Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

TestPilot: An LLM code test assistant for novice programmers

Prathamesh Sai Sankar

April 15, 2024

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Masters in Computer Science

Declaration

Abstract

Imagine you are a novice programmer who has just written some code that you hope works as intended. You are able to describe what your code should do in words, but you struggle to think of test cases and implement them syntactically. You look online for help, but the test cases you find are either too complicated, not adapted to your messy beginner code, or do not compile and pass. Since you do not know the syntax well enough to fix any of these problems, you feel like giving up.

Instead of looking online, you ask a large language model (LLM) to generate test cases. However, this results in complicated tests that are not consistent or suitably documented, since LLMs are trained using a corpus that is not necessarily suited for novice programmers. Furthermore, the tests do not "wrap" around your messy beginner code correctly, since you are unable to prompt the LLM with the necessary context from your limited understanding. Finally, the tests fail at compilation and runtime since LLMs face hallucinations, which cause incorrect tests. From this, you realize that LLMs face the same issues as standard online searches.

The aim of this thesis is to develop TestPilot, a Visual Studio Code extension for Java, one of the most common programming languages used by novice programmers. A novice programmer can input a beginner natural language description of what their code should do, and receive high-quality JUnit test cases that are semantically and syntactically correct, at an appropriate level for a novice programmer, and written in a consistent style that is suitably documented.

The extension interacts with a server that completes a strategic sequence of LLM inferences to generate a suite of test cases using research-driven prompt engineering, fine-tuning, and embeddings. The resulting test cases are displayed to the novice programmer using natural language and code implementations that follow strict testing practices. The code coverage of the tests is displayed to encourage the novice programmer to explore the problem space instead of blindly copying code. It presents a "testing" and "discovery" mode of operation to help a novice programmer depending on their learning goals.

Compared to a single prompt to ChatGPT, TestPilot generates tests that are 28% more likely to compile, 26% more likely to test the intended logic from a vague prompt, and 30% more likely to include documentation through comments. A novice programmer who initially struggled to find help using online searches or standard LLMs is now able to understand tests that are catered for their level and more correct to help them learn about software testing.

Acknowledgements

I would like to thank my supervisor Dr. Jonathan Dukes for his continuous support and feedback which helped guide me throughout the course of my work on this dissertation. I am eternally grateful for his belief in me and his help throughout the academic year.

I would also like to thank my family for their support and encouragement throughout my undergraduate and postgraduate studies at Trinity College Dublin. Their commitment and dedication to education motivated me to get this far and pursue this project.

Finally, I would like to thank my friends and fellow classmates who I met across the past 5 years of my university life. I'm grateful for our shared journey in academia. It has been a long road, but we made it!

Contents

1 Introduction

Testing is an essential part of programming. Writing good tests is not only important to find bugs in your code, but to also explore the problem space. Although novice programmers might have the necessary skills to solve a problem, they might not yet have the knowledge to develop a suite of tests to evaluate the correctness of their program. Implementing tests early has been proven to help novice programmers achieve greater correctness and code coverage, yet 76% of them do not follow consistent testing behaviours across different projects [1]. It is clear that novice programmers must be supported in adopting consistent testing habits early on.

When a novice programmer does not have the necessary skills to test their code, they might be tempted to search for help online. However, tests from different websites do not guarantee a consistent style that is sufficiently documented to educate novice programmers about good testing practices. Furthermore, it may be frustrating to find tests online that are too complicated to understand, not adapted to messy beginner code, or do not compile and pass. Only 1% of Java code snippets from websites such as Stack OverFlow compile on their own due to a lack of class encapsulation, import statements and semicolons [2]. Therefore, online source code is not suitable for an inexperienced novice programmer.

Large language models (LLMs) are a possible solution to bridge the gap between novice programmers and educational software testing. However, the same problems from standard searches online persist [3]. LLMs are trained using a large amount of data from various sources, resulting in a model that may generalize to complicated tests that are not consistent or suitably documented. Furthermore, LLMs can struggle to "wrap" around messy beginner code using limited context from a novice. Finally, LLMs may hallucinate, leading to failed tests at compilation or runtime. Therefore, it is evident that adapting LLMs for novice programmers learning software testing is necessary.

This dissertation introduces TestPilot, a Visual Studio Code extension that leverages LLMs using research-driven prompt engineering, fine-tuning and embeddings to help novice programmers learn how to write JUnit tests in Java. With this interface, a novice programmer can input a beginner natural language description and receive high-quality educational test cases in natural language with corresponding code implementations that are semantically and syntactically correct, at an appropriate level for a novice programmer, and written in a consistent style that is suitably documented.

1.1 Motivation

While novice programmers may have the ability to develop a solution to a problem, they may lack the ability to develop a suite of tests to assess their code. When stuck, novice programmers are often tempted to seek information online [4]. When doing so, they frequently use multiple online sources to help them test their code. Since different websites use different coding standards that may not be at a novice-level, this behaviour does not promote consistent software testing practices that a novice programmer can learn from. Furthermore, tests found online may not be correct semantically (by testing the intended logic) or syntactically (by using the correct programming syntax). Therefore, it is not educational or practical for novice programmers to find tests online.

Each online source has varying levels of consistency and documentation, with optional explanations for code being written in text above or below a code snippet. It has been examined that novice programmers often look directly at the code when seeking help online, skipping all other context before or after it [5]. With this limited context, novice programmers only pay attention to 27% of a code snippet on websites such as Stack OverFlow [6]. When using code with limited context in this manner, novice programmers are not encouraged to test with consistent styling and suitable documentation.

Tests found online may be incorrect for a novice programmer's code because they are semantically incorrect (since there are different implementations of the same problem) or syntactically incorrect (since code blocks might be truncated) [7]. If an inexperienced novice programmer wanted to fix these issues, they might not have the adequate knowledge to fix them, especially if the original code they found was not at a novice-level.

LLMs are a possible solution to provide novice programmers with educational tests. However, issues related to finding tests online continue to exist. Training data for LLMs are not suited for novice programmers, leading to tests that are not consistent or suitably documented. Furthermore, LLMs can struggle to contextualize a novice programmer's messy code to provide an accurate solution. Finally, hallucinations may occur where an LLM provides an incorrect solution. This establishes a strong motivation to adapt LLMs for novice programmers learning software testing in a programming language through an interface.

Novice programmers are often enrolled in introductory programming courses, for which studies suggest that Java is best suited for students learning to program [8]. JUnit is a popular testing framework for Java among novice programmers learning software testing. Furthermore, Visual Studio Code extensions have been a successful interface for novice programmers interacting with LLMs for general use cases such as code completion [9]. Therefore, there is an incentive to build a Visual Studio Code extension that interacts with LLMs to provide reliable and educational JUnit tests for novice programmers.

TestPilot, a Visual Studio Code extension for teaching novice programmers how to write educational JUnit tests in Java, would be helpful to bridge the gap between novice programmers using standard searches online and asking commonly known LLMs such as ChatGPT to generate tests, both of which may be inconsistent, incorrect, or non-educational. The motivation behind TestPilot involves developing a methodology to adapt LLMs for preventing the same issues that occur from novice programmers using arbitrary code found online.

1.2 Proposed methodology

The proposed methodology for TestPilot is developed by taking three pillars into account, which include cost (number of tokens used for LLM input and output), correctness (semantically and syntactically), and education (tests that are consistent with documentation at the level of a novice programmer). Finding the right balance between all of these pillars is difficult, since too much focus on one pillar can impact the rest. These pillars were used to create two modes of operation.

The first mode of operation is "testing", which is aimed at novice programmers who are unsure that their code is correct, and want to test it by giving a natural language description of a test. For example, a novice programmer may write a simple function to calculate the factorial of a natural number using recursion. A novice might not be sure if their code is correct since they have not called it in the main method of the same file yet. By inputting a beginner description "I think my code should return 120 for an input of 5", TestPilot takes their description as the source of truth and generates a test against the novice's code in both natural language on a card and in code. The novice can input more test cases which are appended to the list of cards and test functions. The corresponding test functions are put in a separate testing file which may pass or fail depending on whether the logic in the description matches the logic in the code.

The second mode of operation is "discovery", which is aimed at novice programmers who are happy that their code is correct, but want to discover test cases that demonstrate this. Using the same factorial function from before as an example, a novice might be happy their function is correct since they called it in the main method of the same file with inputs of 0 (only edge case) and 5 (normal case) and it returned 1 and 120 respectively. However, they are unsure how they could discover test cases to demonstrate this formally. By inputting a beginner description "I think my code should return 120 for an input of 5", TestPilot takes their code as the source of truth and generates a list of "blurred" natural language test cases in cards for the entire program space. The card matching the novice's description is "unblurred", and the novice can input more test cases which "unblurs" more cards if they are correct. Test functions for these cards are put in a testing file which always pass.

In either mode of operation, the code coverage of all test cases shown as cards against the novice's code is displayed in a progress bar. Only in "discovery" mode, there is a second progress bar for case coverage, which shows the code coverage of the test cases from all the "unblurred" cards. These two progress bars encourage the novice to focus on *program coverage*. Alternatively to *program coverage*, studies show that novice programmers lack the awareness of *problem coverage* (code coverage for testing the complete solution, which the novice's code might not be) [10]. A focus on *problem coverage* is implemented by displaying a tooltip which shows the code coverage and number of test cases of a previously known working test suite if the code that is being tested has appeared before on TestPilot. If a student has less test cases and code coverage than the previously known test suite, they might have missing functionality in their code.

When generating tests in "testing" mode, the novice's description is converted from a vague description to a clear description using a highly accurate but relatively expensive model such as GPT-4. Since the expensive model only has to produce a single description, expensive tokens from the input and output of lengthy code files are avoided. This concise description is subsequently passed to a cheaper but fine-tuned GPT-3.5 Turbo model to create a "@Test" snippet. Each additional natural language description inputted from the novice uses another GPT-4 and fine-tuned GPT-3.5 Turbo inference, with the resulting test being appended to the list of test cases in cards as well as the code.

When generating tests in "discovery" mode, all necessary test cases in English for complete code coverage are generated using a highly accurate but relatively expensive model such as GPT-4. Since the expensive model only has to produce a list of test cases, expensive tokens from the input and output of lengthy code files are avoided. Each of the N test cases are outsourced to a cheaper but fine-tuned GPT-3.5 Turbo model to create their respective "@Test" snippet which are combined through string manipulation. Each additional description from the novice to "unblur" a test case uses another GPT-4 inference.

When using cheaper models for code generation, they often produce less reliable output. To combat this, it requires careful prompt engineering, fine-tuning and embeddings. LLMs are trained on data that follow human behaviour. Even if prompts including "I'm going to tip $xxx for a better solution!" may seem silly, they are shown to improve the resulting output of cheaper models [11]. Furthermore, fine-tuning allows the model to learn stylistic behaviours which is essential for this use case, since novice programmers require solutions at a novice-level that are consistent, correct, and educational. Finally, embeddings can be used by LLMs to increase their knowledge base. A vector database can contain embeddings for known good working solutions, where any previously successful tests are automatically appended to the database.

Before returning the generated tests, sanity checks are completed by compiling the code with Java and a JUnit. Code coverage is calculated using JaCoCo. Finally, code is formatted using google-java-format before it is returned to the novice programmer. By implementing this methodology, a novice programmer is encouraged to explore the problem space while also benefiting from more accurate tests that are at a novice-level with suitable documentation.

1.3 Challenges

TestPilot involves both education and LLMs. Therefore, challenges for developing TestPilot stem from either education, LLMs, or both. Each of these categories are conflicting of each other. For example, education needs to be reliable to be informative, yet LLMs have been shown to be unreliable [12]. Since both aspects of TestPilot are inherently contradictory, the resulting challenges from this contradiction need to be addressed.

LLM alignment is an issue that causes inconsistent tests with unsuitable documentation [13]. If a novice programmer prompted an LLM to return a novice solution in a consistent style that is documented with comments, this cannot be guaranteed. Overcoming LLM alignment is a challenge that needs to be addressed to prevent novice programmers receiving unsatisfactory tests that are not educational.

LLMs face hallucinations which can cause incorrect tests [14]. Knowledge from LLMs is encoded in a probabilistic manner across many parameters, meaning that the encoding of data is lossy. This results in an LLM generalizing to incorrect tests even if the required knowledge is directly in the training data. Overcoming hallucinations is a challenge to overcome the possibility of incorrect tests being presented to novice programmers.

Fine-tuning requires large amounts of data. Data for LLMs needs to be diverse but accurate, both of which cannot be guaranteed by scraping the web. LLMs are often used to generate training data for other LLMs. However, LLMs are inherently unreliable, therefore the generated data can be unreliable. Researchers have experimented with temperature sampling to increase data diversity but this reduces the accuracy of the data [15]. Overcoming the challenge of data collection for LLMs is essential to find a balance between diverse and accurate data.

Educational resources that use artificial intelligence must avoid learner over-reliance [3]. LLMs pose the risk of influencing novice programmers to not think for themselves, and instead use LLMs to do all the work. Reliance on LLMs to complete a suite of tests removes any educational benefit from exploring the problem space. The challenge of preventing learner over-reliance must be overcome to ensure novice programmers do not rely on LLMs completing all necessary tests on their behalf.

The interface of TestPilot must be designed to foster exploration rather than being a search engine that simply shows results. The user interface must be developed in a way that encourages the novice programmer to explore the problem space. Implementing the user interface in a manner that is educational and useful requires careful planning and intricate workflows which is a challenge.

Tests can use a variety of techniques to make them educational such as coding standards, testing strategies and function layouts. For example, testing strategies such as "Arrange, Act, Assert" emphasize clean coding standards [16]. Having a consistent structure when testing is essential for teaching novice programmers consistent testing habits. Choosing the coding standards, testing strategies and function layouts for TestPilot to be educational and not just a "test generator" for copying and pasting code is a challenge.

1.4 Goals

The aim of TestPilot is to be more accurate than a single prompt to ChatGPT. TestPilot uses multiple LLM inferences with artificial intelligence techniques which involves extra cost. Therefore, it is essential that the extra cost is worth it compared to ChatGPT producing tests that might be inconsistent, incorrect or both.

A goal of TestPilot is to manage the pillars of cost, correctness, and education. Using cheap models to decrease costs reduces correctness and educational benefit since cheap models are less capable. Conversely, prioritizing correctness and educational benefit requires expensive models which increases cost. TestPilot's objective is to manage these three pillars to find the Pareto frontier for this use case.

TestPilot aims to include a robust methodology that works regardless of LLM failures. Since LLMs are inherently unreliable, incorrect results may appear at any point in a chain of LLM inferences. Accurately finding incorrect responses through sanity checks and verification is helpful to the system robust.

A goal of TestPilot is to minimize hallucinations and LLM alignment issues through prompt engineering, fine-tuning and embeddings. These issues are open research problems that are not fully preventable since LLMs are unreliable. However, minimizing these issues as much as possible is a goal when developing TestPilot.

TestPilot aims to use diverse and accurate data for fine-tuning. Generating synthetic data that is both diverse and accurate is difficult. However, a goal of TestPilot is to use seeding techniques to generate diverse data that is verified to be accurate through sanity checks. Using diverse but accurate data will increase the ability of the fine-tuned LLMs to generalize to consistent and suitably documented solutions for a wide range of tests.

A goal of TestPilot is to implement the user interface in a way that is educational. This is completed by taking visual cues into account, such as progress bars and cards that display useful information for a novice programmer. Implementing the user interface in an easy-to-use format helps foster exploration and learning.

1.5 Dissertation Overview

This section will provide a high-level outline for the remainder of this dissertation. The remaining chapters are as follows:

- Chapter 2: Background
    - This chapter will discuss the technical background behind how LLMs function and how they can subsequently be adapted.
    - This chapter will conduct a literature review on the use of generative artificial intelligence by novice programmers.
- Chapter 3: Design
    - This chapter will present the system architecture of TestPilot, including how the frontend and backend systems interact with each other.
    - This chapter will discuss the configuration and usage of LLMs by TestPilot such as synthetic data generation, fine-tuning, prompt engineering, and embeddings.
- Chapter 4: Evaluation
    - This chapter will present metrics to evaluate TestPilot such as the compilation rate, test accuracy rate and documentation rate.
    - This chapter will discuss the performance of TestPilot in terms of cost, correctness, and education.
- Chapter 5: Conclusion
    - This chapter will reflect on the development process for TestPilot.
    - This chapter will discuss the successes and future work that may proceed from this dissertation.

2 Background

It has been established that standard online searches for software testing education is not suitable for novice programmers. Tests found online may be inconsistent, incorrect, or non-educational due to insufficient documentation. Furthermore, LLMs face the same issues as standard online searches. LLMs are trained on data that might be written by experienced programmers, which results in complex tests. LLMs may struggle to contextualize a novice programmer's messy beginner code. Furthermore, LLMs face hallucinations which cause incorrect tests.

Since standalone LLMs are not enough, LLMs need to be adapted in some way to help novice programmers learn software testing. To accomplish this task, this chapter will establish the technical background behind how LLMs are designed and how they can be subsequently adapted. Furthermore, this chapter will discuss the current literature on generative artificial intelligence for software education to show strong reasons why TestPilot would be beneficial for novice programmers.

2.1 Technical background

LLMs are complex, yet powerful. In fact, many of their inner workings have not yet been understood fully by researchers, since it is difficult to "give a narrative description" for their overall behaviour [17]. LLMs use artificial neural networks as their underlying architecture that calculates and adjusts numerical activation values within their internal components, loosely related to human neurons [18]. These artificial neurons contain hundreds of billions of connections which can be used multiple times while processing text. The magnitude of these connections, along with their invocation at various times while processing text, removes any single definite explanation for their overall behaviour.

2.1.1 Transformer architecture

The success of LLMs being applied to our daily lives is due to our fundamental understanding of how they are trained and subsequently tweaked at inference. LLMs are pre-trained on a dataset during the training procedure, usually amounting to billions or trillions of tokens. Parameters exist within the model, which are weights that are adjusted through optimization algorithms such as stochastic gradient descent to minimize the differences between the input and output of a model. This difference is usually measured using a loss function. Models can leverage the transformer architecture, consisting of several similar layers stacked on top of each other. Each layer has an input and output, where the output of $layer_{n-1}$ is the input of $layer_n$. This architecture was used in the very first

GPT (Generative Pre-training Transformer) released by OpenAI in June 2018 [19], which used the following loss function:

$$\mathrm{L}_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta) \tag{1}$$

where $\mathcal{U} = \{u_1, \ldots, u_n\}$ is an unsupervised corpus of tokens and k is the size of the context window where the conditional probability is modelled with parameters Θ. This loss function is also known as the log-likelihood. Furthermore, the softmax function calculates the probability of each token being the next token in the sequence, with the summation of all probabilities equalling 1. This is used to calculate $P(u)$, the probability of each token u in the vocabulary being the next token in the sequence:

$$P(u) = \mathrm{softmax}(h_n W_e^{\mathrm{T}}) \tag{2}$$

where n is the number of layers, h_n is the final hidden state (the output of layer n) and W_e^{T} is the transpose of the token embedding matrix. During training, the input sequence is embedded and split into substrings of length k. Iterating through each substring, the model predicts the next token by calculating the output probability distribution using the final softmax layer.

Each token in the probability distribution is mapped to the probability that the next token is the actual next token in the subsequence. For each substring, the model outputs a probability distribution, where the probability of the true token in the sequence is used for loss calculation. Subsequently, the final loss is equal to the sum of the logarithms of these true probabilities. The resulting log-likelihood loss function maximizes the logarithm of the probability of correctly predicting the next token in a sequence given the previous tokens.

2.1.2 Effects of sampling and temperature adjustment

The sampling used when choosing the next token from the probability distribution can be changed. Greedy decoding involves choosing the word with the highest probability as the next word. Beam search involves generating various possible sequences and using a beam width parameter to choose how many sequences are kept at each step. Top-p sampling involves selecting tokens until the cumulative probability mass of the selected tokens in the probability distribution exceeds a threshold p.

However, the probability distribution itself can be adjusted to produce more deterministic or random outputs on behalf of a novice programmer. Logits refer to raw unnormalised scores produced by the last layer of a transformer before applying a final softmax consisting of a vector $z = [z_1, ..., z_n]$. The temperature parameter T of a model scales the logits by dividing the original logits by T, resulting in a vector $z' = [z'_1, ..., z'_n]$. After this scaling, this is passed to the softmax function to produce the final distribution in the form of $p = [p_1, ..., p_n]$:

$$z'_i = \frac{z_i}{T} \tag{3}$$

$$p_i = \frac{e^{z'_i}}{\sum_j e^{z'_j}} \tag{4}$$

Therefore, lower temperature values where $0 < T < 1$ result in sharpening the probability distribution. This leads to high-probability tokens being more likely to be sampled and subsequently more deterministic outputs. Higher temperature values where $T > 1$ result in flattening the probability distribution. This leads to lower-probability tokens being more likely to be sampled, and subsequently more creative responses that would be out of the ordinary.

