Debugging Problems

Patrick Farmer

Supervisor: Dr. Jonathan Dukes

April 16, 2025

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

A Dissertation submitted in partial fulfillment of the requirements for the degree of MAI in Computer Engineering.

Declaration

I hereby declare that this Dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

I consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

I agree that this Dissertation will not be publicly available, but will be available to TCD staff and students in the University's open access institutional repository on the Trinity domain only, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Signed: Patrick Farmer Date: April 16, 2025

Contents

Acronyms

- AI - Artificial Intelligence
- API - Application Programming Interface
- AST - Abstract Syntax Tree
- CI/CD - Continuous Integration/Continuous Deployment
- CLI - Command Line Interface
- GPT - Generative Pre-trained Transformer
- LLM - Language Model

1 Introduction

1.1 Context

In recent years great strides made in the ability of LLMs to generate code. These LLMs are already being integrated into employee and organisational workflows in companies worldwide. This places a much greater emphasis on the debugging process as the code generated by these LLMs must be inspected and fixed when arise. It has also been noted in many studies that debugging is an extremely important skill as the most difficult part of the programming process. It is also the most often overlooked when teaching programming. More will be discussed on this in the literature review.

The goal of this project is to create a tool that will help teach debugging to students and enabling them develop debugging skills by generating new programs containing bugs. LLMs will also play a crucial role in the functionality of this project, serving as one of the core components of the tool itself. Their role in the project is to generate the working program and, if selected, to insert the bug as well.

1.2 Aims

Create a Tool to Enable Debugging Skills Development: The primary objective of this tool is to enable users to test and develop their debugging skills by generating debugging problems. The user will develop their debugging skills by solving these problems.

Customisable Generation: The tool should allow users to specify the topic and bug type for the generated code. An example output with the topic of the Fibonacci sequence

is shown below. [1]

```
    def fibonacci(n):
    """Return a list containing the first n Fibonacci numbers."""
    sequence = []
    a, b = 0, 1
    for _ in range(1): # Range 1 instead of n
        sequence.append(a)
        a, b = b, a - b  # Bug: Subtracting instead of adding
    return sequence

def main():
    try:
        with open('input.txt', 'r') as file:
            line = file.readline().strip(print) # Print added to
               strip function
            if not line:
                raise ValueError('Input file is empty.')
            n = int(line)
    except Exception as e:
        print(f'Error reading input.txt: {e}')
        return
    sequence = fibonacci(n)
    print(f"Fibonacci sequence with {n} term{'s' if n != 1 else ''}:
       ")
    print(' '.join(map(str, sequence)), fibonacci) # Fibbonacci
       added to print function

if __name__ == '__main__':
    main()
```

Investigate Multiple Bug Insertion Methods: It was also a goal to create two different methods for inserting the bug which can be evaluated and compared with each other.

Multiple Test Cases: It is a requirement of the project to provide a test case that users can use to verify they have resolved the bug. Additionally, the project aims to generate multiple test cases to validate the code against various scenarios and types of bugs.

Frontend is Out of Scope: The creation of a frontend was not a goal of this project, the meaningful work of this project is all undertaken in the backend with the generation of code and the insertion of bugs. A frontend could be added at a later date to improve the

[1] The comments pointing out the bugs were added for clarity, they are not part of the output.

accessability of the tool and also provide additional features such as access to a debugger.

1.3 Methodology Outline

The project was developed in Python. The first step is to collect inputs from the user. If no inputs are provided, default inputs are selected by the program instead. The necessary inputs are model type and problem count. There are also optional inputs for the topic, prompt overrides for both code creation and bug insertion, bug insertion type, test case count, hyperparameter values for the LLM and a few other unimportant parameters.

The program processes these inputs to generate a working piece of code for a given topic. It will do this by querying the LLM with the prompt or topic, if neither are provided it will select a default prompt.

The output of this working code for a variable number of test cases is then collected. This will be done by altering the input.txt file that is generated by the LLM. The output of each of these runs is saved.

The program attempts to insert a bug using either the LLM or the AST bug insertion method, depending on the input provided. The LLM if selected will use a default prompt or the prompt provided by the user if it exists. The AST bug insertion method will be discussed in more detail in the Design and Implementation section.

The test cases are then run to ensure that the code fails. If not, the previous step is repeated. If the code fails as it should, the bugged program is saved, and the program moves on to generate another problem.

1.4 Evaluation Metrics

The choice was made to use desktop metrics alone to measure and evaluate the success of the project. While a user study would provide insight into the user experience from a student perspective, the focus of the evaluation would be based on the characteristics of the code generated. The characteristic measured were selected to cover elements of factors that would influence the user experience. The selected metrics were 'code complexity', 'code diversity', 'attempt count', and 'run time'. The calculation of these metrics and the reasoning behind them is discussed in more detail in the testing and evaluation section.

1.5 Structure of the Dissertation

The structure of the dissertation will be as follows. The background will go through previous work done in the area of debugging and LLMS. It will also give a brief overview of how LLMs work and more importantly how they can be used to generate code, the difficulties that come with using them and where they excel.

The design and implementation will show how the code is structured and will give a brief description of how each part of the code works.

The testing and evaluation section will show the testing that was done on the project and the results of that testing. It will also discuss the metrics used in testing, why they were chosen and how they were calculated.

The conclusion will summarise the project and discuss the future of the project.

2 Background

2.1 Literature Review

2.1.1 Importance of Debugging Skills

Debugging has always been an essential aspect of programming, yet many universities do not teach it specifically. This is highlighted by Li in [2]. As previously mentioned, the significance of debugging skills has grown markedly now that AI is becoming more prevalent in industry, as noted by Denny in [8]. Debugging itself includes several sub-domains, as outlined in [2] as language knowledge, understanding of the specific program, and skill in the debugging process. While the primary focus of this project is to enhance debugging skills, it will also provide exposure to domain knowledge.

