Is More or Less Automation Better? An Investigation into the LLM4TDD Process

Sanyogita Piya
University of Texas at Arlington
Arlington, TX USA
sanyogita.piya@mavs.uta.edu

Anahita Samadi
University of Texas at Arlington
Arlington, TX USA
anahita.samadi@mavs.uta.edu

Allison Sullivan
University of Texas at Arlington
Arlington, TX USA
allison.sullivan@uta.edu

***Abstract*—In today's society, we are becoming increasingly dependent on software systems. However, we also constantly witness the negative impacts of buggy software. To produce more reliable code, recent work introduced the LLM4TDD process, which guides Large Language Models to generate code iteratively using a test-driven development methodology. This work highlights the promise of the LLM4TDD process, but reveals that the quality of the code generated is dependent on the test cases used as prompts. Therefore, this paper conducts an empirical study to investigate how different test generation strategies can impact the quality of code produced by the LLM4TDD workflow as well as the impact on overall effort to conduct the LLM4TDD process.**

***Index Terms*—Alloy, Mutation Testing, Test Generation**

I. Introduction

Our lives are increasingly dependent on software systems. However, these same systems, even safety-critical ones, are notoriously buggy. While there are a plethora of software testing and verification techniques, software failures continue to grow in number. A 2022 study found that software failures cost US companies a staggering $2.41 trillion annually, up from $2.08 trillion in 2020 [9]. Therefore, there is a growing need to find ways to produce reliable software.

Recent work introduces a LLM4TDD workflow [15], in which the test driven development (TDD) process is updated to incorporate Large Language Models (LLMs). The LLM4TDD process relies on the idea that test code is conceptually easier to write than implementation code. In particular, while the implementation code must accurately reflect the intricate logic needed to satisfy the system's specifications, test code only needs to compare that a given input produces the expected output. Using this insight, the LMM4TDD framework has the following steps: (1) the user creates a unit test, (2) a LLM generates code such that this unit test now passes, and (3) the user then provides another unit test and the cycle repeats. As a result, the LLM4TDD process enables a human-in-the-loop design in which the user can incrementally guide code generation through seeding in behavior correcting test cases.

While initial results highlight that LLM4TDD has promise, the approach hinges on the quality of the test cases provided by the user. To address this, this paper explores how the strategy behind designing test cases can impact the overall performance of LLM4TDD. In particular, we explore the efficacy of different manual test generation strategies as well as an automated test generation strategy.

Specifically, we make the following contributions:
Study: We perform an extensive study that explores the performance of manual crafted versus automatically generated test suites within the LLM4TDD process, with detailed analysis on the strengths and limitations.
Practical Impacts: Our empirical study reveals various results the community should be aware of, and provides practical guidelines for how best to utilize the LLM4TDD workflow.
Dataset: Our experimental data (including programs, tests, prompts, and analysis scrips) are publicly available online [1].

II. Background

In this section, we introduce key concepts of test driven development and LLM4TDD.

A. Test Driven Development

Test driven development is an incremental software development methodology that focuses on creating tests before the implementation. Specifically, for a given iteration, a software developer considers a test, if it fails, then the developer adds just enough functionality to the code such that the test case now passes. Then, the process restarts with a new test under consideration. As needed, between iterations, the underlying code is refactored. As an example, consider building a calculator program and starting with a test that adds 2 and 3 together. The first iteration of TDD would produce:

```
def test_add_positives
    assert add(2,3) == 5

def add(x, y):
    return 5
```

The minimal code the user needs to create in order for this test to pass is to simply have the `add` method directly return "`5`." In the next iteration, the user would generate another test for `add`, typically targetting missing functionality:

```
def test_add_positives
    assert add(2,3) == 5

def test_add_mixed
    assert add(-2,3) == 1

def add(x, y):
    return x + y
```

In order to make the new and original test pass, the user will now update the function to add the two integers, `x`

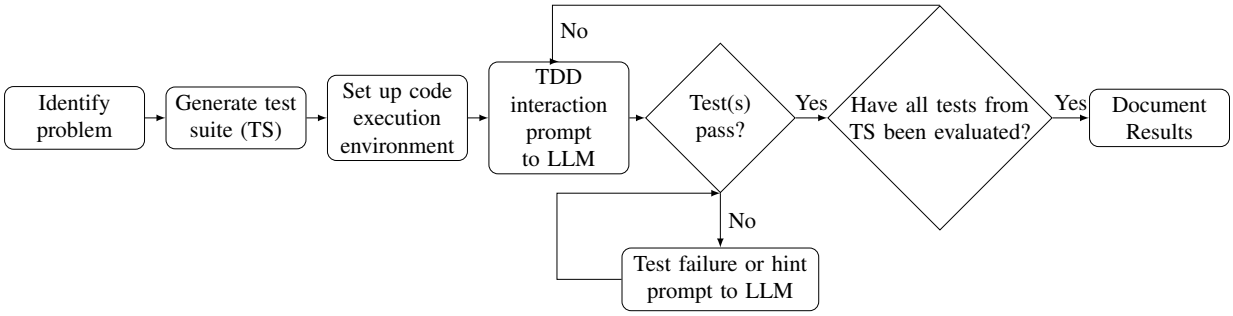

Fig. 1. Overview of the LLM4TDD Process

and `y`, together. Academic and industrial case studies have highlighted that the test driven development process leads to higher code comprehension and quality [4], [13].