$T = 0$ is not typically used since division by 0 leads to mathematical errors. Hypothetically, using $T = 0$ would lead to scaled logits z'_i approaching a positive infinity if they had a positive sign and negative infinity if they had a negative sign. When this is passed to the softmax function, it would always choose the token associated with the highest logit, making all other alternatives impossible.

Furthermore, using $T = 1$ keeps the logits unchanged, since division by 1 results in the original probabilities being used for sampling. Nondeterministic responses might be creative, but often result in code that is not syntactically correct or uses libraries that are out of scope in the provided context. For code generation, using $T = 1$ is not necessarily optimal for determinism and subsequently a higher chance of code that compiles and runs. Using $T = 0$ does not guarantee deterministic responses, although it is better than $T = 1$ which uses the original probabilities for sampling [20].

2.1.3 Fine-tuning

If the performance of a pre-trained LLM in response to a particular task needs to be improved, fine-tuning is an approach that involves teaching the model using a small dataset containing examples relating to the task while maintaining its general knowledge. This helps to teach the model to behave in a particular way, which can be useful for code generation in a particular style. The first GPT was fine-tuned to adapt parameters Θ using a dataset $\mathcal{C}$ where each entry consists of a sequence of input tokens $x_1, ..., x_m$ labelled with y [19]:

$$\mathrm{L}_1(\mathcal{C}) = \sum_{(x,y)} \log P(y|x_1, \ldots, x_m) \tag{5}$$

Language modelling as an auxiliary objective involves the task of predicting the next word in a sequence of text as an additional objective alongside the primary supervised learning task. Implementing this while fine-tuning helps the model generalize better and accelerate convergence, the point at which the model's performance stops improving further. This was used in fine-tuning the first GPT: [19]:

$$\mathrm{L}_3(\mathcal{C}) = \mathrm{L}_2(\mathcal{C}) + \lambda * \mathrm{L}_1(\mathcal{C}) \tag{6}$$

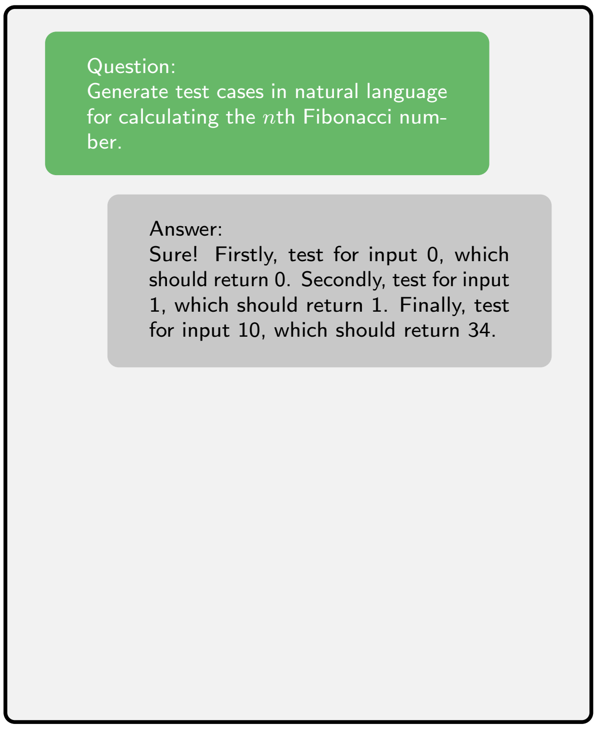
where $\mathrm{L}_2(\mathcal{C})$ is the primary task loss and $\lambda * \mathrm{L}_1(\mathcal{C})$ is the language modelling loss, with hyperparameter λ controlling the importance given to the language modelling compared to the primary task. Learning to predict the next word in a sequence helps the model generalize better with new data for the primary task while reaching convergence quicker through additional signals from language modelling.

Fine-tuning is not suited for improving the knowledge base of a model, but instead focuses on improving the intended behaviour and style of responses. Models fine-tuned to a single task may experience a phenomenon known as catastrophic forgetting, where the weights of an LLM are adjusted to the point where it suits the single task but fails to generalize outside the task by forgetting previously learned knowledge while learning new knowledge [21]. Thus, avoiding overfitting by using a relatively small but carefully chosen fixed set of examples for accurate generalization is key.

2.1.4 Prompt engineering

At inference, LLMs are prompted using a variety of prompt engineering techniques to improve the effectiveness of their responses. Language models are few-shot learners [22], therefore few-shot prompting is a commonly known technique for effective responses. This property of language models first appeared when they were scaled to a sufficient size [23]. Few-shot prompting involves providing the model with a few examples in the prompt to enable in-context learning.

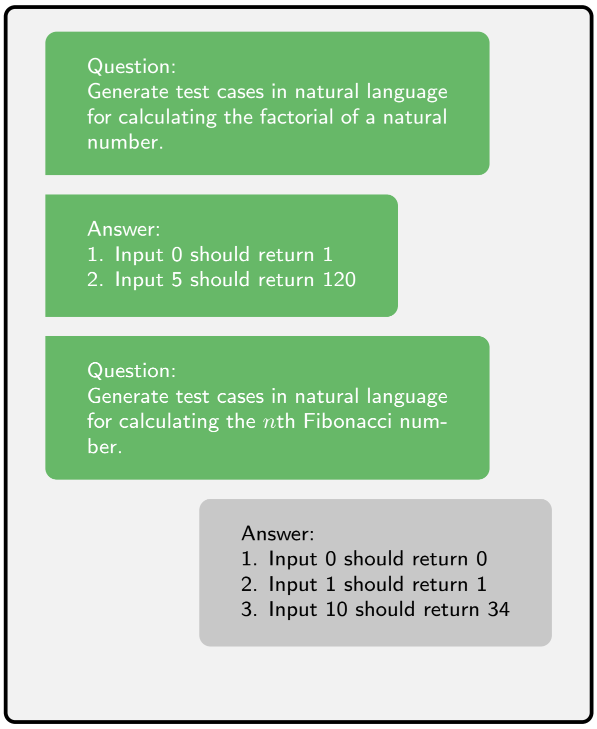
Few-shot prompting is beneficial when an ideal answer requires context from an example. Few-shot prompting relieves the need to fine-tune a model to respond in a particular manner or format which can be cost-effective. The pre-trained model is aware of patterns within language, which makes providing examples inside the prompts effective at helping the model generalize better to new prompts. Zero-shot prompting involves providing no examples but relying on the model to understand the underlying sentiment of a prompt.



Question:
Generate test cases in natural language for calculating the nth Fibonacci number.

Answer:
Sure! Firstly, test for input 0, which should return 0. Secondly, test for input 1, which should return 1. Finally, test for input 10, which should return 34.

Figure 2.1: Zero-shot prompting.



Question:
Generate test cases in natural language for calculating the factorial of a natural number.

Answer:
1. Input 0 should return 1
2. Input 5 should return 120

Question:
Generate test cases in natural language for calculating the nth Fibonacci number.

Answer:
1. Input 0 should return 0
2. Input 1 should return 1
3. Input 10 should return 34

Figure 2.2: Few-shot prompting.

However, the model might struggle to generalize to new prompts that require extensive reasoning capabilities. To combat this, chain-of-thought prompting is a prompting technique which involves providing intermediate reasoning steps inside the prompt [24]. This technique is commonly combined with few-shot prompting to maximize the chance of a response that follows the requested structure and satisfies complex reasoning with minimal hallucinations.

Prompt engineering techniques are useful in combination with specific instructions such as "Let's think step by step" which drastically increases the accuracy of responses in a zero-shot setting [25]. As LLMs are trained with more high-quality data, their responses will improve and subsequently not require specific prompts. But until then, leveraging their similarity to human qualities (through their pre-training with data closely related to human interaction found online) is a valid approach to maximize the efficiency of responses.

2.1.5 Embeddings

To improve the knowledge base of an LLM, embeddings are a valuable tool to use. Embeddings are dense vectors stored in an index in a vector database that are mathematical representations of words in a high-dimensional space. Subsequently, relevant data can be added to a prompt to improve the knowledge base of an LLM without relying on information from its training data. To find the relevant data, the initial prompt is also embedded into a vector to use similarity metrics such as the cosine similarity for finding the data most similar to the initial prompt:

$$\cos(\theta) = \frac{\mathbf{A}\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} \mathbf{A}_i \mathbf{B}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{A}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{B}_i)^2}} \tag{7}$$

where A and B are vectors representing an answer in a vector database and the prompt respectively. Subsequently, the most similar answer in the vector database can be added to the initial prompt in a similar structure to few-shot prompting. This improves the knowledge base of an LLM without requiring the LLM to contain the information in its training data. By storing related data in a vector database, embeddings offer improved responses that use related information in their output.

Converting data to a vector for embeddings captures the semantic meaning of text, unlike lexical information retrieval which looks for exact words from the query. Using a standard database with lexical retrieval would limit any additional knowledge to contain the exact words as in the prompt. Conversely, the semantic meaning from dense retrieval methods helps the model generalize to different but similar problems it contains in its knowledge base.

2.1.6 Scaling laws

The effectiveness of an LLM is not purely based on the number of parameters that exist within the model, but also the volume and quality of data that it is trained on. A model trained with a low amount of data and a high number of parameters can be overfitted and subsequently struggle to generalize with unseen data. A model trained with a high amount of data and a low number of parameters can be underfitted and subsequently struggle to understand complex relationships in the data. Finding a Pareto frontier in this area for the correct balance between the parameters and data of an LLM for has been an ever-changing research problem.

The Kaplan scaling laws introduced by OpenAI stated that a data-to-parameter ratio of around 1.7 is optimal, with 300 billion tokens being used to train an LLM with 170 billion parameters [26]. After this discovery, Chinchilla scaling laws introduced by DeepMind stated that a data-to-parameter ratio of around 20 is optimal, with 1.4 trillion tokens being used to train an LLM with 70 billion parameters [27]. However, it has been shown that high quality data pruning can improve these scaling laws further [28]. This is currently an open research problem which will improve over time.

2.1.7 Synthetic data

The principle of high quality data from scaling laws is the same with synthetic data that is generated for fine-tuning. Synthetic data is generated artificially from LLMs to emulate real-world data. Synthetic data is often used for fine-tuning because real-world data is hard to find at scale [29]. Synthetic data can be generated by an LLM using prompt engineering to receive data in a particular style for fine-tuning. However, raw unvalidated synthetic data from an LLM is unreliable. Since LLMs are inherently unreliable themselves, using unvalidated data to fine-tune an LLM compounds the unreliability.

Grounding, taxonomy-based generation and filtering are proposed techniques used in synthetic data generation by researchers to minimize unfaithful data [30]. Grounding provides real-world examples in prompts that act as seeds for inspiring the model to return diverse data. Taxonomy-based generation includes theorizing k-ways the ideal data can be identified under a classification to sample across k of these approaches, and subsequently asking an LLM to generate data according to one of the classifications. Finally, filtering consists of fine-tuning an LLM to know the difference between synthetic and real-world data and prompting it with grounding data to remove ones that are synthetic.

When these techniques were proposed, grounding proved to be a key element to synthetic data generation. However, taxonomy-based generation struggled to stay true to the underlying classification, since it assumes a uniform distribution over proposed classifications when it might not be the case. Furthermore, filtering struggles to classify whether the data is synthetic or real since it requires classifying data that may be "up for debate" with no single definitive answer.

Choosing from these synthetic data generation techniques based on the type of data required is essential. For generating novice-level code, grounding may involve providing a function from a real-world repository and using it as a seed focal method for an LLM to simplify into a simple novice-level implementation with a similar meaning to the original. Taxonomy-based generation may be skipped to avoid assuming a uniform distribution over proposed classifications. Synthetic data can be verified through use case specific filtering by using compilation and code coverage checks, instead of fine-tuning an LLM to do this operation.

2.2 Literature review

The use of LLMs for teaching software testing to novice programmers is a relatively new concept. However, there is a significant body of research on the use of LLMs in programming education. The use of this literature is important for developing TestPilot, since it serves as a chance to acknowledge the advantages, disadvantages, and opportunities available when bridging the gap between a novice programmer and an LLM.

This section will outline the current literature on the use of generative artificial intelligence by novice programmers. It will discuss research on the usage of OpenAI's Codex, ChatGPT, GPT-3.5 Turbo, and GPT-4 by novice programmers. Finally, this section will summarise the key points from this literature to use when developing TestPilot.

2.2.1 Novice programmers using Codex

OpenAI's Codex was released in 2021 which is a GPT language model fine-tuned on publicly available code from GitHub [31]. Knowing what types of questions a novice programmer typically asks an LLM is important, as it helps researchers improve responses for those types of questions. A study was conducted asking introductory programming learners to use OpenAI Codex to complete code-authoring tasks, which subsequently allowed researchers to note what aspects of programming invoked an LLM inference by these novice programmers [32]. It was discovered that loops and arrays were the most common topics that made students prompt an LLM, with 84% and 85% of inferences relating to those topics respectively.

When novice programmers used OpenAI's Codex to ask a question involving arrays and loops, students copied the highest proportion of the original question. This suggests that novice programmers are inclined to paste more of the original question into the prompt when they struggle with the concepts in it; 42% of the original question was used for loops and 48% of the original question was used for arrays. From this observation, it can be concluded that novice programmers who use OpenAI's Codex struggled with arrays and loops the most, which surprisingly also correlated with the highest amount of copying and pasting from the original question compared to any other concept.

On the subject of prompting, the study highlighted that students with more experience had an average score of 90% compared to students with less experience who had an average score of 71% when using LLMs. Furthermore, the students suggested that they needed to write prompts with a high level of detail to get the correct response they required, for which they would rather code the answer themselves. From this, it can be concluded that people with more experience tend to benefit the most from LLMs since they are more likely to include the correct information in the prompt since they know what to ask for. Conversely, novice programmers often struggle to know what to ask in the first place since their foundational knowledge is lower.

It is important that LLMs are not just a temporary band-aid to a novice programmer's current problem. It was examined that students with previous programming knowledge also did better at a retention test a week after compared to students with less experience. However, students with less experience still did better with OpenAI's Codex than without it. This highlights that novice programmers are inclined to use LLMs as a temporary fix rather than for long-term retention, since it boosts their ability temporarily.

The output of an LLM has to be in a format that is digestible by a novice programmer. The study highlighted that students become overwhelmed by an LLM that outputs a large section of code. Therefore, breaking the output into multiple segments would be a viable solution to this problem, forcing students to work on each segment before moving on to the next. Furthermore, the study mentioned that additional documentation and worked examples accompanying the responses would have improved the experience for novice programmers in this regard.

Regardless of this disadvantage, some students who used OpenAI's Codex reported that they felt less stressed since it "reduced pressure" which may have help foster a better learning environment. On the contrary, some students did not like that the LLM returned the entire answer at times "instead of step-by-step hints to the user". This hindered their learning process, since it did not let the learner think about the problem. Finally, students did not like that the LLM did not always give proper explanations on "why the generated code was what it was". Therefore, it is evident that a lack of explanations with responses was a hindrance for novice programmers.

The study concluded that the usage of LLMs in programming support tools could scaffold learning for novice programmers in the future. From novice programmers using OpenAI's Codex in this study, it is clear that novices improved their ability temporarily but there are challenges that need to be fixed in the future with correct prompting, retention of knowledge, and LLM output formats.

2.2.2 Novice programmers using ChatGPT

OpenAI's ChatGPT was released in 2022 and is a sibling model to InstructGPT [33], a model that was trained to follow instructions using human feedback. While ChatGPT was not fine-tuned for programming in particular, research has been conducted to study its effectiveness for novice programmers. A novice programmer requires an LLM that is highly accurate, since they require a source of truth to test their own beliefs (which may be right or wrong). An LLM that is inaccurate is not suitable for a novice programmer, since the LLM or the novice programmer could be right, wrong or both, with no concrete source of truth to rely on.

A study reviewed the accuracy of ChatGPT for questions from a popular software testing curriculum taken by novice programmers [34]. ChatGPT was only able to respond to 77.5% of questions given. This was an interesting observation which reaffirms that LLM alignment is a major issue for LLMs teaching novice programmers. This also highlights a need for a system that always produces an answer to a question from a novice programmer that is highly accurate.

From the questions that ChatGPT did answer, it only had a 55.6% chance of being correct or partially correct, and a 53% chance of having a correct or partially correct explanation. The main types of incorrect answers were due to poor knowledge, assumptions, or both. Furthermore, the responses that seemed confident had little impact on the actual accuracy of the response, which highlights the issue of hallucinations from LLMs teaching software testing to novice programmers.

In the last subsection, it was established that loops and arrays were the most difficult concepts for novice programmers. In this study, it was shown that ChatGPT performed the worst with questions involving both code and concepts with an accuracy of 31.3%.

This reveals an important feature about LLMs for teaching testing to novices, which is that LLMs are moderate at generating code but struggle with conceptual questions that require higher levels of reasoning. Arrays and loops require reasoning that is hard to capture in the parameters of a model, since not every permutation and combination of loops and arrays are captured in the training data. ChatGPT will make an educated guess based on trends, but this study proves that this is not suitable for a novice programmer.

A reasonable follow-up question would be "If the accuracy of ChatGPT isn't great, does it still improve the skills of a novice programmer?". To answer this, another study reviewed the impact of ChatGPT on the computational thinking skills, self-efficacy and motivation of undergraduate students who took a programming course [35]. The opinions of students were recorded while being given ChatGPT during weekly programming exercises using a likert-type system. Surprisingly, their results show that university students believed ChatGPT helped improve their algorithmic thinking, self-efficacy, and problem-solving skills for complex programming tasks more than students who did not use ChatGPT.

This leads to an important observation, which is that although the accuracy of ChatGPT is not high for novice programmers, novices believe that it helps their ability to program. This raises a concern that novice programmers may experience a false sense of ability without the privilege of such a tool at all times. This observation suggests that novice programmers may rely on LLMs so much that their notion of their programming ability is constructed under the assumption of an LLM by their side, which may be for better or for worse!

Although computational thinking and self-efficacy were improved, the motivation of students who were allowed to use ChatGPT was similar to students who were not when given difficult questions. It was already established in the previous study that ChatGPT struggles with conceptual questions that require higher levels of reasoning. Therefore, it makes sense in this study that the motivation of novice programmers does not increase with complex questions, since complex questions require conceptual reasoning that ChatGPT does not work well with. Incorrect answers from complex questions may be the reason there is no increase in motivation when using ChatGPT for complex questions.

2.2.3 Novice programmers using GPT-3.5 Turbo and GPT-4

OpenAI's GPT-3.5 Turbo and GPT-4 [36] were released in 2023 which followed from GPT-3, however they had better conceptual reasoning abilities. This rapid improvement in understanding influenced researchers to compare it to OpenAI's Codex for use by novice programmers. A study tested the accuracy of GPT-3.5 Turbo against OpenAI's Codex for identifying and explaining the issues in student's code from an online programming course [37]. It was shown that GPT-3.5 Turbo was able to was able to detect all issues 55% of the time compared to only 15% from Codex. This displayed that GPT-3.5 was much better at conceptual reasoning for novice programmers, which was a major issue for LLMs up until then.

While there was an increase in conceptual reasoning, GPT-3.5 Turbo was still not perfect. For example, the study showed that GPT-3.5 Turbo only had a 44%, 54% and 50% success rate at finding all issues related to formatting, unwanted outputs, and missing outputs respectively. This highlighted that LLMs can struggle to "wrap" around an initially incorrect solution with a bug in it, and will subsequently provide an incorrect solution to fix it as a consequence. A common occurrence of this in the everyday life of a programmer may include calling methods or referring to objects using the "`this`" identifier in Java, which may lead to the LLM either making up a new function name or not using the correct identifier in the response.

An interesting observation in the study was that GPT-3.5 Turbo reported imaginary problems 48% of the time when asked 150 questions with help requests, such as mentioning that an extra curly bracket exists when in reality it did not. Furthermore, GPT-3.5 Turbo had a mere 35% accuracy at detecting all issues for problems with logic errors that use conditionals. This emphasized that GPT-3.5 Turbo was a much better LLM for novice programmers than anything on the market at the time, but it still could be adapted further in some way for novice programmers to get accurate results.

On the same topic of fixing errors, the opinions of novices who used GPT-4 to provide hints when faced with compiler errors were recorded in another study [38]. An intriguing observation was made, which was that 56% of students believed the hints were very or extremely useful. Conversely, 20% of students believed the hints were not useful at all. This observation reaffirmed that GPT-4 was perceived as useful, but there is a "need for further improvement of their relevance and accuracy" for novice programmers, which could be completed with "refined prompt engineering or fine-tuning of the AI model to better address diverse needs of students".

The study restated the issue of GPT models struggling with errors that required a deeper conceptual understanding. In fact, students that had access to hints from GPT-4 performed slightly worse than the control group who did not have access to it. This observation emphasized the importance of being careful when giving novice programmers access to LLMs, since there is a potential for novice programmers to do worse with them. This suggests that outsourcing different types of logic in some way might decrease the chance of this happening. An example of this might be delegating an entire task that requires a deep conceptual understanding away from such an LLM, and instead giving more straightforward and confined tasks to various LLM inferences instead.