2.1.2 Novice Programming Errors and Debugging

When novice programmers write code, they inevitably introduce bugs, but as Jadud mentions in [1], they often respond by making minor syntax tweaks and rerunning their code immediately rather than trying to identify and resolve the underlying issue. This is another challenge that this tool addresses: by inserting deeper bugs, students are compelled to scrutinize and understand the code.

Although it is definitely beneficial to insert deep semantic bugs into the code to improve the students' debugging skills, it is also important to teach students how to notice and solve the bugs they will often cause in code they write thprovide This is a project similar to the tool discussed in this paper. The above paper discusses generating parsons problem specifically

while this project is focused on generating debugging problems more generally. It is useful to see that students found the tool enjoyable and beneficial. The particular aspect of the tool they found enjoyable was the ability to customise the topic. This provides good justification for the customisable topic in the tool developed in this project.

A study by Brown [16] challenges the common beliefs held about bugs written by beginners. It analyses a dataset contained over 100,000 blocks of code compiled by students, this is known as the Black Box Dataset. This study is focus on Java, this means that not every observation made is relevant for this project but still provides useful insights. The study found that the most common bugs written by students were mismatched brackets, wrong parameter types, missing return statements, incorrect function calls and confusing assignment with comparison ('=' vs '==='). When professors and novices were given the opportunity to predict the most common bugs there was no difference between the accuracy of their predictions. This is important as it shows that a data driven approach needs to be taken to identifying common bugs written by novices, general assumptions are not enough. Although this study set out to show the difference between common beliefs and the reality of the most common bugs. It backs up the previously made observations that the most common bugs written by novices are shallow bugs.

The above studies focus on the errors made by novices, however also worth mentioning is defects in code, Effenberger investigates this in [17]. This includes not just bugs but poor code too. This is interesting to note as it highlights issues with novice programmers code beyond just bugs. The vast majority of the defects were using while loops instead of for loops or vice versa, redundant blocks of code and poorly named variables. This is not directly addressed by the tool although this could be a piece of future work. However, since the tool is producing code that is written by an LLM, excluding the bug inserted into

the code the code should be high quality that very closely matches industry standards and practices while leaving very little or no redundant code. The exposure to correct style and practices will help students learn to write better code.

2.1.3 Teaching Debugging

As mentioned, an important purpose of this tool is to teach students how to work with and fix code generated by LLMs. For this reason, it is worth discussing how LLMs and developers will misunderstand each other, as this will affect which bugs are more beneficial for the user to learn to fix. This is discussed in [9]. Students often struggle to understand the code written by the LLM as the domain is not always something that they are familiar with. This project will enable the user to set a domain and language for the code so that they can focus on solving bugs and developing their debugging skills.

Some of the other bugs discussed in this paper that LLMs tend to write into code include overlooking edge cases. This is why it will be important to generate test cases to ensure that the code is working as expected. The final and most common bug is a simple misunderstanding of the prompt, where the LLM misunderstands the problem they are being given. This is something that the tool should be able to support by allowing professors to set custom prompts when they are generating code for the student to debug.

2.1.4 Test Cases in Debugging

Test cases are important to ensure that the code has been fixed, and they also serve the purpose of providing feedback to the user. As discussed by Janzen in [10], the test cases will be used to show the user what is going wrong with the code and in what circumstances

the code does or does not work. This is particularly important in an environment where the debugger is not enabled and the user must rely on the output of the code to determine if it is working correctly. This is why the test cases will be generated by the tool and will be used to evaluate the code that is generated by the LLMs. Having multiple test cases is crucial because each one can verify a different aspect of the code, ensuring that various scenarios are tested and that the code behaves as expected under different conditions. This is achieved by creating different input files to test how the code works in various scenarios.

2.1.5 Use of Debugging Tools

Another important aspect of debugging that this tool addresses is the use of debugging tools like Python's debugger, which are part of the sub-domains indicated by Li in [2]. Learning to effectively use these debuggers can considerably improve students' debugging abilities. However, Odell [3] argues that forcing students to think about the program without any tool-assisted help is the most effective instructional approach. Ultimately, each professor can decide whether to allow debugging tools, depending on the objectives of their particular assignment.

2.1.6 Misunderstandings Between AI and Users

As Nguyen states in [9], significant misunderstandings can easily arise between an AI and its user, which is especially true for novices who may struggle to articulate their problems thoroughly. This situation creates a chance to teach students how to harness LLMs effectively. The future frontend may benefit from incorporating a co-pilot-like assistance, allowing professors to enable or disable it depending on the complexity level they want to

set. A comparable methodology is described by Denny in [7], where students restricted to using LLMs alone learned to refine their prompts for more accurate output.

2.1.7 BugSpotter

BugSpotter is a similar tool that was develop to help students identify bugs in their code [13]. It will generate buggy code using an LLM. The student is then given the task to create test cases that expose the bug. This tool will provide a similar benefit to students as the tool developed in this project. This tool also had the resources to carry out a classroom study and an expert review. The expert review came to the conclusion that the tool produced high-quality exercise but there was a noticed different in quality between GPT-3.5 and GPT-4o despite 4o being a more expensive model. The classroom study was tested in an introductory C programming module with 741 students. The conclusion from the classroom study was that the problems were found to match the specified difficulty very well. This study shows the usefulness of a tool that can generate debugging problems for a student or class of students.