B. LLM4TDD

The workflow of the LLM4TDD process is captured in the flowchart diagram seen in Figure 1. First, a problem is identified that the LLM4TDD process will generate code for. This could be any function a developer needs to produce to satisfy their project requirement(s). Second, a test suite is needed that will serve as the bank of tests incrementally sent to the LLM as prompts. From there, LLM4TDD has the developer set up a coding environment. This environment will be used to execute tests against the source code generated. Then, to start the iterative part of the LLM4TDD process, we provide a LLM with a prompt that contains a test case and instructions on how to perform TDD.

As tests are given as prompts to the LLM, tests are also added to the code execution environment. In addition, the LLM response is also transferred to the code execution environment. We then check to see if the generated code actually passes the current test suite. If a test fails, the LLM4TDD has an inner loop in which a prompt providing feedback and asking for a revision is sent to the LLM. Once all tests pass, we restart the cycle with a new test. The LLM4TDD process terminates once all tests in the input test suite pass.

The high level motivation for this split work is that LLM4TDD is designed to keep a degree of developer confidence in the implementation by witnessing the incremental changes that transform the code to address the specific behavior captured by their test. However, manually generating test cases can be a time consuming task. Since test code is conceptually simpler, a developer can often, with minimal effort, understand the general purpose of a unit test. Therefore, if automatically generated tests perform as well or better than manually generated tests, then we could reduce the burden of the LLM4TDD process without loosing this core motivation.

III. Study Design

In this study, we explore the efficacy difference manual and automated test generation strategies within the LLM4TDD workflow.

TABLE I
Decision table for Climbing Stairs

Steps Remaining	1-Step Chosen?	2-Step Chosen?	Expected Ways to Climb
0	No	No	1
1	Yes	No	1
2	Yes	Yes	2
3	Yes	Yes	3

A. Research Questions

We address the following research questions:

- **RQ1:** When manually creating tests, what is the best high level test suite strategy?
- **RQ2:** How effective can automatically generated tests be in the LLM4TDD workflow?
- **RQ3:** What are the trade offs between automated versus manual tests within the LLM4TDD workflow?

B. Subjects

The language model employed for our experiments is **GPT-4o Mini**, which has a knowledge cutoff of October 2023.

For the problem set, we utilize 25 LeetCode problems, carefully selected to ensure the problems are published after the model's knowledge cutoff date to avoid potential bias in results. The same problems are used for both the manual and automated test case generation experiments. The dataset comprises of 10 hard, 11 medium, and 4 easy problems. The problems span a wide array of algorithmic concepts, including arrays, greedy algorithms, dynamic programming, graph theory, string manipulation, bit manipulation, and recursion.

C. Manual Test Generation

We consider four different manual test generation strategies: decision table, boundary value analysis, state transition testing, and input space partition. We chose these four test generation strategies as they are all black box testing techniques, meaning they do do not rely on information about the actual implementation. This is desirable as someone using the LLM4TDD process will not know the implementation in advance.

1) Decision Table Test Generation: A decision table is a tabular representation where each row of the table represents a rule, specifying a particular combination of inputs (conditions) and the expected output (action). The goal of decision table

testing is to turn each row into a test case, which means that every combination of input conditions is covered.

As an example, consider the problem of "climbing stairs" where a person can take 1 or 2 steps to reach the top of a staircase with n steps. The goal is to calculate how many different ways a person can climb to the top. Table I highlights the decision table for this problem. In this scenario, each test case has an input representing the number of steps remaining and the possible choices for taking 1-step or 2-step moves. In the row 2 with "Steps Remaining = 1," the only choice is to take 1 step as the 2-step action is not feasible. Therefore, to encapsulate this row, we would form a test where the input n is 1 and the expected output is 1.

2) Boundary Value Analysis Test Generation: Boundary Value Analysis (BVA) focuses on testing for errors at the boundaries of input domains. This method is effective to expose issues related to off-by-one errors, array overflows, or incorrect handling of upper and lower limits. For example, if it is known that there is an input n that should range between 1 and 100, then BVA would generate the following four tests:

- **Lower Bound:** n = 1 (smallest acceptable value).
- **Just Below Lower Bound:** n = 0 (invalid or out-of-range).
- **Upper Bound:** n = 100 (largest acceptable value).
- **Just Above Upper Bound**: n = 101 (invalid input).