Apart from accuracy, the study showed that having hints from GPT-4 decreased the confusion and frustration of novice programmers when stuck compared to the control group. It has been established in the previous subsection that having access to ChatGPT did not increase the motivation of students. This behaviour is repeated with GPT-4 in this study, which stated that boredom and anxiety did not change regardless of access to GPT-4 or not. These observations conclude that it is indisputable that LLMs are helpful aids for novice programmers in the face of learning (which can be confusing and frustrating), however the motivation and drive of a novice programmer is difficult to improve with LLMs.

2.2.4 Implementations of LLMs for novice programmers

After studying the use of various LLMs by novice programmers, researchers noted the observations and opinions of novices when implementing LLMs into various interfaces. This subsection will examine related interfaces to TestPilot which use GPT-3.5 Turbo and GPT-4 to help novice programmers. These related interfaces implemented feedback from previous observations and opinions, which can subsequently help the development of TestPilot by learning from their successes and mistakes.

GILT

GILT is a Visual Studio Code extension that helps with code understanding [39]. GILT stands for "Generation-based Information-support with LLM Technology for the Python programming language". It uses GPT-3.5 Turbo to create a conversational user interface in the sidebar of Visual Studio Code to help novice programmers understand their code.

As seen in previous literature, novice programmers struggle to write good prompts since they do not always know what to ask for. GILT does not require a novice to input explicit prompts, and instead offers four high-level requests such as code explanations, details of API calls, explanations of domain specific terms, and usage examples of an API. The researchers who developed GILT took a prompt-less approach which leveraged the fact that novice programmers are not good at prompting themselves. As an alternative, GILT also accepts open-ended prompts that are automatically contextualized by passing the selected code from a novice into the prompt to GPT-3.5 Turbo.

It was established in the previous subsections that novice programmers do not like a big chunk of text when asking an LLM for help. The researchers who developed GILT used a need-based explanation approach, where an explanation is only generated if a user requests it. By doing this, a user is not bombarded with a long paragraph of text which is not digestible and includes information they did not ask for. This reduces distractions and information overload for the novice programmer.

Researchers compared GILT with web search in a study using the technology acceptance model (TAM) [40], and found that novice programmers rated it more useful with a TAM score of 33.49 compared to 27.3 for standard online searches. Novices also believed it was easier to use with a TAM score of 34.2 compared to 29.75 for standard online searches. This evaluation concluded that an interface which does not focus on detailed prompts increased the usefulness and ease of use of the interface for novice programmers.

In the paper for GILT, the researchers discussed the implications of GILT and the key principles to follow for future researchers attempting a similar approach. Novice programmers who used GILT mentioned that they frequently did "comprehension outsourcing" where they did not bother understanding the code before prompting the LLM. This reaffirms the statements from the previous subsections that novice programmers are inclined to use LLMs as a temporary fix rather than for long term retention. Although GILT used a prompt-less approach which makes it easier to understand code, the over-reliance of LLMs by novice programmers was still a threat to validity for GILT.

Researchers also mentioned that more work is needed for the user interface. Prompt-less interaction integrated into an IDE was successful compared to online searches, but it was evident that further research is needed to accommodate novice programmers who have different learning styles, such as using a button-based or prompt-based approach to generate explanations. Furthermore, the researchers stated that using more context from the code of a novice will improve the generated responses. This may involve using system context such as programming languages and libraries in the prompt to alleviate prompt engineering efforts elsewhere.

The researchers concluded that prompt-less interactions in an IDE with LLMs is a promising future direction for tool builders. This reaffirms that designing interfaces that reduce the need for explicit prompts from a novice programmer is a valid approach.

Scratch Copilot

Scratch Copilot is a plugin for Scratch which provides code explanations for novice programmers using GPT-4 [41]. Scratch is a visual programming language which helps novice programmers learn to code [42]. It was aimed at programming education for families getting into programming for the first time.

The researchers stated that Scratch Copilot was able to support middle schoolers in understanding Scratch projects. The researchers found that Scratch Copilot was able to explain code 90% of the time for 40 code explanations. Since the Scratch programming language uses blocks of logic instead of normal text like Java, GPT-4 was able to create appropriate metaphors for complex computational concepts such as loops or variables.

Debugging was achievable by Scratch Copilot with an accuracy of 80% for 40 debugging examples. It was established in the previous subsections that GPT-3.5 Turbo reported imaginary problems, and similar issues remain with GPT-4 as well. Researchers mention that Scratch Copilot occasionally suggested creating variables when the actual issue was from conditionals. This highlighted that there would always be a chance that responses will be incorrect, but the odds for novice programmers were much better using GPT-4.

Another issue with Scratch Copilot was that the LLM repetitively suggested the same ideas which did not encourage creative thinking. Managing creativeness and correctness is a difficult challenge. As mentioned in the first section of this chapter, temperature adjustment can change the creativeness of LLM responses which may solve this issue. Researchers stated the importance of future models expressing when they are wrong, which aligns with the literature on ChatGPT in the previous subsections, which state that confident answers do not correlate with the actual accuracy of the response. Since LLMs are not aware of their inaccuracies yet, this is a major difficulty for novice programmers trusting them as a source of truth.

In the paper for Scatch Copilot, the researchers presented design guidelines for future AI-enhanced coding tools. In the previous subsections, current literature suggests that models like ChatGPT already promote self-efficacy. However, the researchers who developed Scatch Copilot suggested the promotion of self-expression, meaning that LLMs should pose strategic questions to novice programmers instead of providing direct answers. Models promote self-efficacy since they make novice programmers more independent, but self-expression should also be a key part of any interface used by novice programmers. This relates to the observation that novice programmers do not want a chunk of text as a response from LLMs.

The researchers also suggested that visual elements along with text would help novice programmers locate specific programming blocks. The use of an interactive user interface that includes visual aids is important since it prevents a novice programmer relying on only the generated code which may overwhelm them. This may include progress bars, buttons and cards which provide multi-modal support to a novice programmer.

Finally, the researchers emphasized the importance of adapting LLMs to be "tailored to the coder's experience level to maximize learning outcomes". It was established in the previous chapter that LLMs are not trained on data that is suitable for a novice programmer. Therefore, this statement from the researchers suggests that future work needs to adapt LLMs in some way to make their responses at a novice-level

VSCuda

VSCuda is a Visual Studio Code extension that offers CUDA syntax highlighting, code help for CUDA APIs, code completion for CUDA functions, and code improvement suggestions [9]. CUDA is a programming language that uses the GPU [43]. VSCuda used GPT-4 to help novice programmers code with CUDA.

The researchers experimented with calling GPT-4 and reprompting if the provided solution was incorrect when compiled for up to five reprompts. This approach was a potential solution for the lack of accuracy by LLMs for novice programmers in the previous subsection. It took 2 reprompts on average to converge on a correct solution with GPT-4. This suggested that reprompting was a viable tool for increasing the accuracy of responses after completing a sanity check through the compilation of the response before the novice sees it.

The researchers also tried to use GPT-4 without reprompting, but instead asked it to make improvements separately through individual inferences instead of all at once. When asked a series of questions about suggested improvements, GPT-4 performed better with individual optimizations. The researchers concluded that the performance of GPT-4 was degraded when performing optimizations that were compounded but stayed stable when performing optimizations individually.

The researchers performed a cost analysis with GPT-4 and found that it was a relatively expensive model, with \$0.03/1000 tokens for input and \$0.06/1000 tokens for output using the 8K context model. They stated that this was at least 10 times more expensive than Llama 2 by Meta [23] which ranged from \$0.001 - \$0.002/1000 tokens. However, experiments showed that Llama 2 required 3 reprompts on average compared to 2 from GPT-4. When not using reprompts, Llama 2 was unable to perform any optimization that was applied in isolation, compared to GPT-4 which was able to do so. Therefore, the researchers chose GPT-4 as the LLM of choice for VSCuda.

The researchers designed the API calls to GPT-4 to be completed after each file save, so that the model could identify issues with the code from the file currently open in Visual Studio Code. Context was passed into the prompts such as thread divergence, accessed to global memory, and computation intensity. This was an expensive route since GPT-4 inferences are expensive, but this provided a responsive user interface that found issues in code quickly.

VSCuda also leveraged the benefits of implementing LLMs straight into an IDE, since it provided suggestions as `"diff"` files which were shown in the front-end Visual Studio Code extension. This was innovative since it was a non-evasive approach to provide suggestions in the background, instead of big chunks of text which were disliked by novice programmers as mentioned in the previous subsections.

For future work, the researchers who developed VSCuda stated that fine-tuning an existing LLM would help the model recognize issues in CUDA code much better. This adaptation of LLMs for novice programmers was previously mentioned in the literature for GPT-4 in the previous subsection. This highlights the importance of fine-tuning when developing LLMs for novice programmers since it is not only mentioned in the literature as a possible solution to make the responses at a novice-level, but also in real-life implementations from researchers who say it can improve the effectiveness of responses.

2.2.5 Summary

From the current literature on the use of generative artificial intelligence by novice programmers, there are many lessons that can be used for the development of TestPilot. This subsection will summarise the key points from the literature review and map them to concepts that TestPilot will implement.

Novice programmers ask LLMs about loops and arrays the most. When receiving a response, LLMs struggle the most with prompts that include code and concepts that require higher levels of reasoning, such as from questions involving loops and arrays. Novice programmers struggle with conditionals that require a deep understanding of logic, yet default LLMs are the worst at answering questions that require higher levels of reasoning. Therefore, TestPilot should increase its knowledge base to have a better chance at answering correctly. This can be done with a vector database that has known good working solutions.

When novices prompt with concepts they struggle with, they had the highest chance of copying and pasting text from the original question, promoting "comprehension outsourcing" which is not educational. Furthermore, novices who are inexperienced struggle to prompt since they do not know what to ask in the first place compared to an experienced programmer. Therefore, TestPilot should use an input system that does not require explicit prompts that are detailed. Vague descriptions should be enough of a requirement.

Novice programmers struggled with the retention of gained knowledge when using LLMs. Novice programmers were inclined to use LLMs as a short-term fix since it boosted their ability temporarily. Therefore, TestPilot should have a mode of operation that focuses on testing the beliefs of a novice, instead of only generating a suite of tests that always pass. When a novice is forced to think critically in this way, they are encouraged to examine their beliefs with tests which may pass or fail, instead of generating tests which always pass for the entire program space.

Large chunks of text in LLM output are not digestible by novice programmers. Therefore, TestPilot should implement test cases in concise and compact cards which are not overwhelming. Furthermore, novice programmers wished that LLMs used consistent documentation in their responses. Therefore, TestPilot should include documentation in the form of comments above each function which describes the implemented test case. The use of visual aids helps novice programmers focus on specific blocks of code better. Therefore, TestPilot should have progress bars representing code coverage and case coverage.

Although LLMs decrease the frustration and stress experienced by novice programmers, it is evident that models such as ChatGPT do not always respond to questions given. When they do respond, they are not highly accurate since they face hallucinations and struggle to "wrap" around messy beginner code. Therefore, TestPilot should use refined prompt engineering and fine-tuning to improve the chance that the model responds correctly. Reprompting the LLM is a valid solution if the model does not answer correctly.

Finally, research suggests that dividing up the overall logic in a task into smaller individual segments over multiple inferences is a valid approach to improve LLM efficiency for novices. An LLM can struggle with completing many tasks in one inference (such as generating tests for many test cases in the case of TestPilot). Therefore, outsourcing each test case to an LLM inference is a valid approach as long as it is cost-effective.

3 Design

It has now been established that current literature also agrees that novice programmers face issues when using various types of LLMs. Online searches are not viable, and default LLMs are not sufficient for novice programmers either. However, the literature also provides various improvements which can be used for TestPilot. This chapter will outline the design of TestPilot, which uses the statistics and opinions from the current literature to create a highly educational interface that is suitable for novice programmers.

3.1 High-level system architecture

This section will outline the possible inputs and outputs when a novice programmer interacts with TestPilot, as well as the possible cases with those inputs. It will also provide a high-level system architecture for the two modes of operation in TestPilot. It will discuss how the different modes of operation work, the outsourcing of logic to different parts of the system, and how this can ultimately be useful for a novice programmer.

For both modes of operation, the inputs from the novice programmer include the code they wish to test and a beginner natural language description of a test case. Subsequently, the outputs from the Visual Studio Code extension include the test cases in natural language and the implemented tests in code. This high-level system architecture will discuss the mechanisms in TestPilot that convert these inputs to outputs.

Modes of operation

Initially, TestPilot only had a "discovery" mode of operation which simply generated tests that always pass. After the initial presentation, feedback and discussion suggested an alternative mode of operation for tests that could pass or fail to test the beliefs of a novice. This resulted in an additional "testing" mode of operation.

Furthermore, it is important to acknowledge the four possible cases when a novice programmer inputs their code and description into TestPilot. Case 1 occurs when the code and description are correct. Case 2 occurs when the code is incorrect but the description is correct. Case 3 occurs when the code is correct but the description is incorrect. Case 4 occurs when both the code and description are incorrect.

	Code	Description
Case 1	✓	✓
Case 2	✗	✓
Case 3	✓	✗
Case 4	✗	✗

Table 3.1: Possible cases for inputs being correct (✓) or incorrect (✗).

3.1.1 Testing mode

A novice programmer might be able to write a solution to a problem, but might not be able to know if their code is correct. Although they might struggle with code as a novice, they are less likely to struggle with natural language. Therefore, TestPilot takes a natural language description of a test case and generates a test based on it. The generated test may pass or fail, indicating to the novice if their code is performing as they describe in natural language.

A card is shown in the sidebar showing this new test case, since it has been established that novice programmers do not like large chunks of text. A test file is created which includes the test case from the card, which can optionally be viewed by the novice programmer. By not forcing a novice programmer to look at the entire solution immediately, they are encouraged to explore the problem space and use the generated code as a learning mechanism and not a copying and pasting mechanism. The code coverage of the test case against the novice's original code is shown in a progress bar to maintain a focus on program coverage. This process is shown below in three main steps.

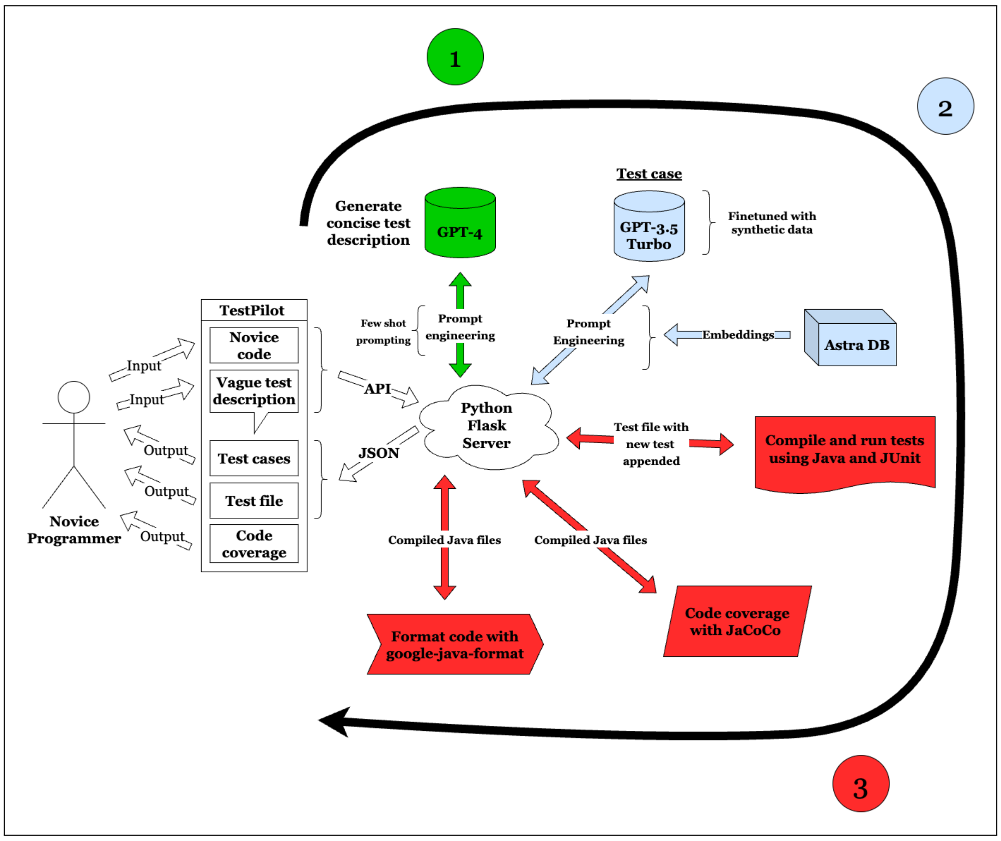
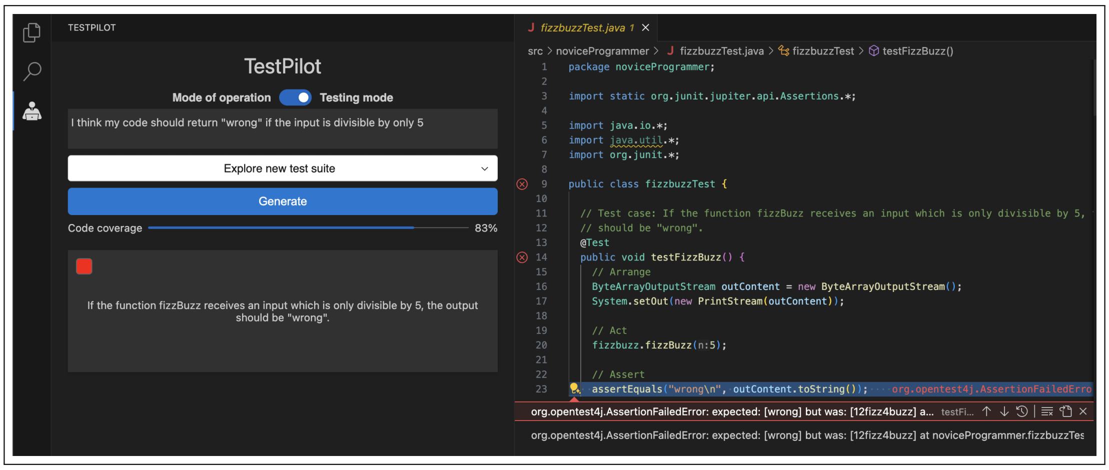


Figure 3.1: High-level system architecture for the "testing" mode in TestPilot.

Step 1: The code and natural language description from the novice programmer is passed by the Visual Studio Code extension to the Python Flask server via an API call. The Python Flask server calls GPT-4 to transform the vague natural language description to a concise test case. This is necessary because a novice programmer's description will be vague and possibly include words such as "I think". It is valuable to note that only short textual descriptions are used for input and output with GPT-4 instead of lengthy code snippets, since it is expensive but accurate compared to other models.

Step 2: After the vague description is sanitized with GPT-4, this test case is passed to a GPT-3.5 Turbo model that is fine-tuned with synthetic data. When prompting GPT-3.5 Turbo, a known good working solution from a vector database is included if it exists which is determined with a cosine similarity. This creates a "@Test" snippet which will represent the test case implementation in code. It is valuable to note that lengthy code generation is outsourced to cheaper fine-tuned models such as GPT-3.5 Turbo.

Step 3: The resulting "@Test" snippet is added to a test file using string manipulation and compiled using Java and JUnit. The status of the test is stored, and code coverage of the test file is calculated with JaCoCo. Finally, the testing file is formatted using google-java-format before it is returned to the Visual Studio Code extension. When returning the test file, the Python Flask server also returns the code coverage which is shown in a progress bar and the status of the tests running which is shown as a green tick or red box in the corner of the card displaying the test case.



I think my code should return "wrong" if the input is divisible by only 5

If the function fizzBuzz receives an input which is only divisible by 5, the output should be "wrong".

```java
package noviceProgrammer;

import static org.junit.jupiter.api.Assertions.*;

import java.io.*;
import java.util.*;
import org.junit.*;

public class fizzbuzzTest {

  // Test case: If the function fizzBuzz receives an input which is only divisible by 5,
  // should be "wrong".
  @Test
  public void testFizzBuzz() {
    // Arrange
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));

    // Act
    fizzbuzz.fizzBuzz(n:5);

    // Assert
    assertEquals("wrong\n", outContent.toString());    org.opentest4j.AssertionFailedErro
```

```
org.opentest4j.AssertionFailedError: expected: [wrong] but was: [12fizz4buzz] a...  testFi...
org.opentest4j.AssertionFailedError: expected: [wrong] but was: [12fizz4buzz] at noviceProgrammer.fizzbuzzTes
```

Figure 3.2: A screenshot of "testing" mode in Visual Studio Code.