2.1.8 Summary

The literature review has shown the importance of debugging and the need for a targeted approach to teach debugging. The review also highlights that the bugs that novices write into their code are generally shallow, often simple syntax errors or smaller semantic errors, this is why the tool will focus on these types of bugs. The importance of test cases are also highlighted and the need for multiple test cases to improve the coverage. This knowledge inspired the design of the tool to support multiple test cases beyond the original 1 in the

prototype. The decision had already been made to use desktop metrics instead of a user study to evaluate the tool. However, the positive results of the BugSpotter tool show that a user study will likely yield good results.

2.2 LLMs and Code Generation

2.2.1 How LLMs work

LLMs in essentials are a type of neural network that is trained on a large dataset of text which is often code. They are trained to predict the next token in a sentence. The common chat bots works by rephrasing your input and allowing the LLM to predict the next token which when it continues will create a response to your question or prompt.

2.2.2 Interacting with LLMs

For interacting with an LLM you will generally use an API. This tool uses two different APIs to interact with LLMs which can be swapped in-between. The first API is the OpenAI GPT API which is a paid API that is very accurate and very fast, in an ideal world this would be used for all versions and iterations for the project. However, this is of course going to get very expensive publisher Association LLMs is used too. The interaction will involve initialising the LLM with a set of hyper-parameters and then simply re-using this object in subsequent queries. The invoke function will take the prompt as the sole input and return the response from the LLM. There will often have to be parsing of the output but there is all that is necessary to interact with the LLM.

2.2.3 Local LLMs

Local LLMs are models that are run locally on your own machine. They are generally much smaller and cheaper models. They are also generally slower unless access to multi-million euro hardware is available. This method is much slower and can be less accurate to varying degrees depending on the model but is free except for the small electricity cost and hardware wear and tear. Local LLMs are also more customisable with adjustable hyper-parameters and can be fine-tuned on custom datasets, although this was not necessary for this project. A variety of models using the Ollama API were used for the vast majority of the project. Koutcheme [14] discusses the use of local LLMs against expensive OpenAI models. High quality results comparable with gpt-3.5-turbo were found from Llama-3.1-70B and Mistral-7B. Both of these models were also used during the development of this project and the same conclusion was found for both of them. However, with the hardware was not able to run the 70B parameter model in a reasonable time so only the Mistral-7B model was used for final evaluations.

2.2.4 LLM Choices

There were six models used for the final evaluation. Two 7 billion parameter models, one 14 billion parameter models and two 32 billion parameter models. The size of the model is important as it will determine the amount of VRAM required to run the model. It is possible to use RAM instead of VRAM but for the already long run times of the models this would be infeasible.

The 7 billion parameter models were Mistral and Llama-3.1. Llama-3.1 is a simple reliable baseline model while Mistral is a more complex model but is the same size. This means that

it will have a longer runtime but requires the same amount of VRAM to hold the model. These would both be useful in settings with limited resources, if runtime is a concern Llama is likely to be better suited. If resources are limited but timing is not an issue Mistral is likely to be better suited.

The 14 billion parameter model used was Deepseek's r1 model. This model is built upon Llama-3.1:14b. This model is a reasoning model, it is designed to 'think' through its output and make improvements. This makes the runtime much longer but the output is also more accurate without having the increase the size of the model.

The 32 billion parameter models were the largest models that the machine had enough VRAM to run. The first was the qwen2.5:32b model and the second was the deepseek r1:32b model which is built upon qwen2.5:32b. This provides a direct comparison between reasoning and non reasoning models. They both require a GPU with a lot of VRAM, this makes them infeasible for a lot of settings but are worth investigating to show how performance scales with the size of the model.

2.2.5 Prompt Engineering

Effective interaction with an LLM required careful prompt design. Prompt engineering plays a critical role in guiding the model's outputs and can significantly impact its accuracy, relevance, and overall performance .The most important aspect of prompt engineering is ensuring to get your point across clearly but there are also more subtle aspects such as phrasing the prompt in a way that will get the model to 'think' deeper about the response. An example of this might be "Before you provide the final answer, please explain your reasoning step by step outlining how you arrived at your conclusion". This will cause

the model to break down its thought process into steps which will encourage a structured response which is more likely to provide what you want.

A few other examples of common prompt engineering techniques are role-playing and few-shot learning. Role-playing is when you tell the LLM what role they should play such as "Act as a teacher for computer scientists". This will cause the LLM to respond in a way more suited to the context. With the OpenAI API this is a built in features. Alongside the hyper-parameters you can give provide a 'system role'. Few-shot learning is when you provide the LLM with a few examples of what you want it to do. This is very useful in the case of this tool. If a professor wants to target a given level of complexity they can provide a couple examples in the prompt to very clearly show what they want.

3 Design and Implementation

The project was designed to achieve the aims as discussed. The key aims being to generate code for a given topic, insert a bug into the code and then provide a test case to ensure that the code is working correctly.

3.0.1 The Code Generator

- The code generator takes simple parameters such as model type, bug type, code topic, and options for overriding default prompts.
- It produces functional code based on the specified topic by querying the LLM, parsing the response, and saving the code to a newly created Python file.
- The output and input to this Python script are also saved upon successful compilation.
- **OpenAI Code Generation:** Uses the OpenAI GPT API to generate code with high accuracy and speed.
- **Ollama Code Generation:** Uses local LLMs for code generation, which are slower but more cost-effective and customizable.

3.0.2 The Bug Inserter

- The bug inserter reads the functional code file and inserts bugs based on the specified method.
- **LLM Bug Insertion:** Queries the LLM to insert a bug, parses the response, and saves the buggy script. The code is then run to ensure it fails.