3) State Transition Test Generation: State transition testing is intended to test systems where different inputs or events can cause the system to transition between various states. This is particularly useful when the system under test can be described by a finite state machine.

For example, consider the problem of a least recently used (LRU) cache that has 4 states and 4 transitions. The states are:

- **State 1**: cache is empty (initial state)
- **State 2**: cache is not full, and a new element can be added
- **State 3**: cache is full, and adding a new element will trigger eviction of the least recently used element
- **State 4**: access an existing element

The transitions and triggering conditions are:

- Adding an element (**State 1** to **State 2**)
- Adding elements while the cache is not full (**State 2** to **State 2**)
- If cache is full, adding a new element (**State 3** to **State 3**)
- Accessing an element (**State 3** to a modified version of **State 3** but now with a different least recently used element).

Each transition in the finite state machine is tested individually, ensuring that the system behaves as expected under every possible state change. For instance, one test case would focus on the transition from **State 3** back to **State 3**, where the cache is full, and adding a new element triggers the eviction of the LRU element. This test would ensures the cache transitions correctly by evicting the oldest element while properly retaining the expected elements.

4) Input Space Partitioning Test Generation:: Input Space Partitioning (ISP), which we will refer to as partition testing, is a testing technique where the input domain of the system under test is divided into distinct groups that are expected to exhibit similar behavior. Instead of testing each input individually, one test case is selected from each partition. ISP testing can help testers focus on critical inputs, but does often require the tester to have a deep knowledge of the problem domain.

As an example, consider the problem of removing one letter from a string to equalize the rate of which each character appears in the string. Partitions can be formed by thinking about the functionality of the method, such as (1) all characters already have the same frequency, (2) one character has a different frequency but can be equalized by removing it, and (3) characters have different frequencies such that no removal can equalize the rate of occurrence. Partitions can also be based on attributes of the type of parameters, for instance (1) empty string, (2) single character string, and (3) string with more than one character. By selecting representative test cases from each partition, we ensured that the solution's behavior was validated across a broad spectrum of conditions.

D. LLM4TDD with Manual Test Generation

For a given problem, we pick the test generation strategy that is the best fit for the problem under consideration. We design an initial test case using this test generation strategy, which becomes out first prompt. The manually written tests are carefully tailored throughout the process according to the results shown by the LLM. When a new test is needed for the next iteration, the manual process designs a new test based on the test generation strategy that covers at least one new coverage criteria.

E. Automated Test Generation

For automated test generation, we use the LLM itself to generate the tests. In particular, we use the Chain-of-Thought (CoT) prompting strategy [16]. Chain-of-Thought prompting explicitly instructs the model to generate intermediate steps in its reasoning process, fostering a multi-step logical progression. This approach has been shown to improve performance on complex tasks requiring sequential decision-making or inference. In the context of automated test generation, the model autonomously generates its own chain of thought based on prompts designed to guide the LLMs to iteratively understand the initial problem and based on that understanding, generate a "strong" test case.

Specifically, we take the following five steps:

Step 1: Processing the Problem Description. The first step in the chain-of-though process is to have ChatGPT interpret the problem we plan to generate code for. For our experiments, this means processing the LeetCode problem description. In order to produce test cases for the problem, ChatGPT will need to extract the program inputs and the expected outputs.

Step 2: Recommending a Solution Approach. The next step for the chain-of-thought process is to have ChatGPT reason over the high level algorithmic structure that ChatGPT thinks is needed to solve the problem description processed in step 1. The idea is to have ChatGPT consider the design of the solution, in order to generate more effective tests. For

instance, if ChatGPT thinks a greedy approach is best to solve the problem, then this step can preempt ChatGPT to design test cases around the greedy choice. On the other hand, if ChatGPT thinks a dynamic programming approach is best, then this step can preempt ChatGPT to design tests around a recursive formula's base and generic cases.

Step 3: Generating Test Cases. Next, we ask ChatGPT to generate test cases using four testing techniques: 1) state transition testing, 2) boundary value analysis, 3) partition testing, and 4) decision table testing. For each technique, we have ChatGPT work with the following goals:

- **State Transition Testing:** We first have ChatGPT identify possible states of the system and the transitions between them. Then, for each transition, ChatGPT generates test cases to ensure all valid state changes are covered.
- **Boundary Value Analysis:** We have ChatGPT design tests by selecting inputs at the edges of input ranges, testing critical boundary conditions where the system is most likely to fail.
- **Partition Testing:** We first have ChatGPT divide the input space into non-overlapping partitions (equivalence classes). Then, ChatGPT selects a representative test cases from each class.
- **Decision Table Testing:** We first have ChatGPT generate a decision table, which is modeled through Boolean logic and truth tables. Then, ChatGPT creates test cases based on combinations of conditions in decision tables, ensuring all possible inputs and their corresponding actions are tested.

