using the provided tests. The tests also seem to have played a significant role in validating mathematical logic, ensuring the correct application of formulas to produce results indicated by the tests. Tests also seem to prompt the system to refine its logic to accommodate a broader range of inputs and edge cases. Consistency in data types, especially in output formats, was another area where tests highlighted discrepancies. In some cases, adding tests led to simplifying logic, replacing initial complex approaches presumably derived from misunderstanding the statement with more straightforward solutions.

2.4.3 Can failed tests help LLMs fix their mistakes?

The remediation loop's role as a corrective mechanism can also be observed, with further improvements noted in problem-solving. For MBPP and HumanEval, the loop seems to effectively refine the code, with an additional 2.8% and 3% of problems solved, respectively. This corresponds to the correct code for 21 additional problems for MBPP, out of which 11 were remediated in the first attempt, seven on the second, two on the third and one on the fourth. For the HumanEval dataset, nine additional problems were solved correctly, of which seven were remediated on the first attempt and one each was remediated on the second and third attempts. It should be noted that in our evaluation pipeline, we allowed for up to five remediation attempts. Although a few cases utilized all five attempts, none were solved. Manually inspecting the data, we find the following cases that are benefited by remediation:

- **Edge Case Handling**: Many problems initially failed due to inadequate handling of edge cases. Remediation loops allowed the system to incorporate specific checks for these cases, leading to correct code generation (e.g., MBPP/92, MBPP/113, MBPP/137)

- **Issues related to data type handling and logical errors** were effectively addressed through remediation loops (e.g., MBPP/115, HumanEval/22)
- **Problems requiring precise mathematical computations** saw significant improvements through remediation. Initial attempts often failed due to issues like integer vs. floating-point division, but remediation advice helped correct these errors (e.g., MBPP/300, HumanEval/77)
- **Problems involving text manipulation**, such as punctuation handling, benefited greatly from remediation loops. Initial attempts often failed due to improper handling of punctuation, but iterative refinements led to progressively better solutions (e.g., MBPP/7)
- **Problems requiring multiple remediation attempts** highlight the importance of iterative refinement. Errors with boundary checks, simple arithmetic corrections, or basic input validation were often resolved through iterative remediation. Each attempt built on the previous one, incorporating new insights and corrections until the correct solution was achieved (e.g., MBPP/160, HumanEval/57)

The general trajectory from initial to final remediation attempts involved a progressive refinement process. In most cases, the first remediation led to basic fixes and the handling of straightforward errors highlighted by the crashes. This often addressed the most obvious errors, such as missing imports or missing type conversions. In the case of a second remediation attempt, more detailed adjustments were involved, addressing deeper logical issues and additional edge cases. A few examples of cases remediation attempts to fix are adjusting modulo operations to map to the correct character range, correcting base cases and loop logic in recursive functions, and handling edge cases like empty arrays or specific input formats.

The remediation agent did not have access to the history to prevent the context length of the models from being exceeded during the evaluation, which means the LLM could not review past fixes. There were cases where the remediation advice misled the generator and often failed to fully implement the remediation advice, leading to repeated errors. We ignore cases like MBPP/462 and HumanEval/145, which occurred due to verifier and environment-related issues, as these are limitations of the evaluator as implemented presently. An interesting question is what kind of problems remain unsolved, manually inspecting we find the following:

- **Misunderstanding core logic or requirements** often led to persistent errors

across multiple remediation attempts (e.g., MBPP/83, HumanEval/38, MBPP/138, HumanEval/83, MBPP/765).

- **Handling specific data structures** like empty arrays, negative numbers, nested lists, and tuples remained challenging despite multiple iterations (e.g., MBPP/119, MBPP/559, MBPP/431, MBPP/630).
- **Logical errors and inadequate adaptation of remediation advice** were common. Despite following remediation advice, the system consistently produced incorrect logic in cases such as rotating binary strings, renaming functions, and counting and pairing differing bits often led to repeated mistakes (e.g., MBPP/109, MBPP/468, MBPP/126, MBPP/595).
- **Misinterpreting input/output formats and complex cases** involving multiple conditions or constraints frequently caused failures. These included handling nested list inputs and incorrectly handling the input format for date validation (e.g., MBPP/278, MBPP/427, MBPP/102, MBPP/581).
- **Performance bottlenecks and inefficiencies** were encountered, particularly with large inputs and complex logic, leading to timeouts and overly complex solutions (e.g., MBPP/765, HumanEval/63).

2.4.4 Do the generated solutions only satisfy the supplied tests or even unseen tests?

Table 2.1: Impact of private tests on GPT-4 based TGen results with improvement offered by tests after validation highlighted

	Function-Level				**File-Level**	
Category	**MBPP (#399)**		**HumanEval (#164)**		**CodeChef (#1100)**	
Solved without tests	69.67%		78.66%		23.00%	
Needs tests	82.45%	**(+12.78%)**	87.81%	**(+9.15%)**	26.09%	**(+3.09%)**
Needs remediation	87.71%	**(+5.26%)**	93.30%	**(+5.49%)**	30.27%	**(+4.18%)**
Unsolved	12.28%		6.71%		69.73%	
Improvement using tests		**18.04%**		**14.64%**		**7.27%**

An interesting question in our study is whether the generated solutions are tailored merely to pass the provided tests or if they are robust enough to succeed against unseen tests as well. This question becomes particularly pertinent when considering the potential

for omitted requirements in the supplied test cases. While TDD advocates for the use of tests as the primary specifications, the risk of overlooking certain criteria due to suboptimal test quality cannot be ignored.

For the MBPP and HumanEval datasets, we employed their EvalPlus [38] enriched variants with 35x and 80x more tests, respectively, offering a rigorous assessment platform. EvalPlus seeds its test generator with high-quality inputs and extends them through type-aware mutation to establish a robust framework for evaluating the true correctness of LLM-synthesized code.

Our extended evaluation of the generated code utilizing private tests, summarized in the first two columns of Table 2.1, indicates that including public tests still results in improved performance across benchmarks. Even though there is a reduction in the number of problems solved corresponding to each category once verified, the addition of public tests yielded a performance increase of 12.78% for MBPP and 9.15% for HumanEval. Further application of remediation loops led to additional improvements: 5.26% for MBPP and 5.49% for HumanEval. This demonstrates that even without the LLM having access to the comprehensive test suites provided by EvalPlus, including some tests, it aids in enhancing the correctness of the generated solutions over not using tests. We discuss the results presented in the last column in Section 2.5.1 that discusses file-level generation in more detail.

The occurrence of failures in some solutions when subjected to private tests highlights a crucial point: the more exhaustive the test suites are, the better the results tend to be. Our findings reveal that solutions developed with test information fare well against private tests, underscoring the benefits of integrating tests in the code generation process to bolster robustness. This effectiveness points to a nuanced balance: while incorporating tests, even partially comprehensive ones, bolsters solution robustness, the ultimate success hinges on the quality of these tests. Having a wide variety of tests ensures solutions are not just tailored to meet the explicit criteria in the problem statement, but are also able to capture implicit requirements.