After this process is completed, the novice programmer may change the code and rerun it to help them understand where they may have a gap in their understanding. Alternatively, they can start over to generate a completely new test file. Through this mode of operation, a novice programmer is able to test their beliefs using natural language to verify their code does what they think it should.

3.1.2 Discovery mode

A novice programmer might be happy that their solution to a problem is correct, but might not know how to formally test it by structuring it into test cases for the entire problem. By providing a vague description of a test case, TestPilot can generate a suite of test cases for the entire problem and show it in "blurred" cards, with the test case from the description being "unblurred". The novice programmer is encouraged to find all the test cases by inputting more vague test cases to "unblur" more cards.

If the inputted test case is incorrect, no cards will "unblur" to inform the novice programmer that their test case is not valid. A test file is created which includes the implemented code for all the tests that will always pass, which can be viewed if the novice programmer is unsure of more test cases at any point. By not displaying a large chunk of text, a novice programmer is encouraged to explore the problem space as cards get "unblurred". The code coverage of all generated tests ("unblurred" and "blurred") and the case coverage ("unblurred" only) are shown in progress bars to inform the novice if they are missing more test cases. This process is shown below in three main steps.

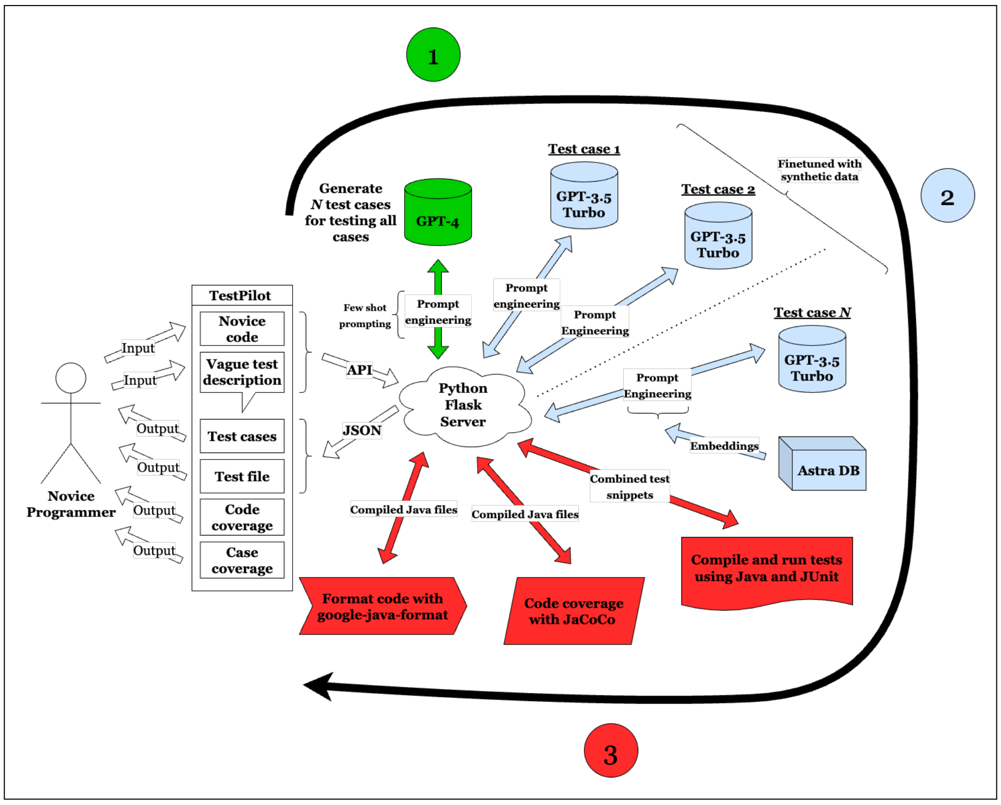
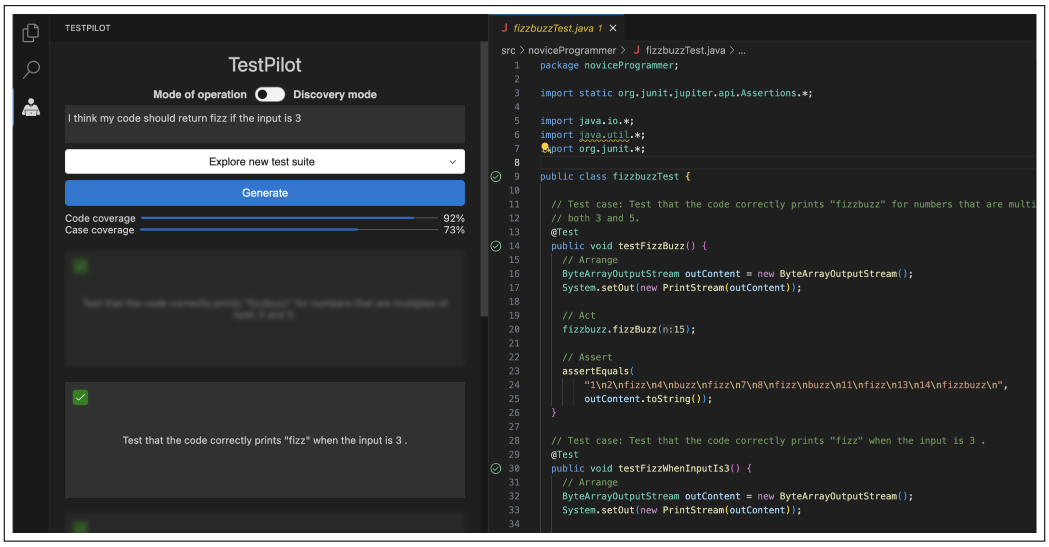


Figure 3.3: High-level system architecture for the "discovery" mode in TestPilot.

Step 1: The code and natural language description from the novice programmer is passed by the Visual Studio Code extension to the Python Flask server via an API call. The Python Flask server calls GPT-4 to generate a list of N test cases in natural language to cover the entire program space. While doing so, it is instructed to put an "(X)" beside the element that relates closest to the novice programmer's vague description. This acts as a starting point to peak the curiosity of the novice programmer; this is the test case which eventually will be shown as "unblurred" to the novice programmer, while the rest remain "blurred". It is valuable to note that only a short description and list of test cases is used for input and output with GPT-4 instead of lengthy code snippets, since it is expensive but accurate compared to other models.

Step 2: After the receiving a list of N test cases with GPT-4, each test case is outsourced to a separate fine-tuned GPT-3.5 Turbo inference. When prompting GPT-3.5 Turbo, a known good working solution from a vector database is included if it exists which is determined with a cosine similarity. Each inference generates a separate "@Test" snippet which will represent the test case implementation in code.

Step 3: The resulting "@Test" snippets are added to a test file using string manipulation and compiled using Java and JUnit. The status of the tests are stored, and JaCoCo is used to calculate the code coverage of the test file and the case coverage of the "unblurred" test cases (which would have an "(X)" representing the novice's description in the list of test cases). Finally, the testing file is formatted using google-java-format before it is returned to the Visual Studio Code extension to eliminate any potential indentation issues from string manipulation. When returning the test file, the Python Flask server also returns the code coverage and case coverage which are shown in progress bars and the status of the tests running which will always show as a green tick in the corner of the card displaying the test case since these tests always pass.



I think my code should return fizz if the input is 3

Test that the code correctly prints "fizz" when the input is 3 .

```java
package noviceProgrammer;

import static org.junit.jupiter.api.Assertions.*;

import java.io.*;
import java.util.*;
import org.junit.*;

public class fizzbuzzTest {

  // Test case: Test that the code correctly prints "fizzbuzz" for numbers that are multi
  // both 3 and 5.
  @Test
  public void testFizzBuzz() {
    // Arrange
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));

    // Act
    fizzbuzz.fizzBuzz(n:15);

    // Assert
    assertEquals(
        "1\n2\nfizz\n4\nbuzz\nfizz\n7\n8\nfizz\nbuzz\n11\nfizz\n13\n14\nfizzbuzz\n",
        outContent.toString());
  }

  // Test case: Test that the code correctly prints "fizz" when the input is 3 .
  @Test
  public void testFizzWhenInputIs3() {
    // Arrange
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));

```

Figure 3.4: A screenshot of "discovery" mode in Visual Studio Code.

After this process is completed, the novice programmer may input more vague test cases to "unblur" more test cases shown as cards, which is completed with another GPT-4 inference to add an "(X)" to the existing list of test cases. Alternatively, they can erase all test cases shown as cards and start over to generate a completely new test file. Through this mode of operation, a novice programmer can discover test cases formally and see code implementations of these test cases. They can interactively unlock test cases by "unblurring" them and can rest assured that they will pass.

3.2 Synthetic data generation

Fine-tuning is an expensive process which requires data that is unique and accurate. Ideal data for fine-tuning requires a wide range of qualities in the data, which also follows the format you require. Scraping data from online is not accurate all the time, but unique data is hard to gather. A solution is to get another LLM to generate this data. This section will discuss how GPT-3.5 Turbo was fine-tuned to create "@Test" snippets using synthetic data that was generated by another LLM. The process of generating synthetic data is shown below.

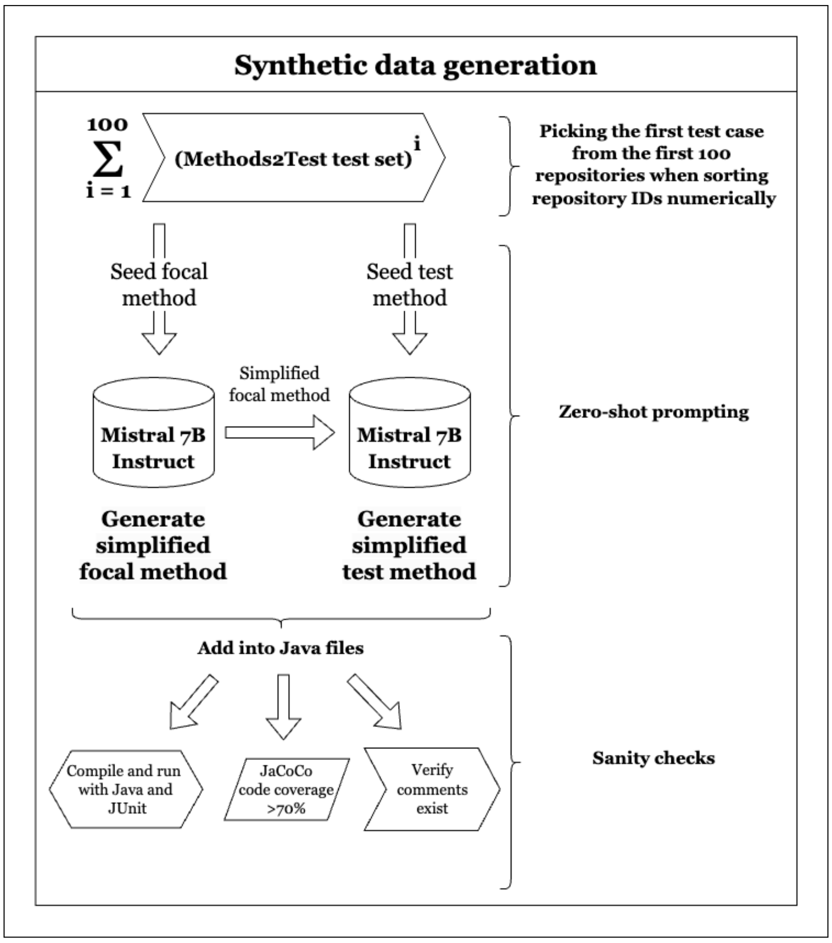

Synthetic data generation
$\sum_{i=1}^{100}$ (Methods2Test test set)^i
Picking the first test case from the first 100 repositories when sorting repository IDs numerically
Seed focal method
Seed test method
Simplified focal method
Mistral 7B Instruct
Mistral 7B Instruct
Generate simplified focal method
Generate simplified test method
Zero-shot prompting
Add into Java files
Compile and run with Java and JUnit
JaCoCo code coverage >70%
Verify comments exist
Sanity checks

Figure 3.5: Synthetic data generation.

Microsoft's methods2test dataset contains 780,944 pairs of focal methods and their associated JUnit test methods from 91,385 open-source repositories [44]. However, this data does not follow the strict coding guidelines which "@Test" snippets should ideally have. Also, LLMs face hallucinations which causes incorrect code. A possible solution for this is to convert a subsection of this dataset into simplified versions by using examples from the dataset as a seed with zero-shot prompting and performing sanity checks to ensure they are correct. Sanity checks can include compiling them and verifying they have a code coverage over 70% with comments. These simplified pairs of focal methods and test cases can be gathered to make a small but concise dataset for fine-tuning GPT-3.5 Turbo that is both unique (since it is inspired from open source repositories) but accurate (since sanity checks are performed).

One of the cheapest models to use in the current market is Mistral 7B [45] which is already fine-tuned to follow instructions with high accuracy. Therefore, it can be used to generate simplified versions of focal methods and test methods from open source repositories. However, it is computationally expensive to iterate through all entries in the methods2test dataset. Research suggests that GPT-3.5 should be fine-tuned by using a small training dataset, but this varies on the use case [46]. Datasets that are too big might catastrophic forgetting where a model struggles to generalize outside the fine-tuning data. OpenAI suggests that around 50 to 100 examples are sufficient [47]. To ensure that the model learns from a consistent testing structure, TestPilot uses 100 examples because too few examples have minimal effect and too many examples cause catastrophic forgetting.

Using the test set in the methods2test dataset (which is 10% of the entire dataset), the first tests in the first 100 repositories (when sorting them numerically by repository ID) are used as 100 pairs of focal methods and test functions. Each pair requires two inferences to generate a simplified focal method (based on the original focal method) and a simplified test method (based on the simplified focal method but with a similar goal as the original test method). The generated pair goes through sanity checks to ensure it compiles, has code coverage greater than 70%, and uses a comment. Comments are verified through regular expression checks in Python. If it fails the sanity check, the next entry is used until 100 examples are collected in this way.

The temperature of the model is set to 0.7 to balance creative and accurate responses, since research suggests that there is not a significant impact on problem-solving ability when sampling temperatures from 0 to 1 for LLMs [48]. For this system, temperature also does not have a massive impact, since all pairs go through sanity checks to ensure they are correct.

3.3 Fine-tuning

GPT-3.5 Turbo (`gpt-3.5-turbo-0125`) and GPT-4 (`gpt-4-0613`) are accessed through the OpenAI API [49]. API keys for OpenAI are stored locally in an `.env.server` file. Mistral-7B (`mistral-7B-instruct-v0.1`) is accessed for synthetic data generation through the Anyscale API [50]. The API key for Anyscale is stored locally in an `.env.processing` file. GPT-3.5 Turbo is fine-tuned on OpenAI's platform [51]. The synthetic data containing 100 examples is split into 80% for training (in `fine-tuning.jsonl`) and 20% for validation (in `validation.jsonl`) for 3 epochs. This resulting fine-tuned model is trained with 30,501 tokens and is able to create "@Test" snippets for test cases using the strict coding guidelines that the synthetic data follows.

By fine-tuning GPT-3.5 Turbo with 100 novice-level "@Test" snippets which follow strict coding guidelines, TestPilot is able to generate test cases that are consistent, suitably documented and at a level that a novice programmer can understand. The aim of fine-tuning is not to increase the knowledge base of GPT-3.5 Turbo, since that responsibility is given to a vector database. The resulting "@Test" snippets will follow the behaviours of the fine-tuning data, which in this case follow strict coding guidelines with an "Arrange, Act, Assert" testing strategy, including comments in test functions, and using simple code that does not have any unnecessary overhead such as external libraries that are not from `java.util`, `java.io` or `org.junit`.

The fine-tuning procedure was completed with a training loss of 0.0009 which indicates that the model fits the data very closely which is required for this use case. Novice programmers need a consistent structure in their tests to learn from. It also had a validation loss of 0.1735 and full validation loss of 0.1837 which suggests that the model is performing moderately well at generalizing with unseen data. In the graph shown below, it is evident that the training loss decreases over time which means the model is fitting the data better over time.

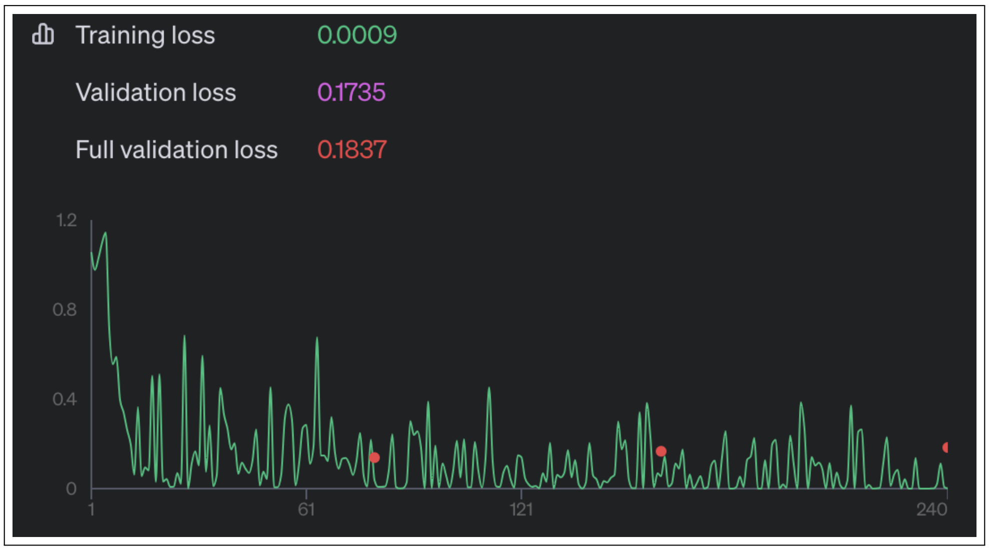


Figure 3.6: The fine-tuning procedure of TestPilot's GPT-3.5 Turbo model with steps on the x-axis versus training loss, validation loss and full validation loss on the y-axis.

3.4 Prompt engineering

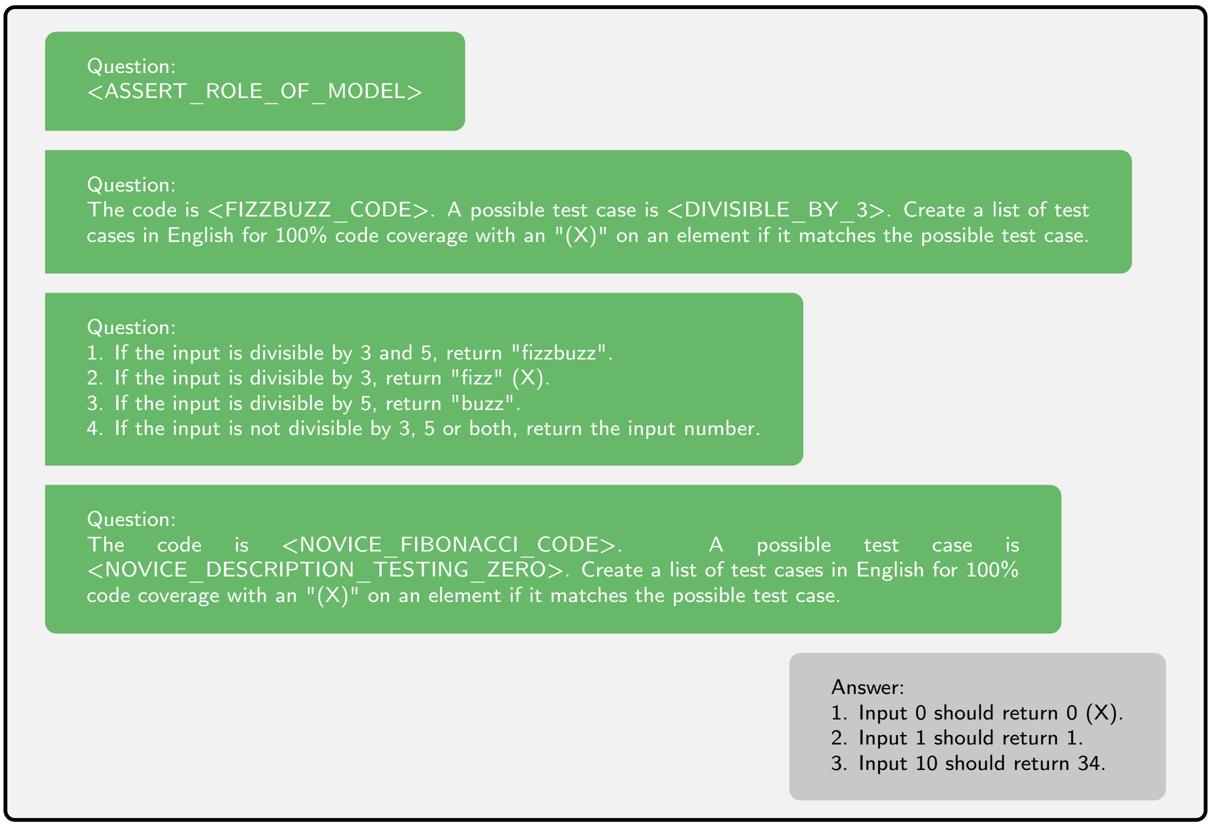
When prompting LLMs, few-shot prompting is used by TestPilot to increase the accuracy of the model. This includes adding an example into the prompt to help the model understand the context behind a question. TestPilot requires responses to follow a strict format such as a list of test cases being numbered in a list or a "@Test" snippet including no import statements or unnecessary text above or below the snippet. Zero-shot prompting is effective, but it can lack accuracy without sanity checks. Alternatively, few-shot prompting is 1 of the 26 prompting techniques which can improve the accuracy of responses [11]. Other prompting techniques which TestPilot uses are mentioned below.