Step 4: Selecting High-Coverage Test Cases. Once test cases generated, the next step is to have ChatGPT select the test case that provides the highest coverage, with the goal of defining coverage is such a way that the test chosen is the most effective for early-stage validation. To do this, we have ChatGPT compare the generated test cases by evaluating how much of the input space, state transitions, and edge conditions are covered by each one. The test case that maximizes coverage of these critical criteria of interest is then selected. This can be framed as an optimization problem, where the objective is to choose the test case T_i that covers the maximum number of unique input-output conditions or code paths. The goal is to optimize the coverage function:

$$\max_{T} f(T) = \max_{T} \sum_{i=1}^{n} \text{coverage}(T_i)$$

Where T_i represents each generated test case, and $f(T)$ quantifies how much code or decision space is covered by that test case.

Step 5: Writing the Test Case in Python. Lastly, we have ChatGPT translates the test case into Python code using libraries like 'unittest' or 'pytest'. The model structures the test case by defining inputs, expected outputs, and assertions that verify the correctness of the solution. This transformation from abstract test conditions to executable Python code ensures that the test can be automated and run in standard development workflows.

F. LLM4TDD with Automated Test Generation

For a given problem, we maintain two separate LLM sessions. In the first session, we have the LLM generate tests following the steps outlined in Section III-E. In the second session, we use the test case outputted by step 5 as a prompt for the LLM4TDD process. When a new test is needed for the next iteration, the first session is used to produce a new test case that must differ from all previously generated test cases.

G. Evaluation Metrics

For our experiments, whether using manual or automatically generated tests, we track the following metrics to assess performance with respect to the LLM4TDD process.

- **Number of LLM4TDD Iterations:** The number of iterations needed before the LLM generates the correct code, where an iteration consists of generating code, running the code against the provided test(s), and evaluating the output. For our experiments, fewer iterations indicate the provides tests result in faster code generation.
- **Number of Prompts:** The number of prompts provided to the LLM. This includes test prompts as well as any prompts asking for clarifications on test cases or guidance on problem interpretation. For our experiments, fewer prompts indicate the test cases provide better clarity, leading to a more accurate problem interpretation by the LLM.
- **Number of Tests:** The number of tests that are provided to the LLM during all the iterations. For our experiments, the goal is to have a prompt to test ratio of 1:1, which would mean that the LLM never needed any of the additional prompts to provide clarity or address compilation or test failure issue.
- **Number of LeetCode Tests Satisfied:** The number of official LeetCode test cases passed by an iteration. Every LeetCode problem has a hidden test suite the submissions are evaluated against.
- **Total LeetCode Tests:** The total number of test cases that LeetCode uses to evaluate the problem's solution. We use this as our baseline for calculating accuracy.
- **Accuracy Percentage:** The percentage of LeetCode test cases passed out of the total test cases after an LLM4TDD iteration. We use the following formula to calculate this:

  $$(\text{Total LeetCode Tests}/\text{\# LeetCode Tests Satisfied}) \times 100$$

  A higher accuracy percentage indicates a more effective test case generation method, as the code generated better matches the intended functionality. We consider the *final accuracy* to be the accuracy after the last iteration.
- **Predicted Best Type of Test Generation Method:** This metric is tracked for automatically generated tests only. Based on the characteristics of the problem, we predict the most suitable test case generation method.

In addition, for each LeetCode problem, we track the following metrics:

- **Problem Publication Date:** Identifies when the LeetCode problem is published and whether the publication date is

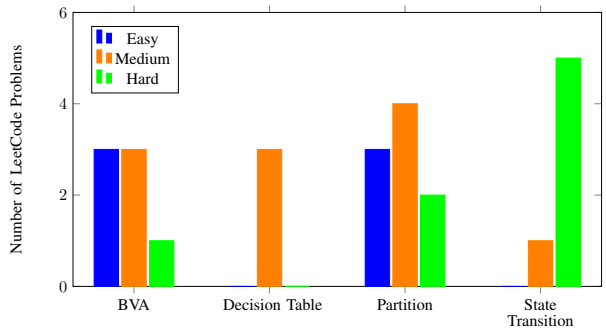

Fig. 2. Test Generation Method Distribution by Difficulty (Manual)

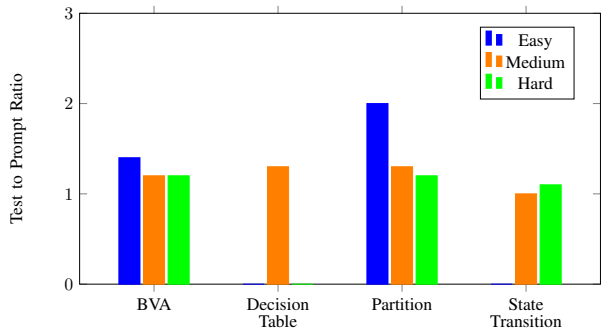

Fig. 3. Test to Prompt Ratio (Manual)

before or after the LLM's training data cutoff. In our experiments, all problems are published after the cutoff, ensuring that performance of the LLM is not biased.