- **AST Bug Insertion:** Uses Abstract Syntax Trees to insert bugs by replacing specific nodes or values with others. The buggy script is saved over the original file.

3.0.3 The Code Evaluator

- Reads all generated files and evaluates them based on several metrics.
- **Diversity Metric:** Compares each file one by one to calculate diversity. Although inefficient, the LLM runtime dominates overall execution time.
- **Complexity Metric:** Uses the Radon library to calculate cyclomatic complexity and cognitive complexity for each file.
- **Attempt Count:** Tracks the number of attempts required to generate the original code and insert the bug.
- All metrics and their averages across debugging problems are saved to a CSV file.

3.1 Design Overview

The Bug Inserter will then take this working code and insert a bug into it, either by using the LLM again or by using inspecting and altering the AST in the code. It will then check if the code now fails or produces the wrong output as it should do. This is only done for the LLM however as the AST bug inserter is not prone to the same reliability issues of the LLM. This altered code is then written back out to the file overwriting the original code. The Code Evaluator will then take this code and run the diversity and complexity metrics on it before writing this to the CSV.

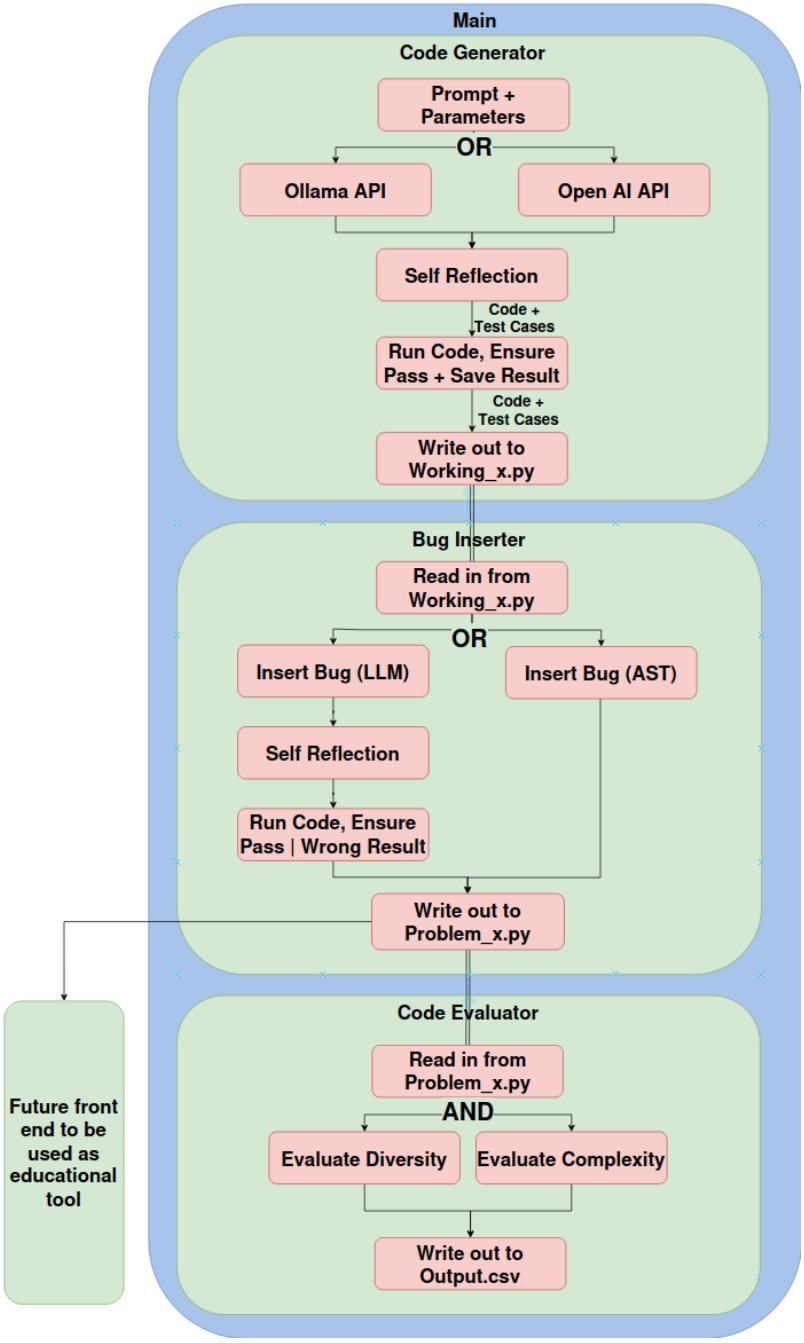


Figure 1: Design Overview

3.2 Tools

3.2.1 Ollama

The first of the two major tools that was used in this project was Ollama. This is a tool that facilitates easy interaction with LLMs running on a local machine. This tool allows for the usage of almost all popular models. This tool was downloaded by cloning the Ollama repository from Github and running the install script. Further models could then easily be pulled and run from command line using the following commands.

```
ollama pull <model>
ollama run <model>
<Interaction with LLM>
```

This works perfectly well for interacting with the LLM like a chat bot, building upon this would require building upon bash scripting which is an unnecessary complexity. Fortunately the Ollama API is much better suited to this task. The constructor for the class is called with some intialising hyper-parameters and then a prompt can simply be passed to using the invoke method which returns the response from the LLM. This was then wrapped in a class for easy use in the project.

Popular alternatives that exist are vLLM and llama.cpp. They both provide the same functionality but are not as easy to setup and use as Ollama. They are also able to take advantage of the hardware more affectively in most cases. However, for neither of these were viable for this project as the computer used has an AMD GPU and Ollama is the only tool that provides built in ROCM support which is required to run the models on AMD GPUs.