- Assigning a role to the LLM as an "expert software tester" when generating a list of test cases and as a "unit test assistant" when generating "@Test" snippets.
- Adding the intended audience by mentioning this is "for novice programmers".
- Adding "“I'm going to tip $100 for a better solution!" when asking an LLM to generate a list of test cases to make them more accurate.
- Adding “You MUST” and “Your task is” when asking an LLM to generate a list of test cases or a "@Test" snippet.
- Adding "Ensure that your answer is unbiased and avoids relying on stereotypes" when sanitizing the novice programmer's vague test case description to avoid an LLM changing it to something else.
- Breaking down "complex tasks into a sequence of simpler prompts in an interactive conversation" by asking GPT-4 to generate a list of test cases and GPT-3.5 Turbo to generate a sequence of "@Test" snippets for example.

Apart from the techniques that were implemented, certain aspects of natural language were avoided to increase the accuracy of responses which are mentioned below.

- Avoiding negative language such as "do not" and replacing it with "do" instead.
- Avoiding unnecessary language to an LLM such as politeness with "I would like to" or "thank you" and using the direct task instead.

The commonly known beginner fizzbuzz problem [52] is used in TestPilot when prompting an LLM with few-shot prompting. For example, few-shot prompting is used when prompting GPT-4 to create a list of test cases. The code for fizzbuzz, a test case for fizzbuzz in natural language, and a list of test cases for the entire problem are supplied in the prompt. An example is shown below for a novice programmer using TestPilot when testing a Fibonacci sequence program.



Question:
<ASSERT_ROLE_OF_MODEL>

Question:
The code is <FIZZBUZZ_CODE>. A possible test case is <DIVISIBLE_BY_3>. Create a list of test cases in English for 100% code coverage with an "(X)" on an element if it matches the possible test case.

Question:
1. If the input is divisible by 3 and 5, return "fizzbuzz".
2. If the input is divisible by 3, return "fizz" (X).
3. If the input is divisible by 5, return "buzz".
4. If the input is not divisible by 3, 5 or both, return the input number.

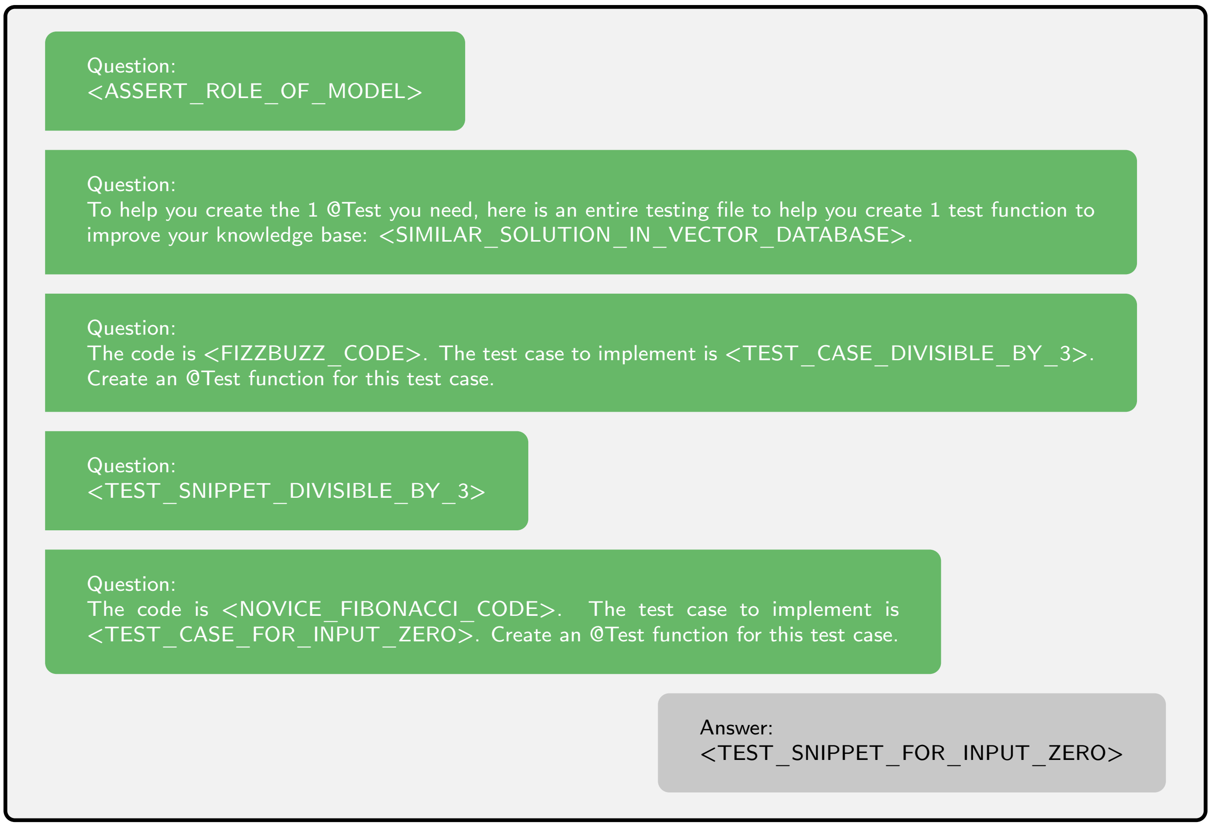
Question:
The code is <NOVICE_FIBONACCI_CODE>. A possible test case is <NOVICE_DESCRIPTION_TESTING_ZERO>. Create a list of test cases in English for 100% code coverage with an "(X)" on an element if it matches the possible test case.

Answer:
1. Input 0 should return 0 (X).
2. Input 1 should return 1.
3. Input 10 should return 34.

Figure 3.7: Prompting GPT-4 with few-shot prompting for a list of test cases.

A similar order of prompts are used for prompting GPT-3.5 Turbo to generate "@Test" snippets, except that they also include a known good working solution from a vector database. Through this approach, GPT-3.5 Turbo benefits by using a known good working solution, few-shot prompting to understand the ideal format of a response, and prompt engineering techniques which were mentioned previously.

Examples from fizzbuzz are included in the prompt to generate a "@Test" snippet since it serves as a solid foundation for the LLM to use as an example. It contains relatively few test cases while also containing multiple implementations (such as iterating from 1 to N or simply using one integer). Few-shot prompting with fizzbuzz aligns the model to generate a test case for a specific version of fizzbuzz and the model follows with this format when prompted with the code from a novice programmer as shown below.



Question:
<ASSERT_ROLE_OF_MODEL>

Question:
To help you create the 1 @Test you need, here is an entire testing file to help you create 1 test function to improve your knowledge base: <SIMILAR_SOLUTION_IN_VECTOR_DATABASE>.

Question:
The code is <FIZZBUZZ_CODE>. The test case to implement is <TEST_CASE_DIVISIBLE_BY_3>. Create an @Test function for this test case.

Question:
<TEST_SNIPPET_DIVISIBLE_BY_3>

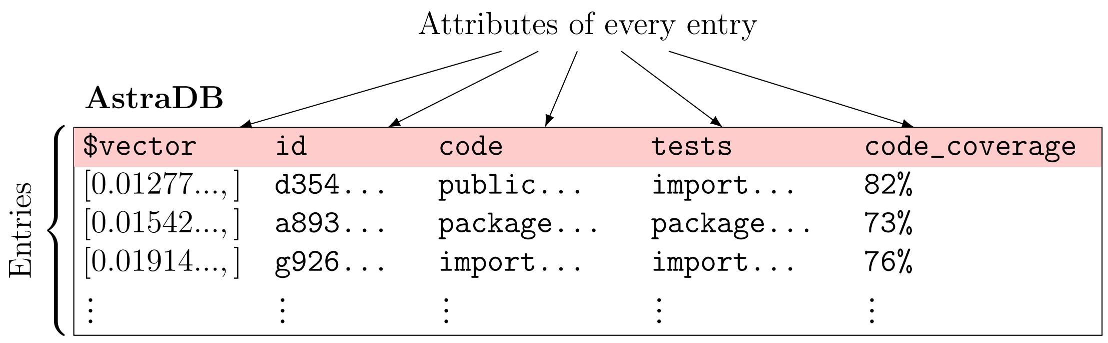
Question:
The code is <NOVICE_FIBONACCI_CODE>. The test case to implement is <TEST_CASE_FOR_INPUT_ZERO>. Create an @Test function for this test case.

Answer:
<TEST_SNIPPET_FOR_INPUT_ZERO>

Figure 3.8: Prompting GPT-3.5 Turbo with few-shot prompting for a "@Test" snippet.

3.5 Vector database

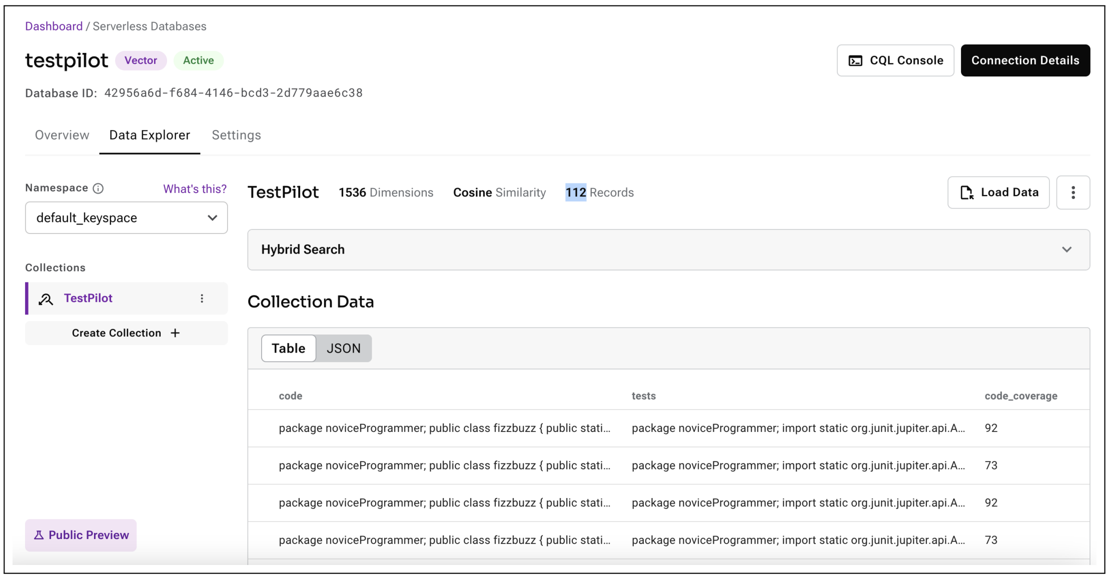
A vector database can contain known good working solutions to help improve the knowledge base of an LLM. Successful test files are appended to the database for improving the knowledge of future LLM inferences. Data is stored as embeddings to capture the semantic meaning behind the data instead of comparing exact text. AstraDB from DataStax [53] is used as a vector database, which gives each entry an ID. The code file, test file and code coverage from every successful test file is added. The code file is converted to an embedding vector using OpenAI's `text-embedding-3-small` model [54]. When generating a "@Test" snippet, the cosine similarity is used to find how similar the embedding of the novice programmer's code is to an embedding of an existing code file in the vector database which has an associated known good working testing file. The required keys for accessing the vector database are stored in an `.env.server` file.



$vector	id	code	tests	code_coverage
[0.01277...,]	d354...	public...	import...	82%
[0.01542...,]	a893...	package...	package...	73%
[0.01914...,]	g926...	import...	import...	76%
⋮	⋮	⋮	⋮	⋮

Figure 3.9: Attributes in the vector database for known good working solutions.

This known good working testing file is added into the prompt when generating a "@Test" snippet to increase the chance of a successful test case, since the LLM has an example when generating a new solution. TestPilot uses a cosine similarity of 0.8 as the threshold to acknowledge if an entry in the database is relatively similar to the code from a novice programmer. This value was used as it balances the success from solutions being similar enough and the failure from solutions not being similar enough. From all the entries that have a cosine similarity of 0.8 or higher, the entry with the highest is used. If no entry has a cosine similarity greater than 0.8, no known good working solution is added to the prompt, and only few-shot prompting with fizzbuzz as an example is used instead.



Database ID: 42956a6d-f684-4146-bcd3-2d779aae6c38

TestPilot 1536 Dimensions Cosine Similarity 112 Records

Collection Data

code	tests	code_coverage
package noviceProgrammer; public class fizzbuzz { public stati...	package noviceProgrammer; import static org.junit.jupiter.api.A...	92
package noviceProgrammer; public class fizzbuzz { public stati...	package noviceProgrammer; import static org.junit.jupiter.api.A...	73
package noviceProgrammer; public class fizzbuzz { public stati...	package noviceProgrammer; import static org.junit.jupiter.api.A...	92
package noviceProgrammer; public class fizzbuzz { public stati...	package noviceProgrammer; import static org.junit.jupiter.api.A...	73

Figure 3.10: TestPilot's AstraDB vector database from DataStax.

An interesting point to note is that, as this vector database is populated, the responses from TestPilot will become more accurate. A wide range of novice code and correct test code can be filled in the vector database. Therefore, novice programmers will benefit more as the database becomes more rich with examples. This is a unique feature since it has the potential to get better over time, since previously successful tests are appended to the vector database which is continuously populated by many novice programmers.

3.6 Backend Python Flask server

The Python Flask server performs the necessary logic to turn the inputs of TestPilot into outputs that are sent back to the Visual Studio Code extension. The server uses the OpenAI client to make API calls to GPT-4 and GPT-3.5 Turbo. The OpenAI client is also used to make API calls to Mistral 7B since the Anyscale platform is compatible with the same client. The server contains a variety of JAR files for performing operations such as a JUnit JAR for running tests [55], a JaCoCo JAR for calculating code coverage [56], and a google-java-format JAR for formatting the resulting testing file [57].

The server contains 2 endpoints for the Visual Studio Code extension to utilize. These endpoints are called when a novice programmer presses the "Generate" button on the Visual Studio Code extension. These endpoints represent the 2 modes of operation that TestPilot offers which are shown below.

- `/testpilot/testing` → "Testing" mode of operation.
    - Parameter: `code` → Code from an opened file on Visual Studio Code.
    - Parameter: `description` → Natural language description of a test case.
- `/testpilot/discovery` → "Discovery" mode of operation.
    - Parameter: `code` → Code from an opened file on Visual Studio Code.
    - Parameter: `description` → Natural language description of a test case.
    - Parameter: `testCode` → Test file contents on Visual Studio Code.
    - Parameter: `testCases` → Test cases in cards on Visual Studio Code.
    - Parameter: `knownTestCaseIndices` → Indices of discovered test cases.

When these endpoints are called, the server completes API calls to OpenAI's endpoints using research-driven prompt engineering. GPT-4 is called first to either generate a concise test case in "testing" mode or to generate a list of test cases in "discovery" mode. A fine-tuned GPT-3.5 Turbo model is subsequently called to generate a "@Test" snippet of the test case. When prompting GPT-3.5 Turbo, a known good working solution from the vector database is added if it exists. The server finds a potential known good working solution by performing a vector search on the vector database to find an entry with a cosine similarity greater than 0.8 compared to the novice programmer's code.

The generated "@Test" snippets from the LLM inferences are combined into one file with correct class encapsulation using string manipulation. The same package name from the original code is added with the necessary imports so that the new test file compiles. To cover all novice cases, the following imports are added using wildcards:

- `import java.util.*;` → Covering utility classes and interfaces.
- `import java.io.*;` → Covering input and output operations.
- `import org.junit.*` → Covering JUnit's annotations and assertion methods.
- `import static org.junit.jupiter.api.Assertions.*;` → Covering JUnit 5's assertion methods without the need to prefix them.

Subsequently, the class of the new test file uses the name of the original class from the novice programmer's code and appends "Test" to it. The compilation of Java files on the server is completed by creating temporary Java files and writing the generated tests into the files. After compiling and running the tests, the generated `.java` and `.class` files are deleted. After running JaCoCo, generated code reports are also deleted including `.exec` and `.csv` files.

These files are also deleted when performing sanity checks on synthetic data for fine-tuning. When generating synthetic data, the server iterates through the test set in the methods2test dataset and creates temporary files for the "@Test" snippets from Mistral 7B which are added to a class with string manipulation and deleted after the files are compiled and run.

The endpoint for the "testing" mode in TestPilot at `/testpilot/testing` returns a JSON response with the following attributes:

- `descriptions` → Test case in natural language.
- `tests` → Test code that was generated.
- `codeCoverage` → The code coverage of the test code.
- `vectorDatabaseCoverage` → The code coverage of a similar solution in a vector database if it exists.
- `testStatus` → A boolean representing whether the test case passed or failed.

The endpoint for the "discovery" mode in TestPilot at `/testpilot/discovery` returns a JSON response with the following attributes:

- `descriptions` → Test cases in natural language.
- `tests` → Test code that was generated.
- `codeCoverage` → The code coverage of the test code.
- `vectorDatabaseCoverage` → The code coverage of a similar solution in a vector database if it exists.
- `caseCoverage` → The case coverage of the discovered tests.
- `testFunctionNames` → Names of all functions in the generated test file.
- `indexOfDescriptionFromNovice` → The index of the test case from the novice.

These JSON responses are subsequently used by the Visual Studio Code extension to visualize the response from the LLM in an educational user interface using cards, buttons, and code implementations in the background.

3.7 Frontend Typescript Visual Studio Code extension

The Visual Studio Code extension provides an interface for a novice programmer to view and interact with the output of an LLM. On startup, the novice programmer chooses a mode of operation from the switch in the sidebar.

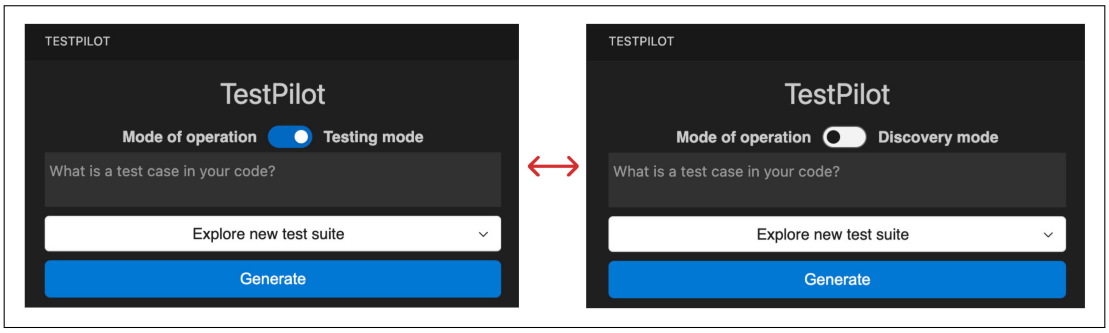
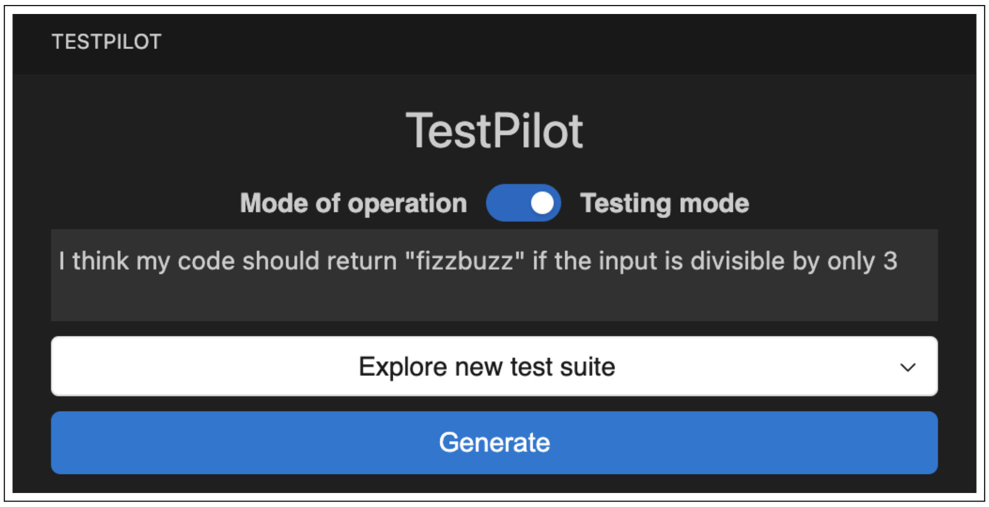


Figure 3.11: Switch in TestPilot to choose the mode of operation.