- **Difficulty Level:** The LeetCode difficulty rating (Easy, Medium, Hard) for each problem.This helps determine whether a test generation method is more suitable for problems of a certain difficulty level.
- **LeetCode Algorithm Used:** The primary algorithm or data structure required for solving the problem, such as dynamic programming, greedy, divide and conquer, tree, graph, or stack. Certain algorithms may work better with specific test generation techniques, which will help refine our recommendations.
- **ChatGPT Session Links:** A reference to the ChatGPT session used for generating the code, including the conversation logs.

H. Stopping Criteria for the LLM4TDD Iterations

Since LLM4TDD is an incremental development process, it is possible that to solve a problem, a prohibitive number of iterations may be needed. Therefore, we define the following stopping criteria to place a reasonable bound on the depth of iterations explored. First, the maximum number of generated tests is capped at 10. If the correct solution is not generated after exploring 10 tests cases, we assume the test generation method might require more developer intervention or better design. Second, The number of prompts for additional information from the LLM is capped at 10. If the LLM requires more than 10 prompts for clarifications, we will stop.

IV. Result Analysis

In this section, we first analyze the performance of the various manual test case generation techniques, assessing their impact on LLM4TDD's ability to produce correct solutions. Next, we evaluate the performance of utilizing a fully automated approach, with the LLM generating both the tests and the code. Finally, we compare and contrast the differences in behavior between using manually generated versus automatically generated tests to determine best practices for the LLM4TDD workflow.

A. RQ1: What makes for effective manual test generation for LLM4TDD?

1) Test Generation Method Versatility Across Problem Difficulty: As part of the manual process, we need to decide which test generation method to focus on for a given problem. Figure 2 illustrates the distribution of test case generation methods across problem difficulties, highlighting the suitability of each approach. Boundary value analysis and partition testing demonstrate balanced usage across easy, medium, and hard problems, suggesting they are adaptable methods capable of addressing a range of complexities. Boundary value analysis's focus on edge cases is similar in nature to partition testing's focus on dividing input spaces into equivalence classes. In contrast, decision table testing is primarily applicable to medium difficulty problems, indicating that if decision tables are relevant for a problem, the problem is likely to have some complexity to it. State transition testing is predominantly used for hard problems. Likewise, the problem space being represented by a finite state machine likely means there is an inherently level of complexity to the task.

Finding 1: While BVA and partition testing offer general versatility, decision tables and state transition testing is more relevant in specialized contexts.

2) Test Generation Method Performance Across Problem Difficulties: The test-to-prompt ratio, shown in Figure 3 provides a quantitative measure of each test case generation method's efficiency, comparing them against an ideal ratio of 1:1, where each test case would only require one prompt for the LLM to produce correct code. Among the test generation methods, boundary value analysis, state transition testing, and decision table testing demonstrate the closest alignment with this ideal. All of these methods focus on testing for specific input to condition relationships or paths through a program. As a result, these test cases may convey the intended behavior of the system in a more direct manor. Partition testing shows a high ratio for easy inputs but becomes more efficient as difficulty increases. For a given problem, the partitions derived could be more open ended and ambiguous in nature, as representing an equivalence class of an input domain is not as concrete as edge cases or important transitions.

Finding 2: Test generation methods that focus on outlining specific "important" inputs to focus on are more likely to provide better guidance as a prompt for code generation than methods with more abstract criteria.

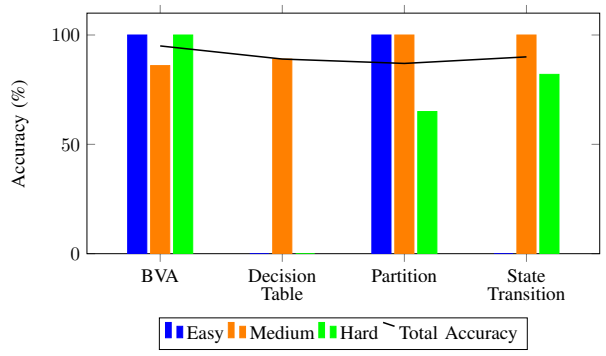

Fig. 4. Accuracy (Manual)

3) Accuracy Across Test Generation Methods: Figure 4 illustrates the average accuracy percentage achieved by each test case generation method, broken down by difficulty levels, with a total average accuracy indicated by the solid black line. Boundary value analysis has the highest accuracy, approaching 95%. This result aligns with BVA's focus on boundary-related issues that are straightforward for the model to generate code for. Decision table, state transition testing and partition testing show moderately high accuracy, with each achieving 89%, 87% and 90% respectively. However, while boundary value analysis' accuracy remains high for hard problems, both partition testing and state transition testing see a dip in accuracy down to 65% and 82% respectively. Accordingly, focusing on edge cases, which is a very specific criteria, provides better guidance than more high-level design patterns like classifications of inputs or covering conditions. as problem difficulty increases.