2 3 4

3.2.2 OpenAI GPT API

The process for using the OpenAI GPT API is a bit simpler since there is no setup but has a few more steps in the code for the protection of the API key. Due to this code being on a Github repo and automated testing being done using Github workflows there is a couple different environments that all need access to the API key. For the Github workflow runner the key was stored as a secret and then passed to the code as an environment variable. For the local machine the key was stored in a file which was excluded from git commits using the .gitignore file. One this has been handled the interaction is very similar. The parameters, prompt and API key are passed to the constructor and then a function is called to get the response from the LLM.

3.3 Code generation

The code generation class is the first class called in the pipeline and also the only class that interacts directly with the LLM. The class is called with the model type, file_path and a number of different switches to enable different modes. The prompt is then adapted and phrased in a way that the code produced will always expect an input file called `input.txt`.

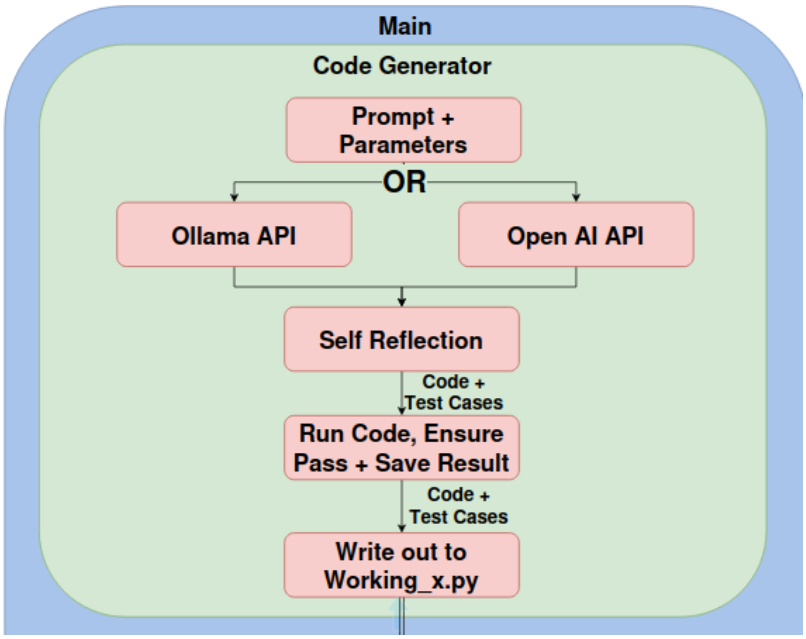
The LLM is then called to generate the code and also generate the `input.txt` file. Both of these responses are then parsed and written to their respective files. The code is then run from a subprocess which will check if the code compiles and runs correctly. If the code

[2] https://github.com/vllm-project/vllm
[3] https://github.com/ggml-org/llama.cpp
[4] https://www.amd.com/en/products/software/rocm.html

does not compile or run correctly the function will return an error and the parent function will handle it.

If it works correctly the function will return the response with the given `input.txt` which is used to determine later whether the bug inserter has worked correctly. Due to the input being in the format of a separate file it is very easy for multiple `input.txt` files to be generated and used as multiple different test cases.

There is also an option for a prompt to be passed to the class which will override the original prompt. This is how the LLM bug insertion is able to interact with the LLM without creating duplicate logic.



3.3.1 Self Reflection & Improvement

As the project progressed it was clear that the local LLMs were having a lot of difficulty producing the correct code. This is very uncommmon with the OpenAI modules but to

solve this self reflection and improvement was implemented in the code.

As it became impossible to feed the LLMs more information in training a new technique emerged when developers would get the LLM to 'think' about their response. This is done by re-querying the LLM after it returns a response and asking it if the response it gave fits the original prompt. This process is called self reflection [11] which is good for filtering out bad responses. Another similar method is called self improvement which is where the LLM is asked to improve on its response. This will attempt to fix bad responses and also attempt to improve already acceptable responses.

The issue that was noticed with this method was a significant increase in the runtime of the project. However it also meant that smaller modeuls could be used and provide better performance. It is not clear which method would be likely to yield a better response but the performance will be compared and discussed in the testing and evaluation section.

A conversation with an LLM using self reflection and improvement would look like this:

"Write a python function that checks if a function is prime"

```
    def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

"Does this code work correctly and cover all edge cases"

"No, ...(some explanation)"

"This is the prompt 'Write a python function that checks if a function is prime', 'This is the code' (include code), Improve the code to better suit the prompt"

```
    def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

3.4 Bug Insertion

3.4.1 LLM Bug Insertion

The first method for bug insertion that was used was the LLM bug inserter. This method is very simple as it uses the same function as the code generation step but with a switch to override the prompt. This will mean it can then very easily run the same test cases and check if the code now fails or produces the wrong output. If the result is the same as before it will be assumed that the bug was either not inserted or does not necessarily change the output. This was a simple addition as there was little additional code to be written but it was also quite unreliable. Although it was in theory completing a much smaller task than the code generation step, it saw a lot more issues with the LLMs hallucinating and providing incorrect responses. This was likely due to the task was a bit less intuitive as there is a lot less information to go on when inserting a bug than when generating code, and as a result less information in the training data and therefore less accurate responses in this case.