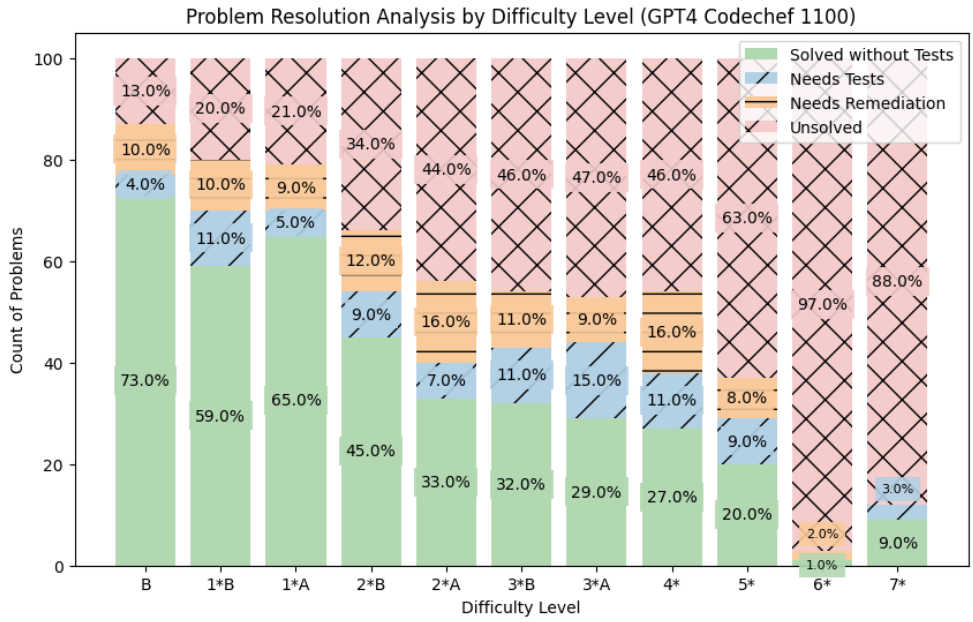


Figure 2.5: Problems resolved and strategies used compared across difficulty levels using GPT4

2.5 Parameters affecting TDD

In this section, we explore the "HOW" question: how do different parameters impact the effectiveness of tests in code generation? We aim to understand the influence of problem difficulty, the number of tests, and the LLM itself. RQs 5 through 7 address these aspects in detail. RQ5 examines how problem difficulty affects code generation results. RQ6 investigates the optimal number of tests needed for effective code generation. RQ7 evaluates whether the results hold when using an open model like Llama 3 as the reasoning engine. This section builds on the findings from Section 2.4, providing a deeper understanding of the factors that influence the success of integrating TDD with LLM-based code generation.

2.5.1 How does problem difficulty affect these results?

Data collection

We curated a problem set of coding problems from CodeChef and their corresponding human solutions. CodeChef assigns each problem on its platform a difficulty score and classifies ranges of difficulties into 11 buckets, as depicted in Table 2.2. To ensure that our study's problem set has a good distribution with respect to difficulty, we fetched the 100 most popular problems from each difficulty level. Here, popularity refers to the number of accepted solution attempts that existed when the data was scraped (November 2023). The curated multi-level dataset of 1100 problems represents a spectrum of real-world coding challenges of varied difficulties that require full program synthesis.

Table 2.2: Difficulty Levels of Selected CodeChef Problems

Level	Range	Count
Beginner	0 - 999	100
1* Beginner	1000 - 1199	100
1* Advanced	1200 - 1399	100
2* Beginner	1400 - 1499	100
2* Advanced	1500 - 1599	100
3* Beginner	1600 - 1699	100
3* Advanced	1700 - 1799	100
4*	1800 - 1999	100
5*	2000 - 2199	100
6*	2200 - 2499	100
7*	2500 - 5000	100
		1100

Results

A breakup of the problems and solution strategies is shown in Figure 2.5. The data shown can be summarized as 35.72% (393) problems solved without providing tests, 7.72% (85) additional problems solved with public tests being supplied to the LLM, and another 9.36% (103) problems solved using the LLM to look into validation issues and remediating the code. In contrast to function-level generation, where we can expect the function signature and input and output schemes to be straightforward, file-level generation, as required for

the curated CodeChef dataset, allows for more flexibility in problem-solving approaches. This flexibility can make CodeChef problems harder to solve, even without considering task complexity. Performance on the CodeChef dataset is markedly lower, with success rates sharply decreasing as problem difficulty escalates. Despite this, 35.72% of these problems were solved correctly with just the problem statement. This highlights the challenges associated with these problems as compared to function-level generation datasets we have explored so far. We need to parse the input correctly and provide the output to the console in the expected format, apart from the logical complexity of the tasks.

Providing test cases led to a consistent increase in the number of problems solved across difficulty levels, increasing the total number of problems solved by 7.72%. This underscores the importance of test cases in aiding code generation across all difficulty levels.

Remediation loops further enhance performance in the CodeChef dataset, with an improvement of 9.36%. This indicates the LLM's capacity for self-improvement and its ability to tackle increasingly sophisticated challenges, particularly for medium-difficulty problems. The complexities associated with correctly handling input and output operations likely contribute to the higher impact of remediation on full program synthesis required for CodeChef tasks. Most of the CodeChef problems (47.18% or 519 problems) remained unsolved in our testing configurations. Allowing more remediation attempts could potentially improve these numbers, but our experiments show diminishing returns with increased iterations.

We consider private tests for additional validation to gauge the impact of supplied tests. These tests, not disclosed beforehand, are used by grading systems on platforms like CodeChef to determine correctness. We leveraged these private test suites to evaluate the generated solutions by evaluating the solutions on the platform itself. The results can be seen in the last column of Table 2.1. We note that we still observe a 7.27% improvement in the correctness of solutions across the dataset comprising 1100 problems. It should be noted that as shown in Table 2.3, most cases in CodeChef only had one sample test available publicly. Further, full program generation is inherently more challenging as it introduces more points for failure and has a verbose problem statement where the objective is not laid out explicitly.

Our analysis, as illustrated in Figure 2.6, delves into the relationship between problem difficulty and the success rate of generated solutions. The data reveals a clear trend: simpler problems tend to be solved effectively using tests alone, while more complex problems often require additional remediation for successful resolution. Notably, the most challenging problems remained unsolved, indicating a direct correlation between problem difficulty and the likelihood of resolution through our current methodological framework. We see that a much less expensive operation of supplying tests quite significantly impacts the

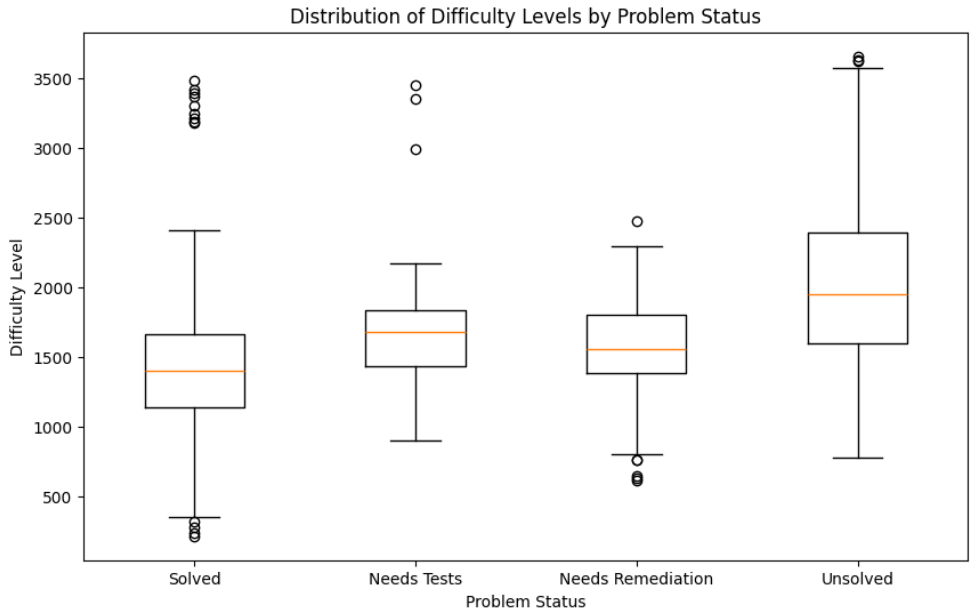


Figure 2.6: Distribution of difficulty levels associated with problems solved by each strategy

fraction of solved problems.