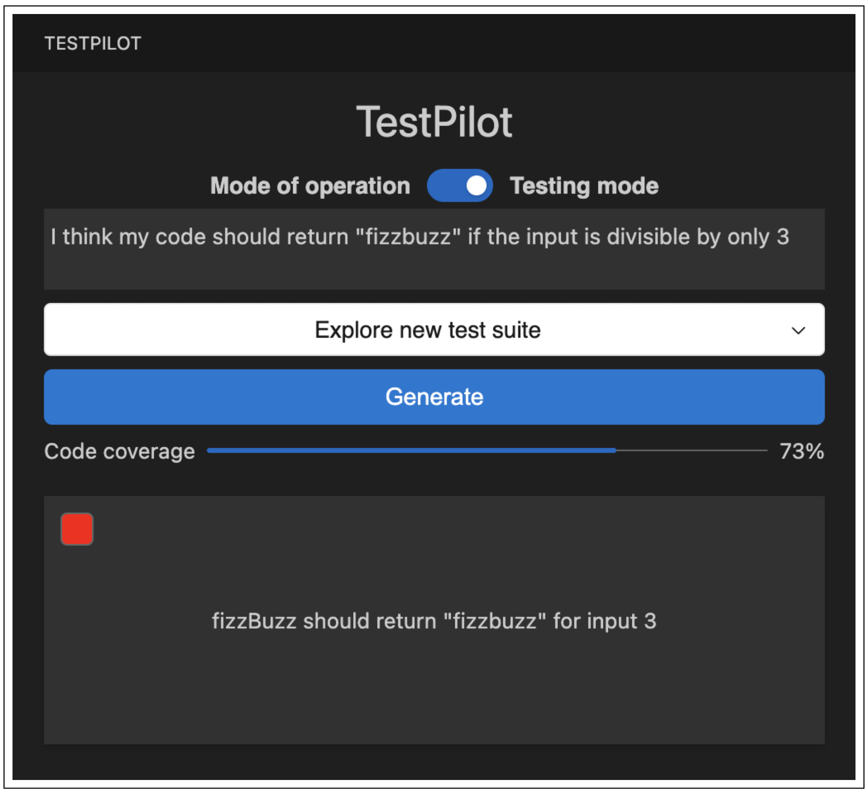
Subsequently, a novice programmer inputs a vague natural language description into the input text box and presses "Generate" to explore a new test suite. The `Axios` library [58] calls the correct endpoint on the server depending on the chosen mode of operation.



I think my code should return "fizzbuzz" if the input is divisible by only 3

Figure 3.12: Text input box in TestPilot to input a natural language description.

The natural language description of a test case and a snapshot of the file currently open on Visual Studio Code is sent to the server. When a response is received from the server, the Visual Studio Code extension visualizes the data in the JSON response using Microsoft's `Fluent UI` library [59] to render cards and buttons. In "testing" mode, a singular card is added to the sidebar representing a test case from the novice programmer's natural language description which may pass or fail.

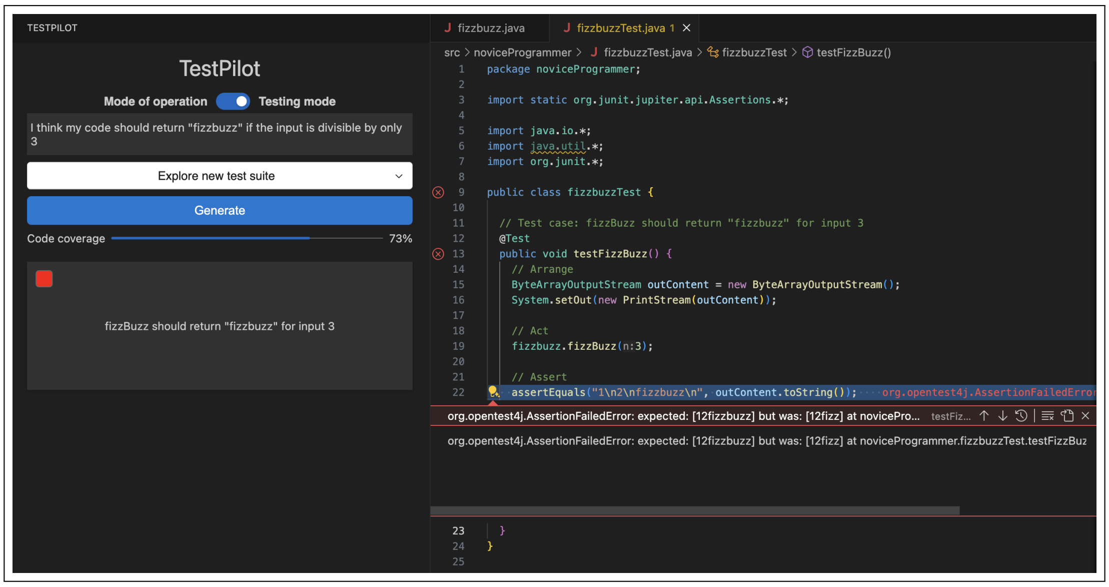


I think my code should return "fizzbuzz" if the input is divisible by only 3

🟥 fizzBuzz should return "fizzbuzz" for input 3

Figure 3.13: Failed test in "testing" mode.

Subsequently, a novice programmer can change the code implementation to make this test pass. Alternatively, a novice can restart by selecting the "explore new test suite" dropdown option to generate a test case for a new vague description. Either way, the generated tests are automatically placed in a testing file located in a "test" folder.



Mode of operation: Testing mode

I think my code should return "fizzbuzz" if the input is divisible by only 3

Code coverage 73%

🟥 fizzBuzz should return "fizzbuzz" for input 3

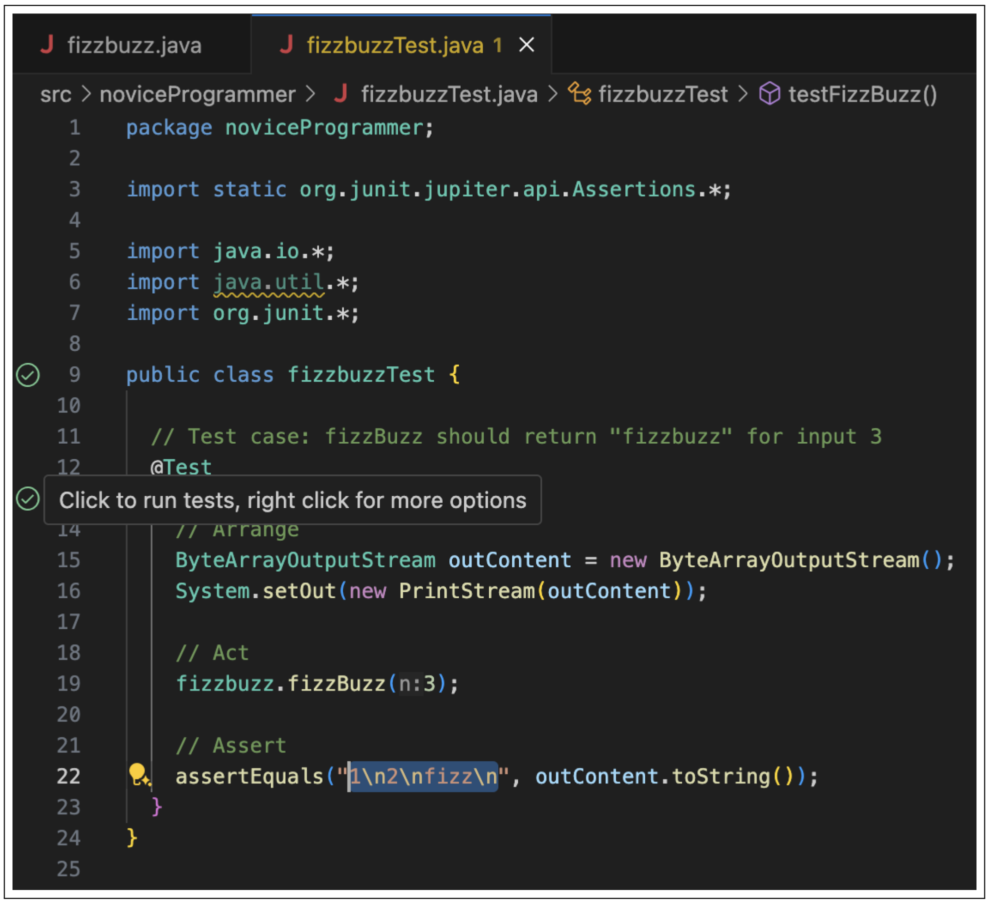
```java
package noviceProgrammer;

import static org.junit.jupiter.api.Assertions.*;

import java.io.*;
import java.util.*;
import org.junit.*;

public class fizzbuzzTest {

  // Test case: fizzBuzz should return "fizzbuzz" for input 3
  @Test
  public void testFizzBuzz() {
    // Arrange
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));

    // Act
    fizzbuzz.fizzBuzz(n:3);

    // Assert
    assertEquals("1\n2\nfizzbuzz\n", outContent.toString());   org.opentest4j.AssertionFailedError
```

```
org.opentest4j.AssertionFailedError: expected: [12fizzbuzz] but was: [12fizz] at novicePro...

org.opentest4j.AssertionFailedError: expected: [12fizzbuzz] but was: [12fizz] at noviceProgrammer.fizzbuzzTest.testFizzBuz
```

```java
  }
}
```

Figure 3.14: Code implementation of a generated test case.

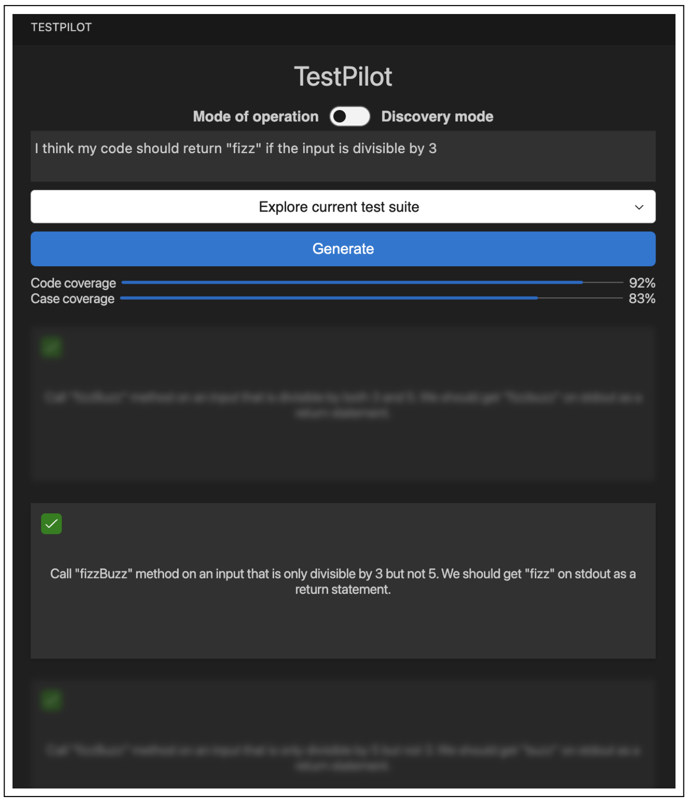
If a novice programmer sees a failing test case in "testing" mode, they can attempt to fix the test and rerun the code in Visual Studio Code to learn from their mistakes.


fizzbuzz.java
fizzbuzzTest.java 1
src > noviceProgrammer > fizzbuzzTest.java > fizzbuzzTest > testFizzBuzz()
Click to run tests, right click for more options

```java
package noviceProgrammer;

import static org.junit.jupiter.api.Assertions.*;

import java.io.*;
import java.util.*;
import org.junit.*;

public class fizzbuzzTest {

  // Test case: fizzBuzz should return "fizzbuzz" for input 3
  @Test

    // Arrange
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));

    // Act
    fizzbuzz.fizzBuzz(n:3);

    // Assert
    assertEquals("1\n2\nfizz\n", outContent.toString());
  }
}
```

Figure 3.15: Changing a failing test to make it pass.

In "discovery" mode, all test cases for the entire program space are shown in cards which are "blurred", with the test case that matches closest to the novice programmer's input being "unblurred" to give the novice programmer an initial start for exploration.


TESTPILOT

TestPilot

Mode of operation Discovery mode

I think my code should return "fizz" if the input is divisible by 3

Code coverage 92%
Case coverage 83%

[illegible]

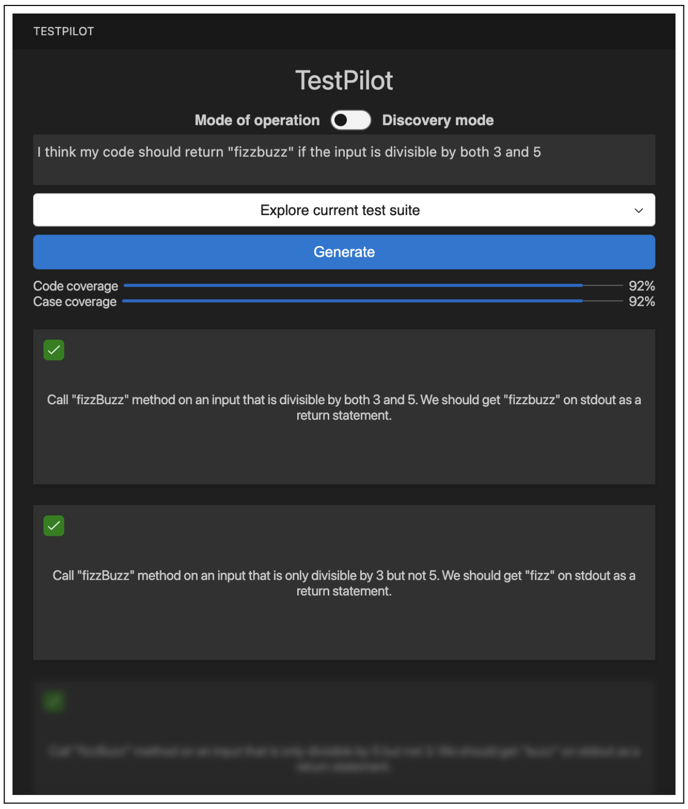
✓

Call "fizzBuzz" method on an input that is only divisible by 3 but not 5. We should get "fizz" on stdout as a return statement.

[illegible]

Figure 3.16: "Blurred" and "unblurred" test cases in "discovery" mode.

More test cases can be inputted to "unblur" more cards if the natural language description matches the test case. If the description does not match, nothing will be "unblurred".


TESTPILOT

TestPilot

Mode of operation Discovery mode

I think my code should return "fizzbuzz" if the input is divisible by both 3 and 5

Code coverage 92%
Case coverage 92%

✓

Call "fizzBuzz" method on an input that is divisible by both 3 and 5. We should get "fizzbuzz" on stdout as a return statement.

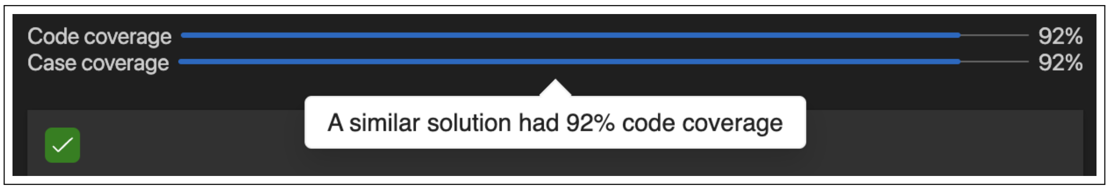
✓

Call "fizzBuzz" method on an input that is only divisible by 3 but not 5. We should get "fizz" on stdout as a return statement.

[illegible]

Figure 3.17: "Unblurring" a test case in "discovery" mode.

The Visual Studio Code extension shows progress bars for code coverage in "testing" mode, but also shows the case coverage of the "unblurred" cards in "discovery" mode. This helps a novice programmer understand how much of the program has been explored. A tooltip shows the code coverage of a similar solution from a vector database if it exists.



A similar solution had 92% code coverage

Figure 3.18: Code coverage and case coverage in progress bars.

These two modes of operation with the use of cards, progress bars and code implementations makes an effective Visual Studio Code extension that helps novice programmers learn how to test using helpful test cases that are consistent, correct, and educational.

4 Evaluation

It has been established that TestPilot aims to improve the cost, correctness, and educational benefit of generated tests for novice programmers because there is a need to do so. This need is satisfied through the intricate design of TestPilot. To test this hypothesis, internal and external evaluations can be completed. Internal evaluations include verifying if prompt engineering, fine-tuning and embeddings improve the generated tests and if so, examining how much they improve the generated tests by.

External evaluations include verifying whether the generated tests from TestPilot are better than a single prompt to ChatGPT. The improvement of tests can be evaluated in the context of cost (from the number of tokens used), correctness (semantically and syntactically) and educational benefit (from adequate documentation). This chapter will perform these internal and external evaluations through the use of metrics, a performance analysis and a cost analysis to make an overall judgement on the efficacy of TestPilot.

4.1 Evaluation dataset

A sequence of 50 source code files were allocated to prompt TestPilot and ChatGPT with. This evaluation dataset consists of 20 Leetcode "easy" problems [60], 17 questions from Exercism.org [61], and 13 questions from the module Introduction to Programming I (CSU11011) at Trinity College Dublin [62]. These code files were chosen since they represent the type of problems that a novice programmer would typically solve. For each of the 50 code files, a beginner natural language description was mapped to it to describe a test case which could be right or wrong. The first 25 code files were mapped to a description which was correct and subsequently should create a passing test. The remaining 25 code files were mapped to a description which was wrong and subsequently should create a failing test.

These 50 code files and 50 natural language descriptions were placed in an `evaluation.json` file to iterate through when prompting. When running the evaluation dataset against TestPilot, the backend server internally calls the `testpilot/testing` endpoint for the "testing" mode of operation to prompt with the evaluation dataset. The "discovery" mode of operation is not called, since the "testing" and "discovery" modes of operation both use the same underlying mechanism to generate a "@Test" snippet from a fine-tuned GPT-3.5 model that uses prompt engineering and embeddings; this mechanism is the focus of the evaluation dataset. Furthermore, 25 source code files have descriptions which are correct and should result in correct tests, therefore the underlying mechanism behind the "discovery" mode of operation is already tested in this way.

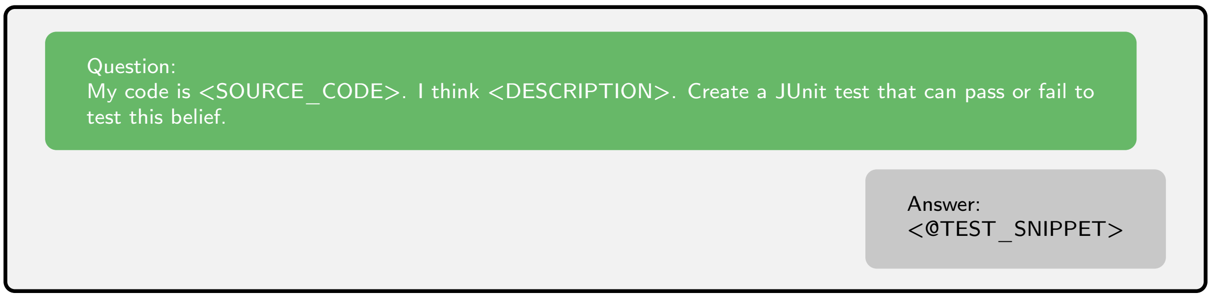
The difference between the modes of operation is that "testing" mode generates a single test case based on a natural language description which may pass or fail, and "discovery" mode generates test cases for the entire program space which aim to always pass. By only calling the "testing" endpoint, the evaluation dataset is effectively testing how well the system can convert a vague natural language description into an accurate test case in code. When running the evaluation dataset against ChatGPT, all 50 source files and descriptions were manually passed into the online ChatGPT interface [63]. Regardless of the description or mode of operation, the resulting tests should compile.

The goal of the evaluation dataset is to evaluate the overall operation of TestPilot by testing how well a natural language description can be converted to a test case which compiles and has adequate documentation. It is important to note that an evaluation dataset for evaluating any LLM will have bias, since different questions which are not in the evaluation dataset might lead to different results. The current evaluation dataset tries to minimize this bias by gathering source code files from various sources. Furthermore, different inferences with the same evaluation dataset will have slightly different results, since LLMs respond differently across different inferences with the same prompt. Therefore, the evaluation dataset must be used to discuss the general trends in the results rather than focusing on exact figures which may change across inferences.