3.4.2 AST Bug Insertion

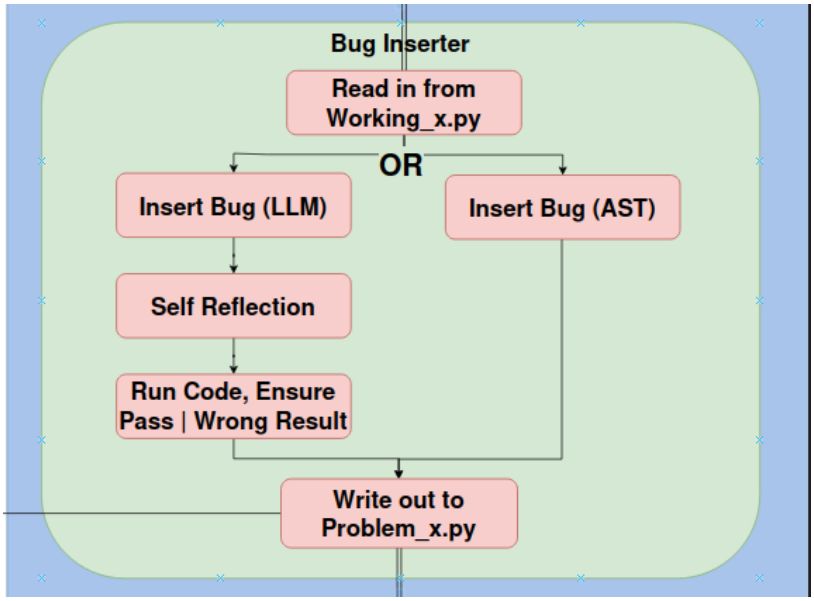
The second method for bug insertion was the AST bug inserter. An abstract syntax tree is a tree representation of a piece of code. It can represent the syntax and logical structure of the code but does not consider its formatting. This is beneficial for analysis of a piece of code. This is why linters and formatters often using ASTs.

This method was much more reliable and also massively faster. However, it has the disadvantage of being much less customisable than the LLM method. There is two different bug types that can be inserted. There is a syntax bug and a semantic bug type. If the syntax bug type is chosen the AST tree will be walked and when a certain keyword is found it will be changed to something else. This is not a difficult task to solve but for very novice programmers it might provide some useful practice.

The semantic bug type provides a much more useful and complex bug. It will walk through the AST and will insert a number of different bugs, there is 5 bug types. The first is a sign bug which will change the sign of a number or operator between two values, this will likely cause some indexing errors and some miscalculations of the result. The second bug either removes an argument from a function call or adds a new argument. It randomly decides whether to remove an existing argument or add a new one from the available variables in the code, this will break function calls and the developer will have to understand both the parent and child function to fix the bug. The third bug type will flip the value of a 1 or 0 to the opposite. This is likely to cause some off by one errors and also some indexing errors. The fourth bug is changing the name of the function called to another function defined in the code, this is not the most complex bug but will force the developer to atleast get a shalow understanding of the functions to solve it. The final bug type alters the range for a for loop and causes, this will cause indexing errors, off by one errors and also some

miscalculations of the result.

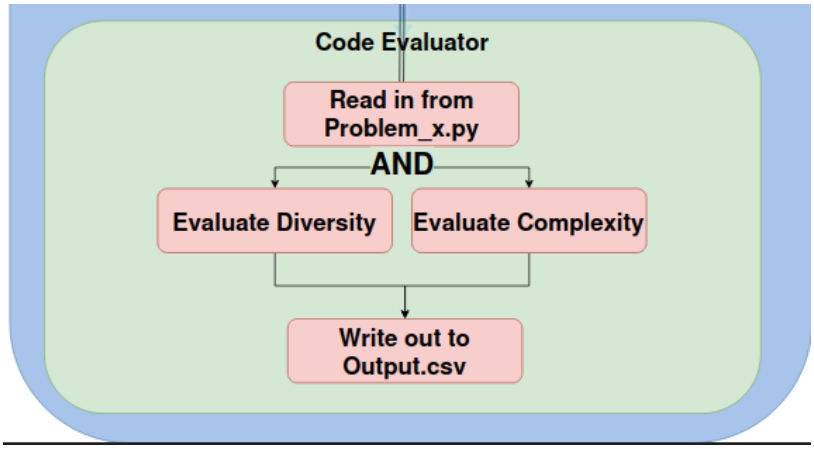
The code does not compile the output from the AST bug inserter as it assumes that a bug has been inserted. In development of this tool it was found that the bug was inserted correctly every time it was run, this is no garauntee that a bug will be inserted but a case where it did not work was not observed. However, this is for the currently small set of bug options, if this tool was to be expanded to include more complex bug insertion it may become possible and reasonably likely that a bug would not be inserted correctly. This would then require a re-run of the debugging problem to ensure the code was inserted correctly.



4 Evaluation Methods

This will discus the testing of the project which will show the improvement of the project over time. This will also include an evalution of the final state of the project.

4.1 Evaluation Overview



4.2 Environments

4.2.1 Environments Overview

There were two main environments used in this project. The first was the local machine which was used for development and implemented of new features. The second was Github workflows which was used for automated testing. This was used as the manual running of benchmarks was becoming a manually intensive task. It also had the benefit of the results of the run being saved as artifacts on Github which can easily be accessed and compared long after the run has finished. This was particularly useful for lengthy runs that were

carried out overnight.

4.2.2 Github Workflows

Github Workflows are a feature of Github that allows for the automation of tasks. It is commonly used in CI/CD pipelines to automate the testing of code before merges and deployments. Two workflows were created. A first workflow was created that would just run the main simple run script. This was run after each commit into a branch with an open pull request. This test would also have to pass before the pull request can be merged. The second workflow was created to run the benchmark test suite which is involves running the main script five or ten times with different parameters.