Finding 3: Boundary value analysis is the best choice for manual test generation to achieve accurate code. However, at large, manually generated test lead to the LLM4TDD workflow producing higher quality code.

4) Impact of Problem Complexity on Method Performance: The nature of the problems plays a significant role in the test to prompt ratio and accuracy of each test case generation method. Dynamic programming and graph problems, which are among the most complex types encountered in this experiment, are primarily handled by state transition and decision table testing. This may have contributed to the slightly lower accuracy and higher test to prompt ratio observed in these methods compared to boundary value analysis.

In contrast, simpler problem, such as mathematical operations and basic input-output mappings, are efficiently handled by boundary value analysis. These problems generally involve straightforward edge cases, which likely lead to a better accuracy and prompt-to-test ratios within the LLM4TDD workflow. String manipulation and array problems, which primarily involve structured but diverse data inputs, were addressed with partition testing. These problems have more room for ambiguity and a more diverse set of possible operators than integer inputs, which likely influences the slightly worse difference in performance between partitioning tests and boundary value analysis, decision tables and state transition testing.

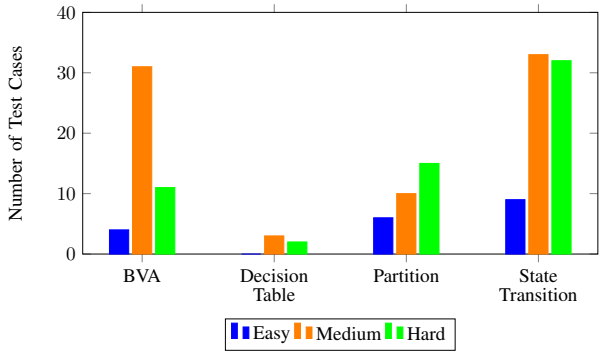

Fig. 5. Test Generation Method Distribution by Difficulty (Automated)

Finding 4: The performance of different test generation methods is coupled with the difficulty of the problems that utilized each method. Of note, decision tables and state transition testing have a good performance, given the underlying complexity of the problems these test generation strategies were applied to.

B. RQ2: Can automatically generated test be effective within LLM4TDD?

To ensure a reliable comparison, we used the same set of problems as in RQ1. However, the test generation method employed for each iteration is determined by the LLM, as outlined in Section III-E. This means that within the same LeetCode problem, the automated approach can utilize multiple different test generation methods. As a result, we do not present results broken down by method, but instead present aggregate results based on difficulty.

1) Test Generation Method Frequency: Although the test generation strategy used by the LLM can vary iteration to iteration, the different test generation strategies are still utilized at different rates throughout the problems. To capture this, Figure 5 shows how often an iteration's test case is generated based off of the four different test generation methods. The automated approach frequently uses boundary value analysis and state transition testing to produce tests, but rarely uses decision table and partition testing. However, when first picking a test, the chain of thought process often starts with a partition or state transition based test case. Then, for subsequent iterations, the process predominately selects boundary value analysis and state transition tests to guide the next iteration of code genration.

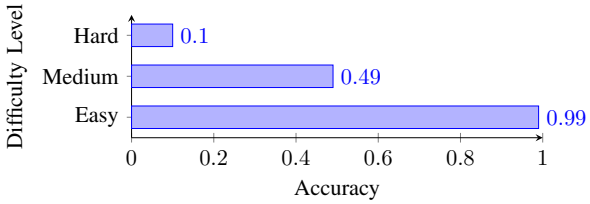

Fig. 6. Accuracy Across Difficulty Levels (Automated)