4.2 Metrics

During internal evaluations, the generated tests will be tested with the default TestPilot system including prompt engineering, fine-tuning and embeddings, and subsequently tested after removing each technique from the system to examine the impact it had. During external evaluations, TestPilot and ChatGPT will be examined with the evaluation dataset where an additional cost analysis will be performed. The improvement of tests is evaluated in the context of *cost, correctness* and *educational benefit.*

For examining *correctness*, the internal and external evaluation will record the **compilation rate** (how many of the generated tests compile) and **test accuracy rate** (how many of the generated tests accurately passed or failed depending on the natural language description). For examining the *educational benefit*, the internal and external evaluation will record the **documentation rate** (how many of the generated tests have comments). For examining *cost*, the external evaluation will include a **cost analysis** to examine the cost of TestPilot versus ChatGPT for the tokens used. ChatGPT is currently free for users, so the cost of the underlying GPT-3.5 model that powers ChatGPT will be used in place of this to represent the true cost. When prompting ChatGPT or using TestPilot without prompt engineering, the prompt shown below was used.


Question:
My code is <SOURCE_CODE>. I think <DESCRIPTION>. Create a JUnit test that can pass or fail to test this belief.

Answer:
<@TEST_SNIPPET>

Figure 4.1: Prompts for ChatGPT or TestPilot with no prompt engineering.

4.3 Interval evaluation

This section will perform an internal evaluation of TestPilot to examine how much each technique used by TestPilot improves the generated tests. The attributes of improved tests include the compilation rate, test accuracy rate, and the documentation rate. The sequence of 50 source code files and descriptions from the evaluation dataset was passed to the "testing" mode of operation in TestPilot with the default setup (using prompt engineering, fine-tuning and embeddings) and subsequently without each of prompt engineering, fine-tuning and embeddings. It is important to note that the compilation rate, test accuracy rate, and documentation rate varied across multiple runs with the evaluation dataset since LLMs return different results across different inferences. However, the general trends from multiple runs are recorded and discussed below.

4.3.1 Prompt engineering

The evaluation dataset was used on the default TestPilot system and also after removing prompt engineering techniques to show the effects on the responses. The results are recorded below in terms of the compilation rate, test accuracy rate and documentation rate.

	Default TestPilot	**Without prompt engineering**
Compilation rate	94% ($\frac{47}{50}$)	62% ($\frac{31}{50}$)
Test accuracy rate	78% ($\frac{39}{50}$)	58% ($\frac{29}{50}$)
Documentation rate	96% ($\frac{48}{50}$)	90% ($\frac{45}{50}$)

Table 4.1: Metrics of tests using TestPilot with and without prompt engineering.

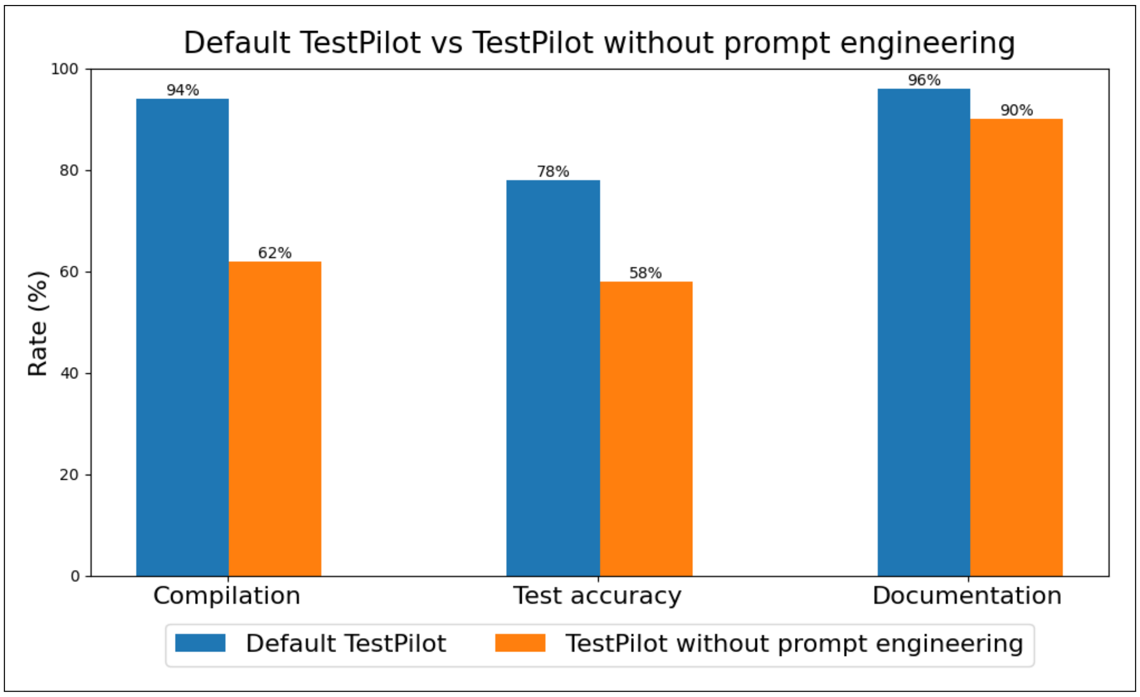


Figure 4.2: Grouped bar chart visualizing the metrics of tests using TestPilot with and without prompt engineering.

From these results, it is clear that the compilation rate and test accuracy rate decrease significantly without prompt engineering. Prompt engineering improved the chance of tests compiling by 32% which can be explained by the use of few-shot prompting with the fizzbuzz example using the correct identifier when calling the function in the format `class_name.function()`. Without this, the system often created function calls with no references to where it was coming from or making up a random function, causing compiler errors from `random_class.function()` or simply `incorrect_function_name()`.

Prompt engineering increased the test accuracy rate by 20% which can be explained by prompting techniques such as "Ensure that your answer is unbiased and avoids relying on stereotypes". The lack of these prompts caused the system to create incorrect tests based on the natural language description. Prompt engineering increased the documentation rate by only 6%. This marginal increase can be accounted for by the few-shot prompting example with fizzbuzz which includes useful comments above and inside functions.

4.3.2 Fine-tuning

The evaluation dataset was used on the default TestPilot system and also after removing fine-tuning to show the effects on the responses. The results are recorded below in terms of the compilation rate, test accuracy rate and documentation rate.

	Default TestPilot	**Without fine-tuning**
Compilation rate	90% ($\frac{45}{50}$)	86% ($\frac{43}{50}$)
Test accuracy rate	82% ($\frac{41}{50}$)	74% ($\frac{37}{50}$)
Documentation rate	92% ($\frac{46}{50}$)	56% ($\frac{28}{50}$)

Table 4.2: Metrics of tests using TestPilot with and without fine-tuning.

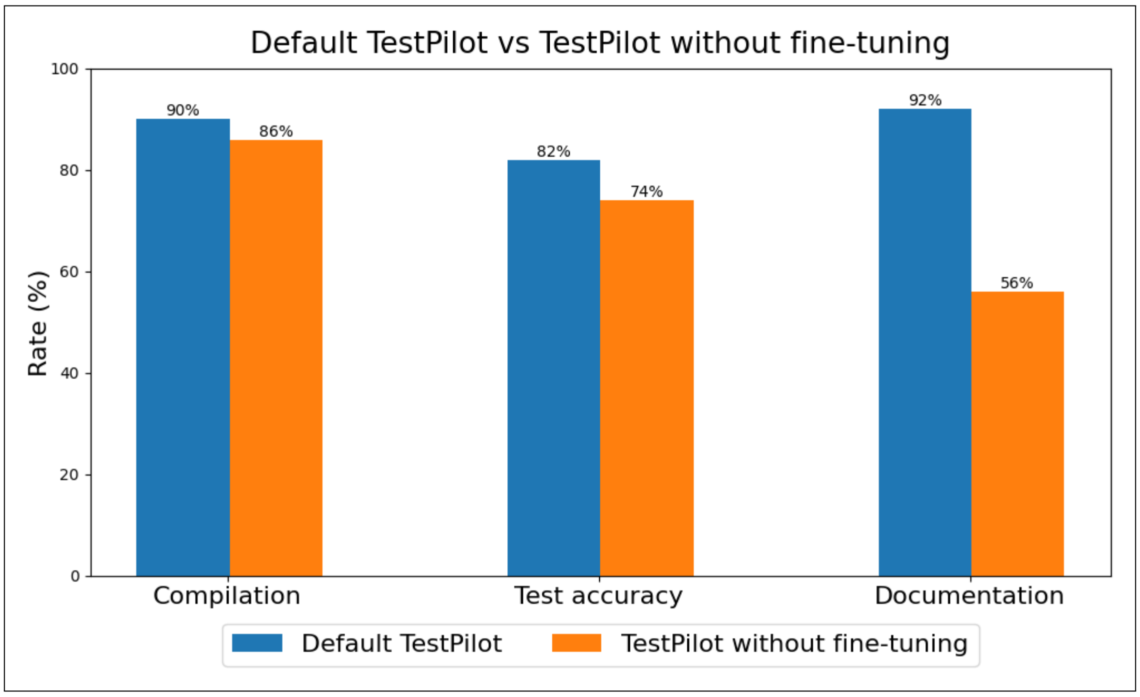


Figure 4.3: Grouped bar chart visualizing the metrics of tests using TestPilot with and without fine-tuning.

From these results, it is evident that the compilation rate and test accuracy rate only decreased slightly without fine-tuning. Fine-tuning improved the chance of tests compiling by 4% and the chance of a natural language description becoming an accurate test by 8%. These increases are marginal because fine-tuning does not aid the system in terms of its knowledge base. The main benefit from fine-tuning is that an LLM learns the ideal format of responses which includes stylistic behaviours. Therefore, these increases are insignificant and can occur because of LLMs responding differently to the same prompt.

However, the documentation rate increased by 36% with fine-tuning. This significant improvement is justified by the stylistic and behavioural improvements from fine-tuning. Fine-tuning was completed with synthetic data that contained comments above every function and inside each function, which followed an "Arrange, Act, Assert" testing strategy with comments for each step. This format in the fine-tuning data improved the rate of documentation significantly when using the fine-tuned GPT-3.5 Turbo model.

4.3.3 Embeddings

The evaluation dataset was used on the default TestPilot system and also after removing embeddings to show the effects on the responses. The results are recorded below in terms of the compilation rate, test accuracy rate and documentation rate.

	Default TestPilot	**Without embeddings**
Compilation rate	92% $(\frac{46}{50})$	88% $(\frac{44}{50})$
Test accuracy rate	74% $(\frac{37}{50})$	74% $(\frac{37}{50})$
Documentation rate	94% $(\frac{47}{50})$	92% $(\frac{46}{50})$

Table 4.3: Metrics of tests using TestPilot with and without embeddings.

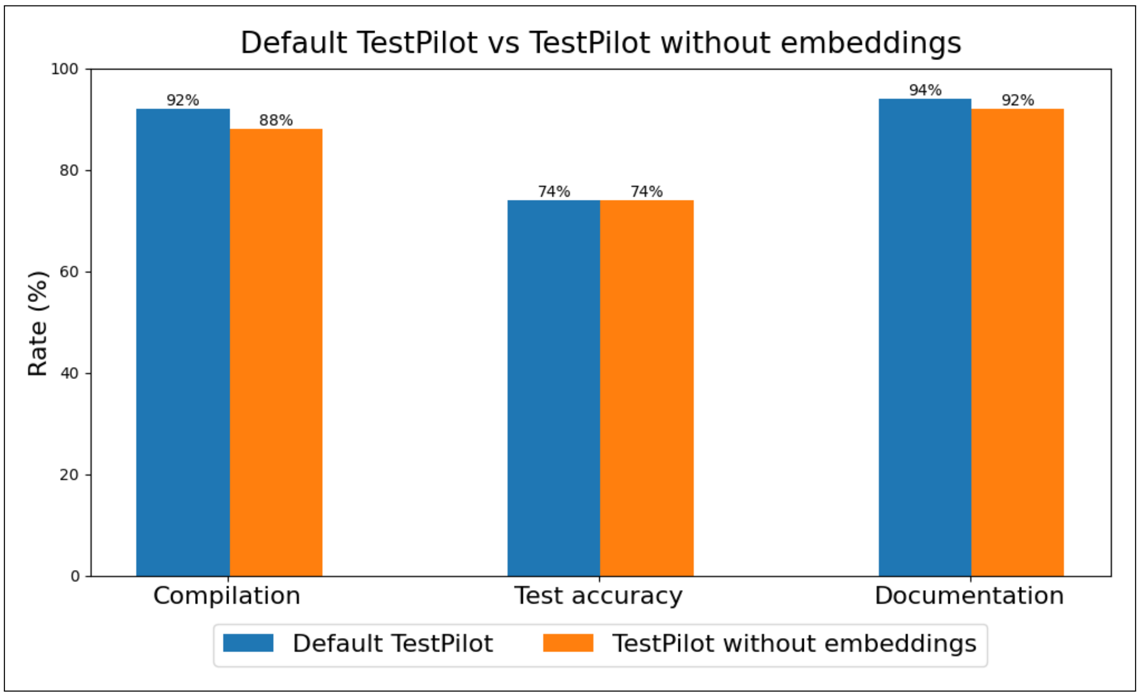


Figure 4.4: Grouped bar chart visualizing the metrics of tests using TestPilot with and without embeddings.

From these results, it is clear that the documentation rate and compilation rate increase marginally with embeddings. These small increases can be accounted for by the addition of code files and test files from the evaluation dataset into the vector database. Across 50 inferences using the evaluation dataset, the inferences from the end of the evaluation dataset benefited from known good working solutions from previous inferences, which improved the compilation rate by 4% and the documentation rate by 2%. However, it is important to note that these are small increases which can also be accounted for by the LLM giving different responses across different inferences, meaning they are not majorly significant.

The test accuracy rate remained the same with and without embeddings, which can be explained by the fact that embeddings do not improve the system's conceptual reasoning to create a correct test for the novice. The test file from the most similar source code in the vector database is added into the prompt, which does not have an impact on test accuracy since the source code files in the evaluation dataset deal with varying concepts that are not shared (unlike the shared syntax and comments which improved the compilation rate and documentation rate respectively).

4.4 External evaluation

This section will perform an external evaluation of TestPilot to examine how well it performs to an alternative standard LLM. ChatGPT uses GPT-3.5 to answer questions, which is viable to use for comparing the improvement of generated tests. The attributes of improved tests include the compilation rate, test accuracy rate and the documentation rate. The sequence of 50 source code files and descriptions from the evaluation dataset was passed into the "testing" mode of operation in Testpilot and subsequently into the online interface for ChatGPT. Furthermore, the average tokens used for input was calculated to perform a cost analysis between TestPilot and ChatGPT, where the costs for GPT-3.5 Turbo used in place of ChatGPT to determine the true cost.

4.4.1 ChatGPT

The evaluation dataset was used on the default TestPilot system and also on ChatGPT to examine the improvement of tests. The results are recorded below in terms of the compilation rate, test accuracy rate and documentation rate.

	TestPilot	**ChatGPT (powered by GPT-3.5)**
Compilation rate	90% ($\frac{45}{50}$)	62% ($\frac{31}{50}$)
Test accuracy rate	80% ($\frac{40}{50}$)	54% ($\frac{27}{50}$)
Documentation rate	94% ($\frac{47}{50}$)	64% ($\frac{32}{50}$)

Table 4.4: Metrics of tests using TestPilot and ChatGPT.

From these results, it is clear that the compilation rate, test accuracy rate and documentation rate all increase with TestPilot. The compilation rate increased by 28% which can be explained by the addition of import statements and class encapsulation using string manipulation for the outputs of LLM inferences. Furthermore, known good working solutions from the vector database also increased the compilation rate, since the correct identifiers and function usage from previous solutions helped with syntax.

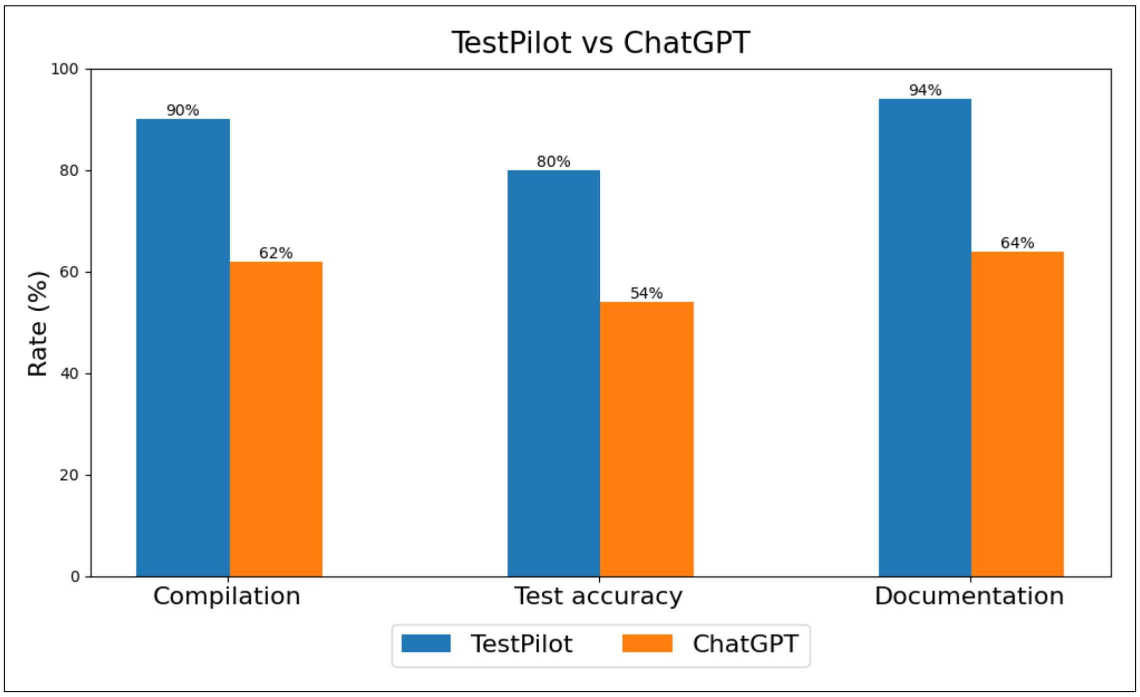


Figure 4.5: Grouped bar chart visualizing the metrics of tests using TestPilot and ChatGPT.

The use of prompt engineering also impacted the test accuracy rate which increased by 26%. Prompt engineering increased the test accuracy rate by specifying that the model should avoid bias, and also effectively convert natural language descriptions into code implementations through prompts such as "I'm going to tip $1000 for a better solution!". The documentation rate increased by 30% which can be explained by the fine-tuned GPT-3.5 Turbo model which was trained on synthetic data that followed strict coding guidelines. The synthetic data contained comments above each function and comments for the steps in the "Arrange, Act, Assert" testing strategy.

The accuracy recorded by ChatGPT in this evaluation aligns with the accuracy reported by current literature which reaffirms the validity of the evaluation dataset [34]. It is evident from this evaluation that ChatGPT often ignores the context of code snippets, leading to almost half of the resulting tests not compiling. Furthermore, conceptual reasoning is lacking in ChatGPT, resulting in only 54% of tests being what the novice programmer intended to test from their vague natural language description. From the improvements mentioned previously, it is clear that TestPilot improves compilation and documentation by a significant margin and the test accuracy by a moderate margin.

4.4.2 Cost analysis

The cost of tokens from LLM usage is an important metric since it shows if a system is financially viable. Since LLMs output different responses across different inferences, this cost analysis will avoid the discussion of output token lengths since this can change highly. Instead, this cost analysis will discuss the token usage from cost of input tokens since this is a stable metric which can be evaluated from the evaluation dataset. This stable metric will explain the extra costs that TestPilot requires. It is important to note that TestPilot uses multiple inferences across GPT-4 and GPT-3.5 Turbo to develop tests, whereas ChatGPT only uses GPT-3.5. Input tokens include a novice natural language description, novice code, and prompt engineering from techniques such as few-shot prompting. This analysis aims to decide if the extra costs from splitting the logic across multiple inferences is worth it for the improvements in tests mentioned previously. The current price for GPT-4 is \$0.03/1000 input tokens. The current price for GPT-3.5 Turbo is \$0.0005/1000 input tokens. ChatGPT uses GPT-3.5 but OpenAI currently does not offer this model directly from an API any more, therefore it is plausible to use the price of GPT-3.5 Turbo in place of this which is \$0.0005/1000 input tokens. The average tokens and average input cost is displayed below for the evaluation dataset.

	TestPilot	**ChatGPT**
Average GPT-4 input tokens	1402	0
Average GPT-3.5 Turbo input tokens	2227	854
Average input cost per test	\$0.0431	\$ 0.0004

Table 4.5: Evaluating the average cost of input with TestPilot and ChatGPT.