There were however a few issues with the Github workflow. The first issue was the machines that Github allocates for users to use are very slow and do not have a GPU. This made the running of the LLMs extremely slow to the point it was unusable. The initial solution to this was to use the OpenAI API for this but the cost of this was very high and was not feasible for a benchmark that might be run on a weekly cadence.

4.2.3 Self Hosted Runner

The solution was to connect the local machine to Github as a runner so that jobs could be sent to the local machine from Github. This did introduce a security concern where any user would be able to create a fork of the repository and run potentially malicious code on the local machine. This was solved by only allowing the runner to be used by the main branch and not any forks.

The process from committing new code and running the benchmark or other workflows is as follows. First the code is committed to the local machine and pushed to Github as normal. Assuming the push was to a branch with an open pull request the simple main workflow will be run. If the user wishes they can then open the workflows tab on Github and manually trigger the benchmark workflow with some inputs. This workflow can be run on any branch, including main.

The self hosted runner will then clone the repository into an new directory it creates, the timestamp is included in the directory name to ensure that newer runs do not overwrite previous runs which they do by default. The environment setup will then be run, this is simply creating a virtual environment and installing the requirements.txt file.

With the environment setup the script will then be run with the given parameters. The results of this run will then be moved to a local directory called artifacts which is then uploaded to Github as an artifact. These artifacts can then be accessed from the Github website, they can also be inspected locally, the new directory created has its path printed to terminal so if the developer wants to investigate some run in more detail and they have access to the machine they can do so.

4.3 Benchmark Test Suite

The decision to not evaluate this project with user feedback meant that desktop metrics were created to review the performance of the tool for a given set of parameters. The metrics that were chosen were code complexity, code diversity, attempt count and run time. The code complexity metric was chosen as it is a good indicator of how difficult the code is to debug. The code diversity metric was chosen as it indicates how useful the tool would

be for either a teacher who wants to generate a different problem for each student or a student who wants to generate a couple of different problems for themselves. The attempt count metric was also selected, this is the number of times the LLM had to be re-prompted because it produced an incorrect or unexpected output. This was primarily a good metric to compare different models and also to compare the self reflection and improvement methods. The final metric was run time, this was chosen as it shows how viable a given model is, this metric along with the rest will be used to weigh up the benefit of run time against the quality of the output. Below is a diagram showing the different layers of testing that can be done on the project.

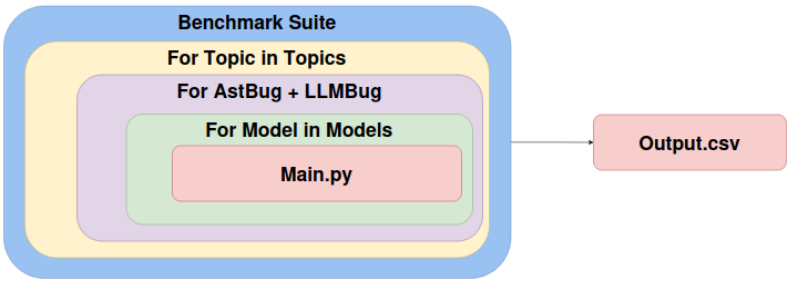


Figure 2: Benchmark Results

The smallest possible test is to just run the main.py file. This will simply take a default topic which was set to be the fibbonaci sequence and generate problematic code for this topic. This was the test run when implementing new features and solving the bugs that were inevitably added with the new features. This would naturally be much faster than the other tests as only one script iteration is run.

The first layer on top of this is a model benchmark. This will loop through a set of models that each run the main.py script with the same parameters. This will show which model is able to create the best results and do this is an acceptable time frame. The selection of

models chosen was the 32 billion parameter qwen model, the deepseek model built on top of this, the 14b parameter deepseek model, the mistral small model and the 7b llama3.1 model. The larger models were expected to perform better and slower than the smaller models. However this was not entirely consistent as will be discussed in the results section.

The second layer on top is very simple it will run the models first with abstract syntax tree bug insertion and secondly with LLM bug insertion. This will show the metrics of each method across a number of models. The results of this will be discussed in the results section.

The final layer is a loop that will run the second layer benchmark with a number of topics. This will show the metrics of each method across a number of models and a number of topics. There is little difference in the results when comparing against different topics but it is an important test to run in case some models are better at generating code for some topics than others.

4.4 Metrics

4.4.1 Code Complexity

The code complexity score is calculated using the Radon Python library. The cyclomatic complexity [12] is calculated for each file using the cc visit function which takes the code in text format as an argument and will return a complexity score. This is a metric that measures the amount of independent paths through the code.

The same is done for the cognitive complexity. This is a metric that attempts to measure how difficult it is to understand the code. In the output CSV for each run the cognitive

and cyclomatic complexity of each problem is given but for the higher layer tests only the average of the complexities is shown.

The complexity score has no ideal value, it is simply an indicator of how complex or otherwise the code generated is. In some settings a high complexity may be desired while in other settings a low complexity may be desired. [5]

4.4.2 Code Diversity

The code diversity score is calculated individually for each file, if the score is high it is more different from the other files. For each file it will compare it one by one to each of the other files, the results are then averaged to get the diversity of the file. The diversity is calculated one by one in this method because the Levenstein distance function in the text distance Python library only supports one to one comparison. If the diversity for a given file is below some threshold the code diversify class is used to re-prompt the LLM with the code and a prompt to create a more unique program. The diversity score itself is actually as similarity score so the lower the score of the similarity the more diverse and the better the results are.