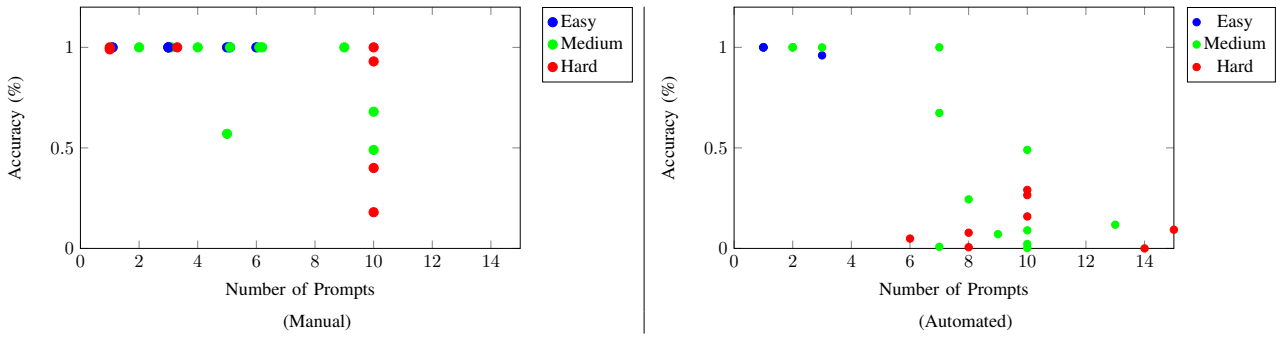

Fig. 7. Relationship between #Prompts and Accuracy.

Finding 5: ChatGPT views partition and state transition testing as a good way to ensure high-priority coverage early, but relies on boundary value analysis and state transition testing to refine behavior.

2) Accuracy Across Problem Difficulties: Figure 6 illustrates the overall accuracy percentage achieved by utilizing automated test generation, broken down by difficulty. As the difficulty of the problem increases, there is a notable decrease in accuracy. For hard problems, some problems achieve as low as 9.3% accuracy or even 0% accuracy. This occurs for a few reasons. First, ChatGPT struggles with solutions that require recursion, and recursion is often associated with higher difficulty problems. Second, detailed descriptions enhance response quality in simple to medium problems, leveraging ChatGPT's language-based processing capabilities. However, in more complex scenarios, overly detailed descriptions often led to confusion instead. Importantly, if an initial solution approach is significantly flawed, ChatGPT may become stuck on refining an incorrect solution rather than starting over. In addition, with our automated tests, ChatGPT often starts with brute-force approach. For easy problems, a this design choice may be appropriate, but as the difficulty increases, a brute force approach is less likely to solve the problem.

Finding 6: As the difficulty increases, automatically generated test cases fail to provide proper guidance, leading to LLM4TDD iterations that produce code that strays further and further from the objective.

3) Prompt to Test Ratio: The average prompt to test ratio across all problems is 1.03, with easy problems having an average ratio of 1, medium problems having an average ratio of 1.06 and hard problems having an average ratio of 1.01. When there was a runtime or compilation error, a guidance prompt with the code correction direction was provided. However, overall, the automated approach rarely entered the inner loop of the LLM4TDD workflow.

C. RQ3: How does manual tests compare to automated tests for LLM4TDD?

1) Overall Accuracy: Ultimately, the goal of the LLM4TDD workflow is to automatically generate code. To that end, using manually generated tests, the overall accuracy is 90.3%, with easy problems having an accuracy of 100%, medium problems having an accuracy of 93.1% and hard problems having an accuracy of 78%. Using automatically generated tests, the overall accuracy is 49.11%, with easy problems having an accuracy of 99%, medium problems having an accuracy of 49% and hard problems having an accuracy of 10%.

Finding 7: Based on accuracy, the automated approach struggles with the reasoning required for more complex scenarios, where custom-designed test cases can better guide the LLM, improving performance.

2) Tradeoff in Effort for Accuracy: The other aspect to LLM4TDD is how much effort will LLM4TDD require of the user? For the fully automated approach, the user effort is simply the time to transfer in the test case for each iteration. For manually generated test, the user effort is the time required to produce a new test, which includes inspecting the current code iteration in order to design a new test that both provides additional coverage and accounts for behavior not currently supported. In our experience, manually generating tests required on average 7 minutes per problem. Quantitatively, the total number of prompts for manual testing was 134 including 106 unique test cases, which results in a prompt-to-test ratio of 1.26. In contrast, for the fully automated approach, the total number of prompts issued was 202 including 195 unique test cases, which results in a prompt-to-test ratio of 1.04. As a result, while less tests, the manual process requires more effort due to the difference in test generation time and the fact that these tests trigger the inner loops of the LLM4TDD process more often than the automated tests.

However, this tradeoff in effort is coupled with tradeoffs in accuracy. To help illustrate this, for both the manual and automated test generation methods, Figure 7 illustrates the relationship between the number of prompts required to achieve the associated accuracy, broken down by difficulty levels. For manually created tests, most of the problems are clustered at the "1" accuracy mark, meaning 100% accuracy. However, the number of prompts needed to achieve this accuracy varies significantly. There is an overall trend that easy problems require less prompts to achieve an accuracy

of 1, while medium problems and hard problem successively require more prompts. In addition, occasionally medium and hard problems can require a large number of prompts, while also failing to achieve a high accuracy, which means these problems occasionally have a high manual cost for little gain

For automatically generated tests, these results further highlight that the automated approach works well for problems with less complexity. In general, easy problems cluster to the top left corner, indicating that these problems both have high accuracy but also require very few prompts, and thus less iterations of the LLM4TDD process. However, hard problems, which often involve solutions that utilize dynamic programming or recursion, cluster to the bottom right corner, indicating that these problems both have a low accuracy despite requiring a high number of problems. Ultimately, a fully automated LLM4TDD process excels at easy problems with little prompting, but is not effective for harder difficulty problems despite a large increase in prompts utilized.