Lastly, the discrepancy observed between the success of solutions on public tests versus their failure on CodeChef's private tests raises important concerns. Our analysis of difficulty scores for problems that failed only on private tests yielded an average score of 1627.75, situating these problems within the medium difficulty range. The performance of GPT-4 generated solutions on these tests, where 16 were partially accepted, two caused judge errors, six resulted in runtime errors, and 25 exceeded time limits—underscores the necessity of comprehensive testing. These findings highlight the importance of developing more robust tests that can anticipate and mitigate such issues despite efficiency not being the primary focus of this investigation.

Table 2.3: Overview of the number of public tests in each benchmark

Benchmark	**Avg.**	**Medium**	**Min.**	**Max.**	**#Tasks**
MBPP	3.10	3.00	2.00	7.00	399
HumanEval	8.05	7.00	1.00	26.00	164
CodeChef	0.94	1.00	0.00	4.00	1100

2.5.2 How many tests is good enough?

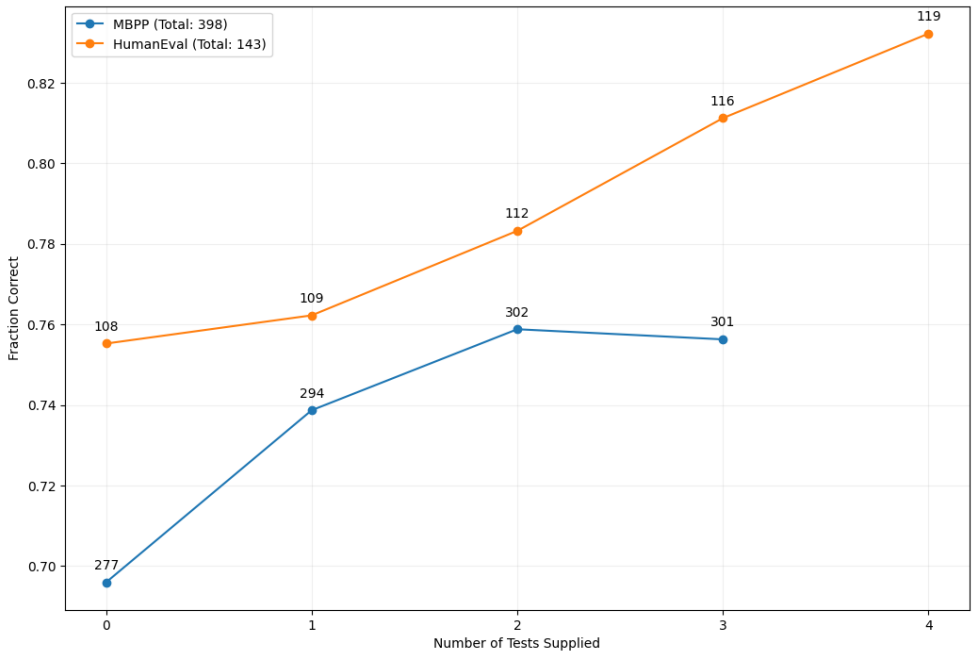


Figure 2.7: Correctness with increasing number of tests supplied

The number of test cases available per problem varies widely across both function-level generation benchmarks as shown in Table 2.3. Considering the scope of our study, we choose to include only the human-written tests in these datasets and not those generated by EvalPlus. To explore the impact of the number of these tests supplied to the LLM, we work with a subset of both datasets so that as we vary the number of tests, we are still commenting on the correctness observed for the same set of problems. The cutoffs are picked so that we have enough cases to analyze while not dropping too many problems from our analysis. For MBPP, we look at the subset of problems with at least three tests, resulting in 398 problems, and only one case is removed. For HumanEval, we look for cases with at least 4 tests, leaving us with 143 problems out of the initial 164. Since most of the tasks in the CodeChef dataset only contains one test we refrain from including it in this analysis.

We run the evaluation pipeline without remediation for each case in these subsets and repeat the experiment while increasing the number of tests supplied. The variation in

the fraction of correct solutions generated in each case is shown in Figure 2.7. We see that the impact varies across the datasets, but we see an upward trend in both cases as more tests are added. However, we do not observe a plateau in the case of HumanEval, which might be attributed to the simpler nature of problems in MBPP, which seem to suffer more from vagueness than complexity. Also, we note from our experiments that as we continue increasing the number of tests, in some cases, we end up with incorrect code being generated that had been correctly solved before. This could be because of the "lost in the middle" problem reported in the literature, where the LLM starts ignoring context in the middle as more and more information is supplied to the model.

Additionally, we look at the sample solution to each of the problems included in the datasets and use it to compute line coverage contributed by each test case. We note that 75% of the cases in HumanEval and 92.4% of cases in MBPP achieve 100% coverage with the addition of the first case itself. While high coverage is generally desirable, it's important to recognize that coverage alone doesn't guarantee test quality or comprehensiveness. Tests achieving full coverage might still miss important edge cases or fail to capture the full functionality of the code. This could be another reason why the curve for MBPP is starting to decline with just three tests. We need a better dataset with more human-written test cases for more rigorous analysis, but we hypothesize, based on our results, that adding more diverse test cases aids in the code generation process. It's important to note that the "quality" of tests plays a crucial role in evaluation. While we selected benchmarks containing human-written tests to include a wide variety of tests, good and bad, this approach has limitations. For instance, EvalPlus [38] has highlighted test insufficiency in these datasets, leading to a reduction in pass@k by 19.3-28.9% when using the refined test suite. This raises an interesting question about the potential impact of using lower-quality tests, such as those automatically generated. However, there is currently no clear consensus on what constitutes 'low quality' tests or a standardized dataset to evaluate them. Future work could explore the comparison between generated tests and human-written tests, as well as the impact of test quality on code generation performance.

2.5.3 Do these results hold with an open model like Llama 3?

Apart from GPT-4 Turbo, we also experimented with GPT-3.5 Turbo v1106 and observed similar trends. However, in our discussion so far, we presented only results from GPT-4, as it represents one of the most powerful LLMs available for public use. An interesting question is whether the benefits of TDD can be seen outside OpenAI's GPT family of models. To address this question, we pick Meta Llama 3 70B Instruct, another popular model that shows promise for code generation, ranking among the best models in code

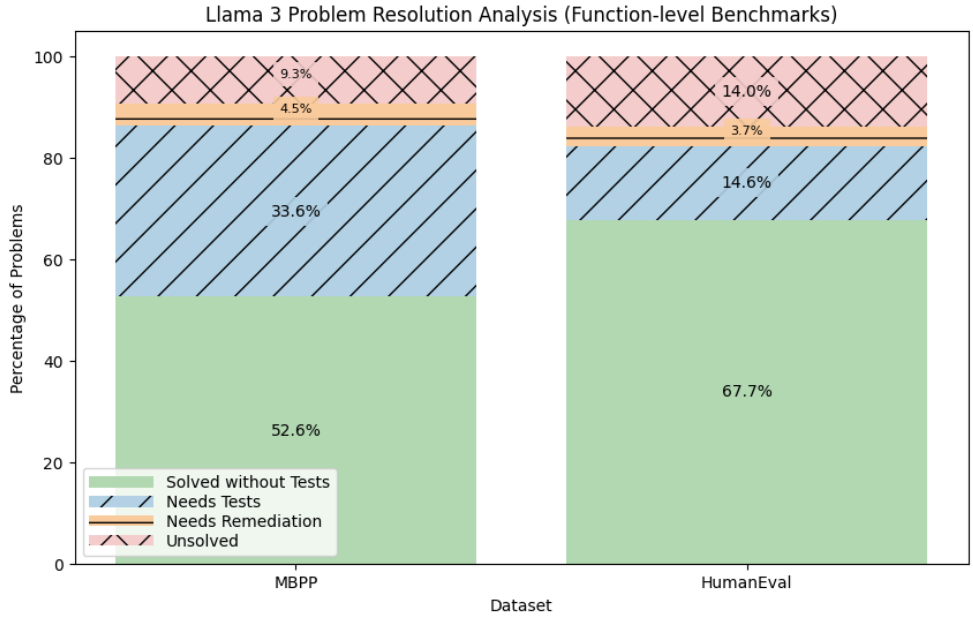


Figure 2.8: Llama 3: Problems resolved and strategies used for Function-level code generation benchmarks

[17, 38] and non-code LLM benchmark leaderboards [13]. The output from *TGen* for GPT-3.5 Turbo is also included in our replication package.