The average cost for GPT-4 input in TestPilot includes 63 tokens on average from the descriptions in the evaluation dataset and 1339 tokens from the prompt engineering required which totals to 1402. The average cost for GPT-3.5 Turbo in TestPilot includes 698 tokens on average from the code in the evaluation dataset and 1529 tokens from the prompt engineering required which totals to 2227. Therefore, the total cost of input tokens for TestPilot is $(\frac{1402}{1000} \times 0.03) + (\frac{2227}{1000} \times 0.0005) \approx 0.0431$. The average cost of GPT-3.5 Turbo in ChatGPT includes 63 tokens on average from the descriptions in the evaluation dataset, 698 tokens on average from the code in the evaluation dataset and around 93 tokens for a standard prompt which is mentioned in figure 4.1 which totals to 854. Therefore, the total cost of input tokens for ChatGPT is $\frac{854}{1000} \times 0.0005 \approx 0.0004$.

These results show that ChatGPT is extremely cheap for inputting the standard prompts and completing all test generation in one inference. However, it has been established that this leads to unsuitable tests that are inconsistent, incorrect, or non-educational. These results show that TestPilot is around 10 times more expensive for input tokens, but this price is also extremely cheap since the original cost was small to begin with. This highlights that the improvement in tests comes at a price, but this is worth it since TestPilot offers a 36% improvement in compilation, an 18% improvement in test accuracy and a 30% improvement in documentation as mentioned in the previous subsection. It is clear that the additional benefits from this small cost are justified from the educational benefit that follow.

This cost analysis discussed the input tokens required to effectively transfer information from a novice into large language models. TestPilot was around 10 times more expensive, but this was because the original cost of an inference was extremely cheap. TestPilot requires extra tokens because it splits the required logic into various inferences. It is valuable to note that future implementations of TestPilot may include further costs. For example, AstraDB is free for up to 80 GB of embeddings, and the generation of synthetic data was also free from the \$10 free credits on Anyscale for Mistral 7B which costs \$0.15/million tokens. Furthermore, the conversion of text to embeddings with text-embedding-3-small from OpenAI costs \$0.00002/1K tokens, however this is too insignificant to make a difference and is justified from the small increase it provides in the compilation rate by 4%.

4.5 Conclusion

TestPilot was evaluated internally to validate if parts of the system improve the effectiveness of the generated tests. It was found that prompt engineering increased the compilation rate of tests by 32%, the test accuracy rate by 20% and the documentation rate by 6%. This suggested that prompt engineering affects the compilation of tests and accuracy of tests the most because of the model being prompted using few-shot prompting which contains a structure of a test that will compile as well as requesting that the model is unbiased in its responses. However, prompt engineering did not significantly improve the documentation of tests, since this it does not impact the stylistic behaviour of responses as much as fine-tuning does.

Fine-tuning increased the compilation rate by 4%, the test accuracy rate by 8% and the documentation rate by 36%. This suggested that fine-tuning improves the ability of an LLM to respond with consistent comments that document the code. However, the compilation rate and test accuracy were not improved by a large factor, since fine-tuning does not improve the knowledge or conceptual reasoning of an LLM. Embeddings improved the compilation rate by 4%, the test accuracy rate by 0%, and the documentation rate by 2%. This suggested that embeddings do not result in massive improvements in tests, however they do result in small improvements which are justified from the minor costs that are associated with them.

TestPilot was evaluated externally to validate if the system was effective compared to a single prompt to ChatGPT. It was shown that TestPilot improves the compilation rate by 28%, the test accuracy rate by 26% and the documentation rate by 30%. This suggested that TestPilot provides tests that are more usable by novice programmers because they are more correct and documented. The costs of TestPilot were analysed, and it was 10 times more expensive on average for input tokens compared to ChatGPT. This suggested that the improvements in the generated tests comes at a price, relating heavily to the commonly known "no free lunch" theorem that was popularized for optimizations in machine learning [64]. This extra cost was justified because of the massive improvements in the generated tests which was mentioned previously. In combination with the educational layout of TestPilot, this evaluation suggests that TestPilot is an efficient system that is cost-effective in return for the improvements in tests that it offers.

5 Conclusion

This dissertation has established that novice programmers are unable to use tests found online because they are inconsistent, incorrect, or non-educational. It has also established that standard LLMs face the same issues from online searches because of their training data containing complex code, a lack of context from a novice programmer's messy code, and hallucinations causing incorrect tests. To solve this, this dissertation introduced TestPilot, a Visual Studio Code extension for teaching novice programmers how to write JUnit tests in Java. By inputting a vague natural language description of your code, it returns high-quality JUnit test cases which are semantically and syntactically correct, at an appropriate level for a novice programmer, and written in a consistent style that is suitably documented.

TestPilot introduced two modes of operation, a "testing" mode for testing the beliefs of a novice programmer by generating a single test which may pass or fail, and a "discovery" mode for generating a suite of tests for the entire program space. By using a strategic sequence of LLM inferences with prompt engineering, fine-tuning, and embeddings, TestPilot converts a vague natural language description into test cases in code that have a higher chance of compiling, testing the intended logic from a vague prompt, and including documentation alongside code through comments. Compared to a single prompt to ChatGPT, TestPilot increased the compilation rate by 28%, the test accuracy rate by 26%, and the documentation rate through comments by 30%.

5.1 Reflection

Upon reflection, the development of TestPilot was a difficult process that involved many challenges. The development of TestPilot started by gaining an understanding of how LLMs work and how novice programmers can utilize them from current literature. This initial period of understanding how LLMs can be adapted for novice programmers was a vital but demanding step, since artificial intelligence is a constantly changing area. For example, OpenAI posted 28 updates on their blog over the course of TestPilot's development [65]. Adapting to new updates and current research was a demanding but important part of developing TestPilot.

In retrospect, the process of constantly updating TestPilot was also a vital part of TestPilot's development. An initial version 1 was developed which made simple API calls to GPT-4 to generate tests for the entire program space with only prompt engineering. This was not cost-effective and did not provide novice-level code that was consistently documented. Subsequently, a version 2 was developed which outsourced lengthy code generation to a fine-tuned GPT-3.5 Turbo model that used embeddings which was cost-effective and generated code at a novice-level with consistent comments for documentation while improving over time with known good working solutions. Finally, a version 3 was implemented after the initial presentation for TestPilot which suggested that a "testing" mode of operation should exist to generate tests that may pass or fail to test the beliefs of a novice programmer. This iterative development process which involved producing new versions based on results and feedback was challenging but also rewarding. Ultimately, constantly adapting to new research and opinions was the biggest challenge, but this also made it an enjoyable experience.

5.2 Successes

The development of TestPilot resulted in a number of accomplishments which made it ultimately successful. The educational user interface in TestPilot was successful because it implemented advice from current literature which suggested that novice programmers do not like large chunks of text which simply hand over the entire solution. Subsequently, the use of cards, progress bars showing code coverage, and code implementations in a testing file in the background were successful aspects of TestPilot's design which resulted from using current literature.

TestPilot was successful at implementing useful features for novice programmers, such as a "testing" and "discovery" mode of operation. The use of feedback from the initial presentation resulted in developing various modes of operation. Generating tests that may pass or fail in "testing" mode helps novice programmers who are unsure if their code works as intended, and wish to test it using natural language as the source of truth. Generating a suite of tests for the entire program space in "discovery" mode helps novice programmers who wish to explore all possible test cases by using their code as the source of truth. These modes of operation made TestPilot suitable for novice programmers who have different learning goals, and ultimately made it more successful.

In terms of metrics, TestPilot improved compilation of tests by 28% and test accuracy by 26% which was done through research-driven prompt engineering to aid class encapsulation and correct function calls. TestPilot also improved the rate of documentation using comments by 30% which was done through fine-tuning to aid in the stylistic behaviour of responses which also made them a novice-level. The use of embeddings improved the overall rate of compilation and test accuracy, leaving a potential for responses to get better over time. Overall, TestPilot uses artificial intelligence techniques to make the process of learning software testing easier for novice programmers, which makes it successful.

5.3 Future work

This section will outline potential work which can be completed in the future as a result of reflecting on TestPilot and its successes. Prompt engineering and fine-tuning had significant improvements on the compilation rate, test accuracy rate, and documentation rate of generated tests. However, embeddings only improved these metrics by a small margin. The costs of embeddings were minimal for vector databases as mentioned in the cost analysis, therefore the use of embeddings was justified for TestPilot since any increase in the metrics mentioned previously was welcomed. However, there is a potential to expand on this work by populating the vector database with more known good working solutions, depending on the use case of a novice programmer. For example, lecturers or teaching assistants teaching a programming module may populate the vector database with known good working solutions from previous academic years, which may help students gain from bigger increases in the metrics mentioned previously. There is a potential for future work on the implementation and usage of vector databases being adapted to specific use cases in software testing education.

Furthermore, TestPilot focused on using relatively popular LLMs such as OpenAI's GPT-3.5 Turbo and GPT-4. However, lesser known LLMs were also used, such as Mistral 7B for the generation of synthetic data. There is a potential to use alternative LLMs in the future as more LLMs are released in the future with better conceptual reasoning abilities. For example, OpenAI's current CEO Sam Altman has confirmed that GPT-5 will be released in the near future. Future iterations of interfaces that teach software testing using LLMs may not require intensive prompt engineering, since LLMs in the future will get better at understanding the context behind a prompt without needing to prompt it in a particular format.

Fine-tuning was an essential part of making TestPilot generate tests at a novice-level that were suitably documented. Future work may involve using alternative datasets to use as fine-tuning data. TestPilot used synthetic data that was generated by another LLM, however this data could be swapped for alternative data depending on the testing strategies and coding guidelines the novice programmer should learn from. For example, a lecturer or teaching assistant could use data from a marking scheme which uses the "ideal" style of code they wish their students would follow.

Finally, future work may also involve creating a unit test assistant using LLMs for other programming languages. TestPilot focused on Java since it is one of the most popular programming languages used by novice programmers. There is a potential to create an alternative interface that works for other programming languages such as Python. A challenge with alternative languages is that there may be various testing frameworks that are commonly used. Therefore, a decision must be made to support multiple testing frameworks or a single testing framework. Nonetheless, it serves as an opportunity to transfer the benefits offered in TestPilot to different programming languages for novice programmers learning software testing.

Bibliography

[1] Buffardi et al. Effective and ineffective software testing behaviors by novice programmers. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, page 83–90, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322430. doi: 10.1145/2493394.2493406. URL `https://doi.org/10.1145/2493394.2493406`.

[2] Yang et al. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 391–402, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901767. URL `https://doi.org/10.1145/2901739.2901767`.

[3] Denny et al. Computing education in the era of generative ai. *Commun. ACM*, 67 (2):56–67, jan 2024. ISSN 0001-0782. doi: 10.1145/3624720. URL `https://doi.org/10.1145/3624720`.

[4] Muller et al. Exploring novice programmers' homework practices: Initial observations of information seeking behaviors. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 333–339, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366885. URL `https://doi.org/10.1145/3328778.3366885`.

[5] Skripchuk et al. Analysis of novices' web-based help-seeking behavior while programming. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 945–951, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394314. doi: 10.1145/3545945.3569852. URL `https://doi.org/10.1145/3545945.3569852`.

[6] Chatterjee et al. Finding help with programming errors: An exploratory study of novice software engineers' focus in stack overflow posts. *Journal of Systems and Software*, 159:110454, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110454. URL `https://www.sciencedirect.com/science/article/pii/S0164121219302286`.

[7] Alghamdi et al. Novice programmers strategies for online resource use and their impact on source code. In *2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 92–104, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi: 10.1109/CHASE58964.2023.00018. URL `https://doi.ieeecomputersociety.org/10.1109/CHASE58964.2023.00018`.

[8] Farooq et al. An evaluation framework and comparative analysis of the widely used first programming languages. *PLOS ONE*, 9(2):1–25, 02 2014. doi: 10.1371/journal.pone.0088941. URL https://doi.org/10.1371/journal.pone.0088941.

[9] Chen et al. Vscuda: Llm based cuda extension for visual studio code. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 11–17, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. doi: 10.1145/3624062.3624064. URL https://doi.org/10.1145/3624062.3624064.

[10] Shin et al. A model of how students engineer test cases with feedback. *ACM Trans. Comput. Educ.*, 24(1), jan 2024. doi: 10.1145/3628604. URL https://doi.org/10.1145/3628604.

[11] Bsharat et al. Principled instructions are all you need for questioning llama-1/2, gpt-3.5/4, 2024. URL https://doi.org/10.48550/arXiv.2312.16171.

[12] Zhou et al. Relying on the unreliable: The impact of language models' reluctance to express uncertainty, 2024. URL https://doi.org/10.48550/arXiv.2401.06730.

[13] Wang et al. Aligning large language models with human: A survey, 2023. URL https://doi.org/10.48550/arXiv.2307.12966.

[14] Jha et al. Dehallucinating large language models using formal methods guided iterative prompting. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*, pages 149–152, 2023. doi: 10.1109/ICAA58325.2023.00029.

[15] Chung et al. Increasing diversity while maintaining accuracy: Text data generation with large language models and human interventions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-long.34. URL http://dx.doi.org/10.18653/v1/2023.acl-long.34.

[16] Ma'ayan et al. The quality of junit tests: an empirical study report. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, SQUADE '18, page 33–36, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357371. doi: 10.1145/3194095.3194102. URL https://doi.org/10.1145/3194095.3194102.

[17] Stephen Wolfram. *What Is ChatGPT Doing ... and Why Does It Work?* Wolfram Media, Inc., 2023. URL https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/.

[18] Goodfellow et al. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[19] Radford et al. Improving language understanding by generative pre-training, 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[20] Ouyang et al. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation, 2023. URL https://doi.org/10.48550/arXiv.2308.02828.

[21] Luo et al. An empirical study of catastrophic forgetting in large language models during continual fine-tuning, 2023. URL `https://doi.org/10.48550/arXiv.2308.08747`.

[22] Brown et al. Language models are few-shot learners, 2020. URL `https://doi.org/10.48550/arXiv.2005.14165`.

[23] Touvron et al. Llama: Open and efficient foundation language models, 2023. URL `https://doi.org/10.48550/arXiv.2302.13971`.

[24] Wei et al. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL `https://doi.org/10.48550/arXiv.2201.11903`.

[25] Kojima et al. Large language models are zero-shot reasoners, 2023. URL `https://doi.org/10.48550/arXiv.2205.11916`.

[26] Kaplan et al. Scaling laws for neural language models, 2020. URL `https://doi.org/10.48550/arXiv.2001.08361`.

[27] Hoffmann et al. Training compute-optimal large language models, 2022. URL `https://doi.org/10.48550/arXiv.2203.15556`.

[28] Sorscher et al. Beyond neural scaling laws: beating power law scaling via data pruning, 2023. URL `https://doi.org/10.48550/arXiv.2206.14486`.

[29] Lu et al. Machine learning for synthetic data generation: A review, 2024. URL `https://doi.org/10.48550/arXiv.2302.04062`.

[30] Veselovsky et al. Generating faithful synthetic data with large language models: A case study in computational social science, 2023. URL `https://doi.org/10.48550/arXiv.2305.15041`.

[31] Chen et al. Evaluating large language models trained on code, 2021. URL `https://doi.org/10.48550/arXiv.2107.03374`.

[32] Kazemitabaar et al. Studying the effect of ai code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394215. doi: 10.1145/3544548.3580919. URL `https://doi.org/10.1145/3544548.3580919`.

[33] Ouyang et al. Training language models to follow instructions with human feedback, 2022. URL `https://doi.org/10.48550/arXiv.2203.02155`.

[34] Jalil et al. Chatgpt and software testing education: Promises & perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2023. doi: 10.1109/icstw58534.2023.00078. URL `http://dx.doi.org/10.1109/ICSTW58534.2023.00078`.

[35] Yilmaz et al. The effect of generative artificial intelligence (ai)-based tool use on students' computational thinking skills, programming self-efficacy and motivation. *Computers and Education: Artificial Intelligence*, 4:100147, 2023. ISSN 2666-920X. doi: https://doi.org/10.1016/j.caeai.2023.100147. URL `https://www.sciencedirect.com/science/article/pii/S2666920X23000267`.

[36] OpenAI. Gpt-4 technical report, 2024. URL https://cdn.openai.com/papers/gpt-4.pdf.

[37] Hellas et al. Exploring the responses of large language models to beginner programmers' help requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, ICER 2023. ACM, August 2023. doi: 10.1145/3568813.3600139. URL http://dx.doi.org/10.1145/3568813.3600139.

[38] Pankiewicz et al. Navigating compiler errors with ai assistance – a study of gpt hints in an introductory programming course, 2024. URL https://doi.org/10.48550/arXiv.2403.12737.

[39] Nam et al. Using an llm to help with code understanding, 2024. URL https://doi.org/10.48550/arXiv.2307.08177.

[40] Lee et al. The technology acceptance model: Past, present, and future. *Technology*, 12, 01 2003. doi: 10.17705/1CAIS.01250. URL https://doi.org/10.17705/1CAIS.01250.

[41] Druga et al. Scratch copilot evaluation: Assessing ai-assisted creative coding for families, 2023. URL https://doi.org/10.48550/arXiv.2305.10417.

[42] Resnick et al. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, nov 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592779. URL https://doi.org/10.1145/1592761.1592779.

[43] David Kirk. Nvidia cuda software and gpu parallel computing architecture, 10 2007. URL https://doi.org/10.1145/1296907.1296909.

[44] Tufano et al. Methods2test: a dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22. ACM, May 2022. doi: 10.1145/3524842.3528009. URL http://dx.doi.org/10.1145/3524842.3528009.

[45] Jiang et al. Mistral 7b, 2023. URL https://doi.org/10.48550/arXiv.2310.06825.

[46] Pornprasit et al. Gpt-3.5 for code review automation: How do few-shot learning, prompt design, and model fine-tuning impact their performance?, 2024. URL https://doi.org/10.48550/arXiv.2402.00905.

[47] OpenAI. Example count recommendations for fine-tuning, 2024. URL https://platform.openai.com/docs/guides/fine-tuning/example-count-recommendations.

[48] Renze et al. The effect of sampling temperature on problem solving in large language models, 2024. URL https://doi.org/10.48550/arXiv.2402.05201.

[49] OpenAI. Gpt api documentation, 2024. URL https://platform.openai.com/docs/models/.

[50] Anyscale. Mistral-7b-instruct-v0.1 api documentation, 2024. URL https://docs.endpoints.anyscale.com/text-generation/supported-models/mistralai-Mistral-7B-Instruct-v0.1.

[51] OpenAI. Openai platform for fine-tuning, 2024. URL https://platform.openai.com/finetune.

[52] Kevin Brock. The 'fizzbuzz' programming test: A case-based exploration of rhetorical style in code, 2016. URL http://computationalculture.net/the-fizzbuzz-programming-test-a-case-based-exploration-of-rhetorical-style-in-code/.

[53] DataStax. Astradb vector database documentation, 2024. URL https://docs.datastax.com/en/astra-serverless/docs/.

[54] OpenAI. Release of text-embedding-3-small, 2024. URL https://openai.com/blog/new-embedding-models-and-api-updates.

[55] JUnit 5. Junit 1.8.2 jar, 2021. URL https://mvnrepository.com/artifact/org.junit.platform/junit-platform-console-standalone/1.8.2.

[56] JaCoCo. Jacoco 0.8.11 jar, 2023. URL https://github.com/jacoco/jacoco/releases/tag/v0.8.11.

[57] Google. Google-java-format 1.21.0 jar, 2024. URL https://github.com/google/google-java-format/releases/tag/v1.21.0.

[58] Axios. Promise based http client for the browser, 2024. URL https://axios-http.com/docs/intro.

[59] Microsoft. Fluent ui web components, 2024. URL https://learn.microsoft.com/en-us/fluent-ui/web-components/components/overview.

[60] Nikhil Lohia. Leetcode "easy" java solutions, 2024. URL https://github.com/nikoo28/java-solutions/tree/master/src/main/java/leetcode/easy.

[61] Rodrigo Pombo. Exercism.org java solutions, 2023. URL https://github.com/RodrigoPombo1/Exercism-Java.

[62] Trinity College Dublin. Introduction to programming i (csu11011), 2024. URL https://teaching.scss.tcd.ie/module/csu11011-introduction-to-programming-i/.

[63] OpenAI. Online chatgpt interface, 2024. URL https://chat.openai.com/.

[64] Wolpert et al. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893. URL https://doi.org/10.1109/4235.585893.

[65] OpenAI. Openai blog, 2024. URL https://openai.com/blog.

A Appendix

A.1 TestPilot frontend source code

Source code for the frontend Visual Studio Code extension can be found at `https://github.com/saisankp/TestPilot/tree/main/TestPilot`.

A.2 TestPilot backend source code

Source code for the backend Python Flask server can be found at `https://github.com/saisankp/TestPilot/tree/main/server`.

A.3 Evaluation dataset

The evaluation dataset for TestPilot can be found at `https://github.com/saisankp/TestPilot/tree/main/server/evaluation/dataset`.

A.4 Evaluation results

The evaluation results can be found at `https://github.com/saisankp/TestPilot/tree/main/server/evaluation/results`.