Finding 8: The fully automated approach requires less effort. However, the automated approach lacks the adaptability needed to effectively handle more complex problems. Therefore, the LLM4TDD process is best suited for manually generated tests, as despite the increase in effort, these test better convey missing or complex behavior.

V. Threats to Validity

There exists several threats to the validity of our results. First, our evaluation focuses on LeetCode problems, which may not be representative of the performance the LLM4TDD process can achieve over real world problems. However, these problems do focus on single method solutions. Since the TDD process centers on unit tests for a single method, achieving strong performance over LeetCode problems is a good basis to refine the LLM4TDD process over. Second, the performance for the manual process can vary depending on the expertise of who is generating the tests. Third, our experiments focus on four different test generation strategies, but our results indicate performance is likely to vary with different, not currently evaluated test generation strategies.

VI. Related Work

Code and Test Generation with LLMs. In recent years, code generation techniques utilizing large language models have been explored [12], [10], [2]. In addition, several LLMs have been developed specifically for generating code, including Codex [3], CodeGen [14], InCoder [5] and PolyCoder [17], which could be used as the backend LLM in the LLM4TDD process. For test generation, CodeCoT [8] applies Chain-of-Thought prompting to generate both solutions and test cases for programming tasks. Similarly, Reflexion and AgentCoder utilize LLMs to generate test cases and execute them on programs to provide feedback for self-correction. While we focus on Chain-of-Thought prompt, integrating the prompt design from these recent works could be considered to improve the automated version of LLM4TDD.

Test Driven Development. Several studies have found that test driven development practices lead to better code, often measured by the code passing more test cases, at the expense of the development cycle taking more time [7], [6], [11]. A study conducted at IBM additionally found that developers felt that they better understood the system's design when using test driven development compared to other development cycles and that developers felt that their code was better designed and more readable [13]. Test driven development has also been evaluated in an academic setting, in which students reported feeling as if they better understood their programs and this also felt more confident in making changes to their code [4].

VII. Conclusion

The LLM4TDD workflow is an incremental development process in which the user guides code generation by gradually presenting unit tests to the LLM for the LLM to generate code to pass. As a result, this process hinges on the test cases that are used as guiding prompts. Therefore, this paper explores how different test generation strategies can impact the efficacy of LLM4TDD. Our results highlight that careful crafted test cases that are manually produced lead to higher quality code.

References

[1] https://anonymous.4open.science/r/LLM4TDD-Manual-Vs-Automated-CAA1/README.md (2024)

[2] Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.G., Chen, W.: Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397 (2022)

[3] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

[4] Desai, C., Janzen, D., Savage, K.: A survey of evidence for test-driven development in academia. ACM SIGCSE Bulletin **40**(2), 97–101 (2008)

[5] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.t., Zettlemoyer, L., Lewis, M.: Incoder: A generative model for code infilling and synthesis. ICLR (2022)

[6] George, B., Williams, L.: An initial investigation of test driven development in industry. In: SAC. pp. 1135–1139 (2003)

[7] George, B., Williams, L.: A structured experiment of test-driven development. Information and software Technology **46**(5), 337–342 (2004)

[8] Huang, D., Bu, Q., Qing, Y., Cui, H.: Code-cot: Tackling code syntax errors in cot reasoning for code generation. arXiv preprint arXiv:2308.08784 (2024)

[9] for Information, C., Quality, S.: The cost of poor software quality in the us: A 2022 report (2023)

[10] Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping language to code in programmatic context. arXiv preprint arXiv:1808.09588 (2018)

[11] Janzen, D., Saiedian, H.: Does test-driven development really improve software design quality? Ieee Software **25**(2), 77–84 (2008)

[12] Liu, Z., Chen, C., Wang, J., Che, X., Huang, Y., Hu, J., Wang, Q.: Fill in the blank: Context-aware automated text input generation for mobile gui testing. In: ICSE. pp. 1355–1367. IEEE (2023)

[13] Maximilien, E.M., Williams, L.: Assessing test-driven development at ibm. In: ICSE. pp. 564–569 (2003)

[14] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022)

[15] Piya, S., Sullivan, A.: LLM4TDD: best practices for test driven development using large language models. In: LLM4CODE@ICSE. pp. 14–21 (2024)

[16] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. NIPS **35**, 24824–24837 (2022)

[17] Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J.: A systematic evaluation of large language models of code. In: MAPS. pp. 1–10 (2022)