The results observed by replacing GPT-4 with Llama 3 in the evaluation pipeline can be seen in Figure 2.8. We see that the trends observed with proprietary models extend to open models. Code generation using Llama 3 benefits from the additional tests and remediation setup provided by *TGen*, and we even see a greater impact than we observed with GPT-4. As compared to GPT-4, which had over 80% of the problems being solved in the baseline, we see a lower performance of 52.6% and 67.7% in MBPP and HumanEval benchmarks, respectively. Despite the worse code generation performance out of the box, we note that the addition of tests and remediation both significantly improve the results.

The final measure of improvements observed after evaluating the correctness of generated solutions using the EvalPlus can be seen in Table 2.4. We see that after validating the solutions on private tests, we see a 38.6% improvement in MBPP and a 21.95% improvement in HumanEval as compared to their respective baselines, which is almost double the improvement we saw in the case of GPT-4 shown in Table 2.1.

Table 2.4: Impact of private tests on Llama 3 based TGen results with improvement offered by tests after validation highlighted

Category	**MBPP(#399)**		**HumanEval(#164)**	
Solved without tests	46.37%		62.20%	
Needs tests	75.94%	**(+29.57%)**	75.61%	**(+13.41%)**
Needs remediation	84.96%	**(+9.02%)**	84.15%	**(+8.54%)**
Unsolved	15.04%		15.85%	
Improvement		**38.60%**		**21.95%**

2.6 Related Work

Evaluating code generation performance is an area that is being actively explored. Liu et al.'s EvalPlus [38] proposes rigorous benchmark enhancements for evaluating LLM-generated code's functional correctness, highlighting the insufficiency of current benchmarks in depth and breadth. Ding et al. present CrossCodeEval [15], which challenges LLMs with multilingual, cross-file context code completion. Jimenez et al.'s SWE-bench [30] uses real-world GitHub issues to test LLMs, with results showing current models solve only a fraction of problems.

Improving code generation using tests has emerged as a popular trend among other works that have explored other methods, such as improving the reasoning ability of LLMs by using diverse prompts, voting, and other reasoning verification strategies apart from tests [36]. Wang et al. employ test execution feedback while training the model to enable the model itself to discern incorrect code [56]. Chen et al.'s Codex [11] demonstrates the limitations of single-sample code generation and suggests that one could similarly exploit multiple test iterations to refine code quality. There have also been attempts at making compilable code by incorporating feedback from compilers [57]. Shinn et al.'s Reflexion [51] method allows agents to learn from past mistakes, emphasizing the need for approaches that can iterate over code generation and testing for improved decision-making in software development. Recent agentic systems like Yuntong et al.'s AutoCodeRover have also explored utilizing test cases for spectrum-based fault localization when retrieving context for code generation [64]. However, most of these explore the problem in a program repair setting.

Generating tests is another way tests for code generation are being explored recently [49]. Lever [44] attempts to rank generations based on trained verifiers derived from the prompt. A comprehensive exploration is presented by Codet, where they assess the quality of generated test cases and their impact [10]. Lahiri et al. [32] explore the use of

test cases for user intent formalization, but they rely on developer feedback for mutating and ranking test and code suggestions effectively using generated tests. Fakhoury et al. build upon this by leveraging user feedback through LLM-generated tests to improve the correctness of generated code presented as Ticoder [20]. While Lahiri et al. and Fakhoury et al. present 'test-driven' methodologies, they focus on intent formalization during which both tests and code are generated iteratively, adapting to user feedback. In contrast, our approach aligns more closely with traditional TDD principles by starting with user-provided tests that formalize initial requirements. Their user study [20] highlights the correct evaluation of AI-generated code for entry-level tasks but, also shows participants failing to distinguish wrong tests surfaced by the recommender, which is concerning. We thus refrain from leveraging user feedback during generation or commenting on the effort and accuracy tradeoffs involved. Similarly, 3DGen [19] uses LLMs to formalize user intent into test cases, but emphasizes symbolic techniques for exhaustive test generation and formal verification. However, both methods still focus on generating tests rather than starting from user-written test cases. A key gap in the current literature is a lack of user studies evaluating the impact and effectiveness of such approaches in the long run.

The aforementioned works showcase the potential for using verification and validation techniques to improve the performance of LLMs across different complexity levels and tasks. Further, they also hint at the efficacy of using LLMs for program repair through feedback-based techniques. None of these studies, however, answers key questions surrounding the impact of including human-written test information and how these mechanisms can be employed in a test-driven workflow. We wish to bridge this gap and structure our experiments to explore how to use tests efficiently and how test-driven development principles can be used to improve code generation.

2.7 Threats to Validity

Prompt engineering, while a powerful tool to guide LLMs, is also subject to limitations. Poorly designed prompts can lead to sub-optimal results, skewing the model's results and potentially misleading it. To minimize this, we keep the prompt structure consistent across our experiments and vary only the parameters under consideration. Using popular benchmark datasets, such as MBPP and HumanEval, has inherent limitations. These benchmarks do not capture the full complexity and variability of real-world programming problems but help shed light on the impact of tests. We also use our own dataset of problems with a clear notion of difficulty, but this might not be representative of the difficulty of tasks in software engineering.

A significant threat to validity in evaluating LLMs on benchmark datasets is the potential for data leakage. Given the closed nature of training data for models like GPT-4, we cannot guarantee the absence of overlap between the benchmarks and the model's training data. This is a common concern for all evaluation benchmarks used with LLMs. To partially mitigate this, we curated an additional dataset from the competitive programming website CodeChef. While this doesn't completely eliminate the possibility of leakage, it provides a different source of problems that may be less likely to have been included in the model's training data. It's important to note that our study focuses on the incremental improvements achieved by incorporating tests, which we believe provides valuable insights into LLM capabilities regardless of the exact composition of their training data.

This study focused on human-written tests from established benchmarks or competitive programming websites to include a wide variety of tests. However, these tests might not be representative of those found in real-world projects. Future work should investigate the quality of the test suite and its impact on code generation. Further, our assumption that test runs serve as ground truths for the corresponding problems may not always hold, particularly for non-deterministic problems. We excluded such problems from our curated dataset and filtered out errors through manual inspection. Recent literature also suggests that the tests used in widely adopted benchmarks may not be robust enough for evaluating code generation. To address this, we used the refined versions of the HumanEval and MBPP datasets curated by EvalPlus [38], which include additional tests for more thorough evaluation. The performance of TDD with LLMs might vary with different models, architectures, or training data. Further, the inherent variability in LLM outputs can introduce inconsistencies. We employ a fixed seed value and temperature setting to improve the reproducibility of our results, but some degree of unpredictability remains.

2.8 Conclusion

In the rapidly evolving landscape of LLMs for code, where these models are being increasingly used for code generation and remediation, embracing TDD emerges as a strategic paradigm shift. TDD enables us to use tests in the generation phase and for validation. By incorporating test cases, and employing remediation loops, we are able to solve complex problems that the LLM cannot solve normally. Using GPT-4, we observe a significant increase in the correctness of the code generated. We find that this improvement is more pronounced for less performant models that can only solve a much lower fraction of problems initially. We also highlight the cases in which these approaches work well and the kind of tasks that present-day LLMs struggle with. Our experiments with GPT-4 and

Llama 3 show that merely providing the LLM with tests during generation improves correctness by 9.15 to 29.57%. Adding remediation loops results in an additional gain of 5.26 to 9.02% across popular function-level benchmark datasets after rigorous evaluation with private tests. We also find that the proposed approach improves correctness by 7.27% on our dataset of 1100 file-level generation tasks where GPT-4 performs poorly and struggles with more difficult problems. Our findings validate the impact of tests, and we therefore advocate for the widespread adoption of test-driven methodologies to maximize the benefits of LLMs in code generation. While our results demonstrate the benefits of TDD in LLM-based code generation, future work should focus on quantifying the tradeoffs between the effort required to write tests and the resulting increase in accuracy. A user study could provide valuable insights into how this approach affects overall development time and code quality in real-world scenarios.

Chapter 3

Evaluating LLM-based test generators on buggy code

3.1 Introduction

Software testing remains a critical yet time-consuming aspect of software development. With the advent of Large Language Models (LLMs), there has been a significant shift towards automating various software engineering tasks, including test case generation. Tools like GitHub Copilot, which serves millions of developers [16], are increasingly being adopted for automated test generation. This trend is further visible in recent research efforts [14, 18, 46] and commercial tools like CodiumAI's CoverAgent [47]. While there are various approaches to automated test generation, including fuzzing and crash-based testing, our study focuses on LLM-based tools that generate complete test cases with oracles and assertions.

LLM-based test generation tools [46, 47] promise to improve code coverage and generate tests by observing the existing implementation and filtering out or attempting to "fix" failing tests. Such an approach raises critical concerns about their effectiveness in achieving the primary goal of software testing: detecting bugs. This challenge aligns with the broader "test oracle problem", where distinguishing correct from faulty behaviour requires reliable oracles [5].

Unlike Test-Driven Development (TDD), which generates tests based on requirements before implementation, these LLM-based tools generate tests based on the code itself. It should be noted that this reliance on code-driven test generation fundamentally differs

from requirement-driven approaches like TDD, potentially limiting their usefulness to just detecting bugs. This study explores how current LLM-based test generation tools handle buggy implementations, highlighting the risks of relying on such automatically generated tests. We present empirical evidence demonstrating how these tools can inadvertently validate incorrect behaviour through their test generation and filtering mechanisms. Our findings suggest that current LLM-based test generation approaches may be fundamentally misaligned with the core objective of software testing (revealing bugs), pointing to a critical gap that must be addressed before these tools can be reliably integrated into software testing practices.

3.2 Related Work

Automated test generation has been extensively studied, with approaches ranging from specification-based to search-based software testing (SBST) [21]. For Python specifically, Pynguin [39] employs SBST techniques, using genetic algorithms to evolve test cases for improved coverage. Recent work has enhanced these traditional approaches with LLMs, as demonstrated by CodaMosa [34], which uses LLMs to overcome stalled search states in Pynguin's genetic algorithm. It however explicitly refrains from using LLMs for generating oracles and only uses them to generate inputs. While Pynguin itself can generate oracles by running the program on the input, this approach relies solely on runtime behaviour rather than leveraging LLMs for oracle generation.

The success of LLMs in Software Engineering has led to the development of several specialized test-generation tools. MuTAP [14] focuses on mutation testing, iteratively prompting LLMs to generate assertions for surviving mutants. MuTAP suffers from the same problems highlighted for other tools in this study as it follows a similar filtering mechanism. However, MuTAP is not designed to be easily executable on an arbitrary codebase hence we instead include a much more recent research tool Coverup [46]. Tools like Codium CoverAgent and CoverUp take different approaches to coverage-driven test generation, with CoverUp specifically using coverage measurements to guide LLM prompting and maintaining an iterative dialogue for test refinement. These tools represent a shift from traditional automated testing approaches to LLM-assisted generation, though their effectiveness and reliability remain areas of active research.

Recent work by Huang et al. raises critical concerns about LLM-based test generation, showing that incorrect code can significantly mislead LLMs in generating reliable and bug-revealing tests [28]. Our work builds upon these findings by specifically examining if and

how LLM-based test generation tools might not only fail to detect bugs, but potentially validate faulty code through their design.

3.3 Methodology

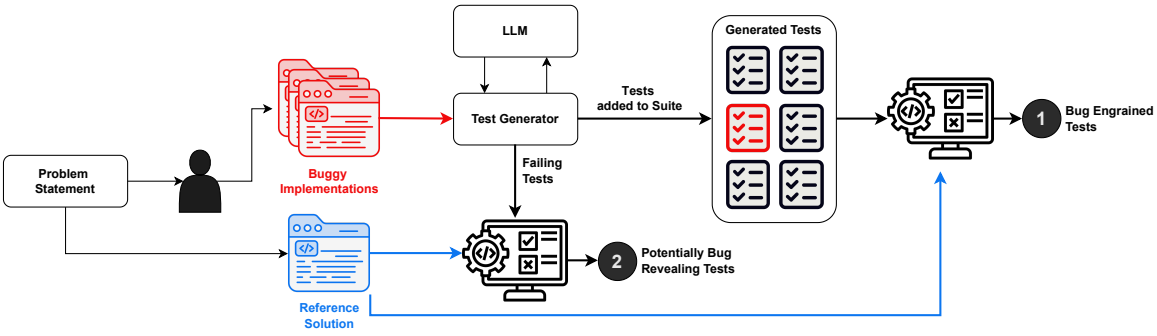


Figure 3.1: Overview of the evaluation pipeline

We design our experiments, such that we can analyze both the passing and failing tests generated by recent LLM-based test generation tools. We aim to understand how different approaches, adopted by these tools affect the generated test suite and if these tests would validate bugs or unintended behaviour when given potentially buggy code.

3.3.1 Dataset

For our study, we utilize the Refactory dataset [27], a benchmark for Python bug repairing. This dataset is particularly suitable for our exploration as it contains real-world buggy code snippets generated by humans. We chose to focus on Python as it is widely used across various domains and as a best-case scenario as LLMs have been shown to demonstrate better performance in synthesizing Python code compared to other languages [14]. Refactory comprises submissions from 361 undergraduate students for 5 programming assignments in an introductory Python course at the National University of Singapore. It contains 2,442 correct and 1,783 buggy program attempts. All programs in Refactory are self-contained and mostly involve the creation of a single function, allowing test generation tools to focus on these alone, reducing factors that could influence the results we highlight in this study.

This dataset allows us to evaluate test case generation on actual human-written buggy implementations and a reference solution based on the same requirement (problem statement here) and compare them across different tools, reflecting the reality that code is often imperfect and prone to bugs. This dataset allows us to observe and comment on the performance of these tools in a controlled yet realistic setting, without the added complexity of seeded bugs in large-scale projects.

To prepare the dataset for our experiments, we performed the following steps:

1. We retained cases with at least one passing and one failing test to focus on implementations that are executable and partially meet the requirements of the original test suite.

2. The existing input-output tests in the dataset were converted to the format expected by the unit testing library PyTest to ensure compatibility with the tools under evaluation, which require a configured testing framework for execution.

3. To keep our analysis tractable, we removed duplicates and selected a subset of cases where the test coverage, based only on passing tests, was less than 100%. This also ensures that the selected cases provide opportunities for the tools to improve test coverage.

Due to the complexity introduced by multiple target functions, we excluded assignment 2 from our analysis which requires 3 functions to be implemented, unlike the other problems. We also retained details about which tests passed and failed for each buggy implementation to aid with manual inspection of the results. After applying these filters, our final dataset consisted of 287 data samples across 4 questions, which we used to explore our research questions.

3.3.2 Test Generation Tools

For our study, we picked three prominent LLM-based test generation tools: GitHub Copilot, Codium CoverAgent, and CoverUp that represent different approaches to automated test generation using LLMs that are available to Python developers today. We go over other past approaches in Section 3.2.

GitHub Copilot

GitHub Copilot is a widely adopted AI-powered code completion and generation tool that integrates with various development environments. While primarily known for code generation, Copilot also offers capabilities for generating unit tests. As of writing it operates on a pure generation model, suggesting test cases based on the context of the code in the editor or highlighted snippets. Copilot can propose input parameters, expected outputs, and assertions for functions, potentially aiding in creating test cases for edge cases and boundary conditions [18].

However, it's important to note that Copilot generates tests without executing them or verifying their correctness against the implementation. While potentially time-saving, this approach may lead to the generation of tests that don't compile, fail to run, or don't increase code coverage.

Codium CoverAgent

Codium CoverAgent (now renamed to qodo-cover) is an open-source implementation[1] inspired by Meta's TestGen-LLM research [2] which adopts an "Assured LLM-based Software Engineering" approach aiming to augment existing test suites with additional test cases while ensuring no regression. The tool employs an iterative process:

a) Measuring existing test suite coverage (if any)

b) Generate multiple test cases

c) Filter out tests that don't compile or run

d) Discard tests that don't pass on the current code

e) Retain only tests that increase code coverage

f) Repeat the process while disincentivizing the creation of tests like the failing ones

This approach attempts to address common issues in LLM-based test generation, such as hallucination and the generation of low-value or non-compiling tests. However, by design, it may inadvertently reinforce existing bugs in the code by only keeping tests that

[1] Qodo offers an enterprise version which we don't have access to

pass on the current, potentially faulty, implementation. Further, the explicit prompt to not generate tests similar to the failing cases might end up with bug-prone regions being ignored.

CoverUp

CoverUp represents a more recent approach in LLM-based test generation, focusing on creating high-coverage regression tests for Python [46]. It employs an Agent [61] style approach that interleaves coverage analysis with LLM interactions. The process involves:

a) Measuring existing test suite coverage (if any)

b) Identifying uncovered code portions

c) Prompt the LLM to generate tests for each uncovered portion

d) Verifying that new tests run and increase coverage

e) Refining prompts based on updated coverage information

f) In case of suite failure, test failure details are used to further engage in dialogue with the LLM

g) Continue onto the next uncovered portion (if any) after the max iterations are reached

CoverUp aims to guide the LLM towards generating tests that specifically target uncovered code, potentially leading to higher overall coverage compared to other approaches. It also provides the LLM with the ability to collect context about different parts of the codebase if it wishes to do so. As compared to CoverAgent, this design is potentially more susceptible to creating bug-engrained tests as it attempts to "repair" failing test suites iteratively.

These tools were chosen for our study due to their distinct approaches to test generation and their potential impact on software testing practices. While each tool claims to improve the efficiency and effectiveness of test creation, our research aims to critically evaluate their performance, particularly in the context of potentially buggy implementations. By examining these tools, we seek to understand the implications of relying on LLM-generated tests for bug detection and code validation.

3.3.3 Evaluation Pipeline

Figure 3.1 provides an overview of the evaluation pipeline adopted for this study. We utilize the 287 buggy samples filtered from the Refactory dataset as described in Section 3.3.1 and also collect the reference solutions associated with each of the four problem statements. We deliberately utilize the default parameters wherever possible and only make the necessary changes to have the tools work on our extracted samples. This also means we stick with the default LLMs adopted by the tools which at the time of writing is OpenAI's GPT4O (Version 2024-08-06) for CoverAgent and Coverup. The exact LLM used is not very important in our study as we attempt to focus on the design and objectives of pipelines within these tools rather than the power of the LLM itself, using GitHub copilot as sort of a moving baseline being a popular pure generative approach to the problem providing the bare minimum functionality for "one-click" test generation with machinery to collect the required context from the project. We had each test generator write tests from scratch effectively populating an empty test suite. A more detailed breakdown of the automated test generation setups we used in each case is provided below:

GitHub Copilot

We implemented an automated framework to evaluate GitHub Copilot's performance in generating unit tests. Using Playwright, the script programmatically interacts with GitHub Codespaces to ensure a consistent and reproducible process, this also helps us execute Copilot commands as would be used by developers within VSCode without introducing any variability. The framework begins by setting up the environment, including authentication and session persistence, to streamline task execution.

Each task, described in a dataset with function signatures and sample content, is processed iteratively one code snippet at a time, resetting the environment at the end of each iteration. Source and test files are created automatically, and the specific method to be tested is selected within the editor to provide context for Copilot. The Copilot command `'Github Copilot: Generate Tests'` is invoked to trigger test generation. User interactions, such as accepting or applying generated suggestions, are automated to accept all suggestions. Generated test files are then organized into a designated directory for further evaluation.

Codium CoverAgent

Our script iterates through directories containing buggy code samples and their corresponding test files which contain blank templates with a dummy empty test but can also be instantiated with existing tests if required. It invokes CoverAgent using the following command structure:

```
cover-agent --source-file-path [source] --test-file-path [test] --code-
    ↪ coverage-report-path [report] --test-command [pytest] --coverage
    ↪ -type [cobertura] --desired-coverage [100]
```

CoverAgent is thus tasked to generate tests iteratively until the desired code coverage or the maximum number of iterations is reached (defaults to 5). Each iteration filters out tests that fail to compile or run, retaining only those that pass and contribute to increased coverage. We instrument the implementation of CoverAgent such that we gain access to all tests generated and also collect additional information surrounding rejection. Logging is enabled to capture execution details and any errors encountered.

CoverUp

Our automated pipeline integrates CoverUp by processing buggy code samples and preparing a structured environment for its execution. For each code sample, the script creates a dedicated directory structure with separate `src` and `tests` folders, copying the source files and optionally generating blank test templates similar to the setup for CoverAgent.

The tool is executed using the command:

```
coverup --tests-dir [tests] --package-dir [src]
```

CoverUp iteratively measures coverage, identifies uncovered code segments, and generates tests to target those segments. It attempts to cover each uncovered segment It refines the process by analyzing test failures and adjusting its prompts to improve coverage. Success and failures on test suite execution are collected for analysis, and the process continues until the tool exhausts its iterations or achieves the desired coverage.

3.4 Results

Table 3.1 shows the outcomes of evaluating test suites generated by each tool against both the buggy implementations (OG) and reference solutions (REF) in each case. For each

tool, we analyze how the generated test suites behave when executed against these two implementations, capturing four possible outcomes that help us understand if the tests are effectively identifying bugs or inadvertently validating them. These are:

Table 3.1: Evaluation Results of Generated Test Suites

Tool	**OG Failed**		**OG Success**	
	REF Success	**REF Failed**	**REF Success**	**REF Failed**
GitHub Copilot	194	66	15	12
CodiumAI	470*	1296*	116	171
CoverUp	400*	235*	29	62

OG - Original buggy implementation used to generate the suite
REF - Reference Implementation
* These test suites were rejected during the generation process.

- **OG Failed, REF Success**: Tests that fail on the buggy implementation, but pass on the reference solution. These represent potentially valuable tests that correctly identify bugs. For GitHub Copilot, 194 test cases (67.6% of generated tests) fall into this category. However, both CodiumAI and CoverUp reject such tests during their generation process, with 470 and 400 such tests being discarded respectively.
- **OG Failed, REF Failed**: Tests that fail on both implementations. These typically represent poorly generated tests that either don't compile or contain invalid assertions. GitHub Copilot generated 66 such cases (23% of its output), while CodiumAI and CoverUp rejected 1296 and 235 such tests respectively during their filtering process.
- **OG Success, REF Success**: Tests that pass on both implementations. These tests exercise correct behaviour common to both implementations. This category represents the smallest portion for all tools: 15 cases (5.2%) for Copilot, 116 cases (40.4%) for CodiumAI, and 29 cases (31.9%) for CoverUp's suites.
- **OG Success, REF Failed**: The most concerning category, is tests that pass on buggy code but fail on correct implementations. These tests effectively validate bugs by treating incorrect behaviour as expected. We found 12 such cases (4.2%) in Copilot's output, while CodiumAI and CoverUp produced 171 (59.6%) and 62 (68.1%) such cases respectively in their final test suites.

It should be noted when looking at these numbers that in the first 2 cases (with OG Failed), CoverUp and CoverAgent do not generate test suites and reject all tests. We collect these "failed tests" by tapping into the tool's inner workings. Additionally, CoverAgent disincentivizes the creation of tests similar to the failed cases. CoverUp failed to generate a test suite for 196 cases (68.3% of the dataset) as its repair attempts were unsuccessful. This high failure rate, combined with the concerning proportion of tests that validate bugs in the final suites (68.1%), suggests that current approaches to test generation might be fundamentally flawed in their ability to handle buggy implementations and detect bugs effectively.

3.5 Discussion

The results demonstrate a concerning trend: tools that implement filtering mechanisms (CodiumAI and CoverUp) tend to produce final test suites that are more likely to validate bugs compared to pure generation approaches like GitHub Copilot. This is evidenced by the higher percentage of "OG Success, REF Failed" cases in their final outputs, suggesting that their filtering and fixing mechanisms might be counterproductive on real code that tends to have bugs.

Simple experiments can expose fundamental issues with current LLM-based test generation approaches. Consider the example below:

```
# Buggy Implementation
def find_sum(a,b):
    return (a+b+1) # Bug: Adds 1 to sum

# Copilot Generated Test Suite
def test_find_sum(self):
    self.assertEqual(find_sum(1, 2), 4)
    self.assertEqual(find_sum(-1, 1), 1)
    self.assertEqual(find_sum(0, 0), 1)
    self.assertEqual(find_sum(100, 200), 301)
    self.assertEqual(find_sum(-5, -5), -9)
```

In this case, the function incorrectly adds 1 to every sum, yet the generated test suite validates this incorrect behaviour by asserting the buggy outputs as expected results. While GitHub Copilot might generate such tests through pure generation, tools like CodiumAI and CoverUp systematically retain these bug-validating tests while rejecting potentially

valuable tests that could expose the bug. While one might argue that the incorrect behaviour could align with some specific use case, the lack of a docstring or other semantic cues leaves the intended functionality ambiguous. Unlike Copilot, which can be directed to write specific tests, tools like CodiumAI and CoverUp infer requirements from the code itself and make their sole objective to increase coverage. This reliance on inferred intent introduces challenges when bugs are ingrained in the code. Such bugs may distort how developers interpret test failures, as it becomes unclear whether the issue lies in a faulty generated test (requiring updates to both the test and the code) or in the code itself (necessitating a code fix and a new test). This diverges from the typical development process, where test failures primarily indicate code issues.

To ascertain the implications of this behaviour in the real world, we manually ran and inspected the output of these tools on a few select issues from SWEBench [31], a dataset used to benchmark LLMs on their ability to solve Software Issues. For instance, we analyzed a bug in SymPy's crypto module (Issue #16884) where an incorrect Morse code mapping for the digit "1" persisted for over three years (January 2016 to May 2019). The bug involved mapping "—-" to "1" instead of the correct ".—-". When presented with this buggy implementation, both CoverUp and CoverAgent discarded tests involving the encoding of "1", effectively masking the bug. Highlighting how the current approach of filtering "failing" tests can systematically hide important edge cases and validation scenarios.

Another issue we observed was with the use of code coverage as the primary objective for test generation. Consider Django issue #34243, where the `timesince()` function crashed with timezone-aware dates due to incorrect timezone handling. While the LLM-based tools generated test suites achieving high coverage, they failed to test crucial functionality across different timezones as test generation is prematurely stopped once coverage targets were met. We also note that when generating from scratch the generated suite was much less comprehensive compared to the human-written test class. We thus strongly believe that as we move closer toward AI-native software engineering, code should be written from requirements which may be expressed through tests in line with Test Driven Development [43]. The design of test generation tools thus probably needs to be such that they assist the developer in writing quality tests rather than attempting to infer the requirements of the user.

3.6 Threats to Validity

The primary threat to our study's validity stems from the dataset composition. While using student-written buggy implementations from Refactory provides us with controlled examples where we have both incorrect implementations and verified reference solutions, these may not fully represent the complexity of bugs found in production systems. However, our findings from real-world examples suggest that the identified issues persist in production environments. Another consideration is our implementation of automated test execution pipelines for each tool. To mitigate this, we maintained default configurations wherever possible and documented necessary modifications in our replication package. While the inherent non-deterministic nature of LLM outputs could affect individual results, our focus on systematic issues in test generation pipelines rather than LLM capabilities means this variability does not significantly impact our main findings.

Our evaluation assumes that the reference solutions represent correct implementations, and our interpretation of test failures and successes might be affected by implementation-specific details of each tool's test generation pipeline. To address these concerns, we manually verified a sample of the reference solutions and focused our analysis on clear-cut cases where tests either definitively validate bugs or fail to detect them, rather than analyzing borderline cases that could be subject to interpretation.

3.7 Conclusion

Our study reveals fundamental issues in the design of current LLM-based test generation tools that could significantly impact software quality. Through empirical evaluation of three tools GitHub Copilot, CodiumAI CoverAgent, and CoverUp, we demonstrate how certain design choices can lead to the validation of bugs rather than their detection, with up to 68.1% of their final test suites validating bugs by passing on incorrect implementations while failing on correct ones. The use of code coverage as a primary objective and the systematic filtering of failing tests can mask critical bugs and create a false sense of security. While these tools show promise in reducing the effort required for test creation, their current design philosophies need significant revision to make use of LLMs for oracle generation and better serve the fundamental purpose of software testing. We strongly believe that code should be written based on requirements and not the other way around. Until these issues are addressed, developers should exercise caution when relying on automatically generated tests, particularly during active development phases when code is most likely to contain bugs.

Chapter 4

Conclusion

This thesis situates its findings within the broader vision of AI-native software engineering. Through empirical studies and TDD-inspired frameworks, we highlight both the promise and the challenges of integrating LLMs into software development workflows.

Chapter 2's conclusions underscore that TDD principles offer significant improvements in LLM-generated code correctness, especially for complex tasks or less capable models. These insights build a case for embedding human guidance, in the form of tests, into automated development workflows. Using tests as guiding constraints during code generation leads to improvements of 9.15% to 29.57% across function-level benchmarks such as MBPP and HumanEval. Incorporating remediation loops further enhances correctness by 5.26% to 9.02%. These benefits are even more pronounced for file-level generation tasks, where traditional LLM workflows struggle. Moving forward, researchers should consider the potential tradeoffs between test creation effort and accuracy gains, and how best to integrate TDD into practical development environments.

Chapter 3's conclusions highlight critical flaws in the design of current LLM-based test generation tools. By exposing how these tools that write tests from code frequently validate bugs, we stress the urgency of rethinking test oracle generation and evaluation strategies. As automated test generation continues to evolve, a future iteration of such tools must better align with the fundamental purpose of testing: verifying requirements and exposing defects, rather than deriving requirements based on the existing code's behaviour. While the current state of LLM-based test generation tools reveals significant limitations, it also highlights opportunities for innovation.

We emphasize based on our findings that code should be written from requirements which may be expressed through tests in line with TDD. This thesis underscores the

potential of TDD to bridge critical gaps in LLM-assisted software development, offering a design that could help build more robust, reliable, and developer-friendly workflows.

4.1 Opportunities for Future Research

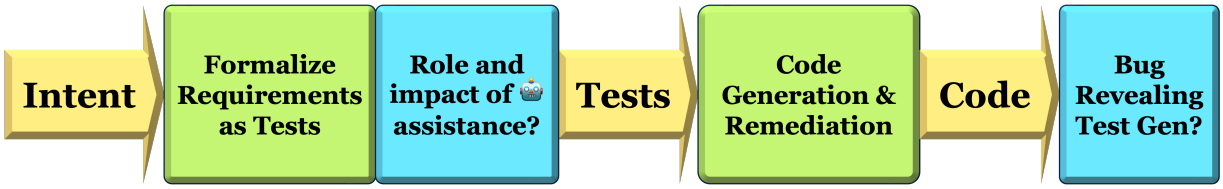


Figure 4.1: General vision and opportunities for future work

This thesis highlights several opportunities for future research in the domain of LLM-assisted programming workflows. Figure 4.1 depicts key questions that could be answered by future work along with the proposed workflow based on the findings of this thesis.

4.1.1 Investigating the Tradeoff between Effort and Accuracy in TDD Workflows

While our findings in Chapter 2 demonstrate the effectiveness of TDD-inspired workflows in improving LLM-generated code correctness, the effort required to write high-quality tests remains a key consideration. Conducting user studies in real-world settings by extending to a full repo context could provide valuable insights into the tradeoff between the time and effort needed to create tests versus the gains in accuracy and robustness of the generated code. Such studies could also shed light on ways in which assistive tools can be built to help developers in this process. *TGen* could be extended to assist developers in writing high-quality tests before generation.

4.1.2 Improved Test Generation Frameworks

The analysis in Chapter 3 reveals significant limitations in current LLM-based test generation tools, such as their reliance on inferred requirements and the inadvertent validation of bugs. Based on our findings, post-hoc test generation setups could be explored to find bugs in code.

References

[1] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too quiet in the library: An empirical study of security updates in android apps' native code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1347–1359. IEEE, 2021.

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 185–196, 2024.

[3] Owura Asare. Security evaluations of github's copilot. Master's thesis, University of Waterloo, 2023.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

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

[7] Kent Beck. *Test driven development: By example*. Addison-Wesley Professional, 2022.

[8] Vasili Braga. Decentralised autonomous society through large language models' based agents: a pathway to empower small communities. *Journal of Engineering Sciences*, pages 88–120, 2023.

[9] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023.

[10] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[12] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232*, 2023.

[13] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*, 2024.

[14] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468, 2024.

[15] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *arXiv preprint arXiv:2310.11248*, 2023.

[16] Thomas Dohmke. The economic impact of the ai-powered developer lifecycle and lessons from github copilot, 2023.

[17] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[18] Khalid El Haji, Carolin Brandt, and Andy Zaidman. Using github copilot for test generation in python: An empirical study. In *Proceedings of the 5th ACM/IEEE*

International Conference on Automation of Software Test (AST 2024), pages 45–55, 2024.

[19] Sarah Fakhoury, Markus Kuppe, Shuvendu K Lahiri, Tahina Ramananandro, and Nikhil Swamy. 3dgen: Ai-assisted generation of provably correct binary format parsers. *arXiv preprint arXiv:2404.10362*, 2024.

[20] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *arXiv preprint arXiv:2404.10100*, 2024.

[21] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE, 2011.

[22] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):1–49, 2015.

[23] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, 70(1):212–230, 2019.

[24] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, et al. Towards ai-native software engineering (se 3.0): A vision and a challenge roadmap. *arXiv preprint arXiv:2410.06107*, 2024.

[25] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[26] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[27] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 388–398. IEEE/ACM, 2019.

[28] Dong Huang, Jie M Zhang, Mingzhe Du, Mark Harman, and Heming Cui. Rethinking the influence of source code on test case generation. *arXiv preprint arXiv:2409.09464*, 2024.

[29] Ronald Jabangwe, Darja Šmite, and Emil Hessbo. Distributed software development in an offshore outsourcing project: A case study of source code evolution and quality. *Information and Software Technology*, 72:125–136, 2016.

[30] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[31] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[32] Shuvendu K Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*, 2022.

[33] Jae Yong Lee, Sungmin Kang, Juyeon Yoon, and Shin Yoo. The github recent bugs dataset for evaluating llm-based debugging applications. *arXiv preprint arXiv:2310.13229*, 2023.

[34] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023.

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[36] Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5315–5333, 2023.

[37] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[38] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[39] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[40] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python. *Empirical Software Engineering*, 28(2):36, 2023.

[41] Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. Automatic programming: Large language models and beyond. *arXiv preprint arXiv:2405.02213*, 2024.

[42] Christopher D Manning. Human language understanding & reasoning. *Daedalus*, 151(2):127–138, 2022.

[43] Noble Saji Mathews and Meiyappan Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1583–1594, 2024.

[44] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

[45] David Noever. Can large language models find and fix vulnerable software? *arXiv preprint arXiv:2308.10345*, 2023.

[46] Juan Altmayer Pizzorno and Emery D Berger. Coverup: Coverage-guided llm-based test generation. *arXiv preprint arXiv:2403.16218*, 2024.

[47] Qodo. GitHub - qodo-ai/qodo-cover: Qodo-Cover: An AI-Powered Tool for Automated Test Generation and Code Coverage Enhancement! — github.com. https://github.com/qodo-ai/qodo-cover. [Accessed 13-12-2024].

[48] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[49] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

[50] Ensheng Shi, Fengji Zhang, Yanlin Wang, Bei Chen, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. Sotana: The open-source software development assistant. *arXiv preprint arXiv:2308.13416*, 2023.

[51] Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

[52] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Susmit Jha, Prem Devanbu, and Toufique Ahmed. Quality and trust in llm-generated code. *arXiv preprint arXiv:2402.02047*, 2024.

[53] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 391–400, New York, NY, USA, 2011. Association for Computing Machinery.

[54] Yuvraj Virk, Premkumar Devanbu, and Toufique Ahmed. Enhancing trust in llm-generated code summaries with calibrated confidence scores. *arXiv preprint arXiv:2404.19318*, 2024.

[55] Jin Wang, Zishan Huang, Hengli Liu, Nianyi Yang, and Yinhao Xiao. Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism. *arXiv preprint arXiv:2309.15324*, 2023.

[56] Xin Wang, Xiao Liu, Pingyi Zhou, Qixia Liu, Jin Liu, Hao Wu, and Xiaohui Cui. Test-driven multi-task learning with functionally equivalent code transformation for neural code generation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2022.

[57] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*, 2022.

[58] Yixuan Weng, Minjun Zhu, Shizhu He, Kang Liu, and Jun Zhao. Large language models are reasoners with self-verification. *arXiv preprint arXiv:2212.09561*, 2022.

[59] Elaine J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 11 1982.

[60] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[61] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[62] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. Prompt-enhanced software vulnerability detection using chatgpt. *arXiv preprint arXiv:2308.12697*, 2023.

[63] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

[64] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024.