4.4.3 Attempt Count

The attempt count metric is split up into two sections. There is the code generation attempt count which will give the amount of times that the LLM had to be re-prompted in order to create the code. This is generally very low and the majority of models will generate the original code with the first response most of the time. The second attempt count is the

[5] https://radon.readthedocs.io/en/latest/

bug insertion attempt count. This is only relevant for the LLM bug insertion, it will often require 2 or 3 re-prompts of the LLM until it will produce the desired code in the correct format. This metric is important as it will indicate if a model is too simple for the task being given. The lower the attempt count is the better as it means the program was able to produce the correct results with less queries

4.4.4 Run Time

The run time metric is very simply the length of time it takes a given run of main.py to complete. This metric is counted in seconds as the time taken to query an LLM and for it to respond can often take 5-200 seconds with the models selected. Seconds will give the most appropriate level of detail for this metric. Of course a larger run time is bad and a lower run time is good.

5 Results and Discussion

5.1 Evaluation of LLMs

Using an LLM is very easy. You simply give the LLM the prompt and read the output. However, there are some difficulties especially with the local LLM. Although the reponses were generally good, they could not simply be taken to be in the expected format. LLMs often provide a response not in the desired format, for that reason there is a lot of code which is used to clean up and parse the response from the LLM. This also leads to the issue of sometimes the LLM will hallucinate and provide an incorrect response, this is the reason for the implementation of self reflection and improvement in the code generation which will be discussed further in the design and implementation section. However, the simple and very significant benefit of using an LLM is that it can generate code with very little effort which makes it the only option for the generation of code for this project.

5.2 Hyperparameter Tuning

From all of the above testing a lot of data for model performance was collected. There has been no testing for the optimal hyper-parameters for these models though. This will only be done for the best performing model in the previous examples. There is clear best model but considering the trade off between run time, reliability (attempts needed) and diversity the qwen2.5:32b model gives reliable, high quality results and does not take too excessive a quantity of time like the deepseek models. For that reason the hyperparameter-tuning will be performed on the qwen2.5:32b model using AST bug insertion. The parameters that will be tuned are the temperature, the max tokens, the top p and the top k. The temperature

controls the amount of randomness in the output, the max tokens controls the maximum length of the output, the top p controls the probability of the output token being selected and the top k controls the number of tokens that are considered for the output. The tuning will be done using a grid search approach where the model will be run with a large set of possible combination of hyper-parameters and the results will be compared. Each bar of each graph has the average results of the program with the current hyperparameter at the given value. The graphs for the hyperparameter tuning for each hyperparameter are shown below:

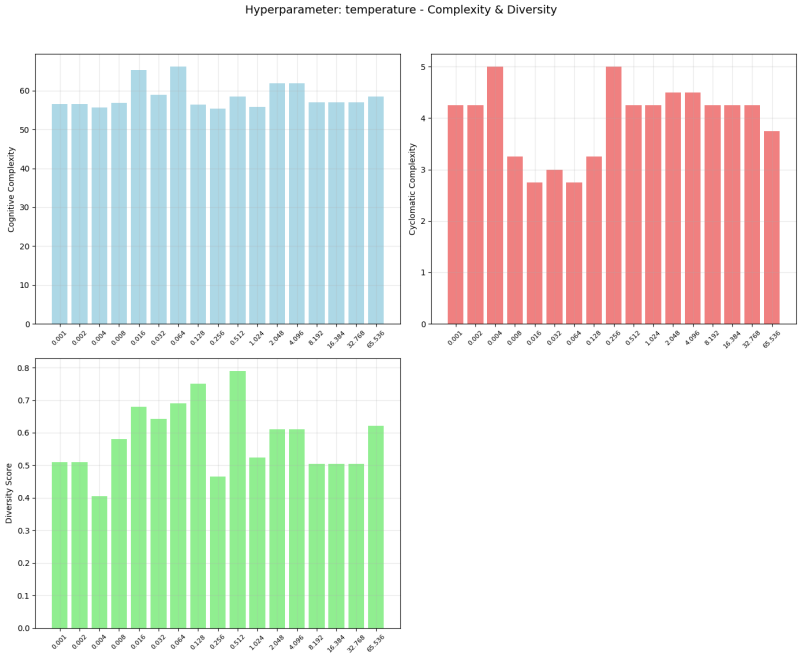


Figure 3: Complexity Comparison Threads

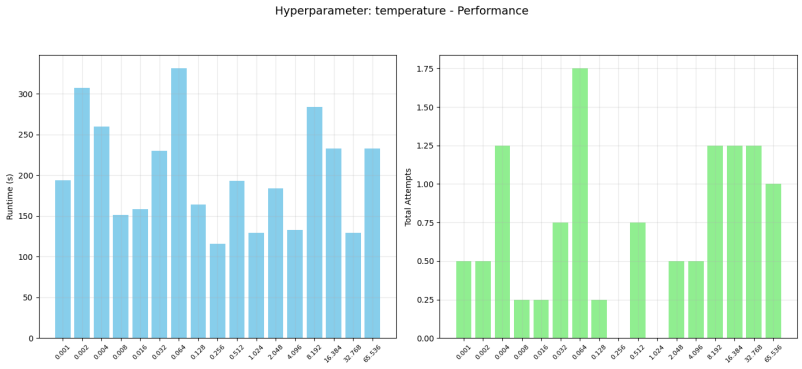


Figure 4: Complexity Comparison Threads

As mentioned above temperature is the amount of randomness in the output. From this we might expect to see that the attempts required and resultant run time is a bit higher with a higher temperature. We would also expect to see a higher diversity score at higher temperatures. We can see in the first chart that once the temperature reaches approximately 8 the accuracy does begin to fall. We do not however see the expected results for diversity, this is likely because the topic is quite constrained and the bug insertion is done using the AST bug insertion which is not affected by temperature. For this hyperparmeter there does not appear to be a clear best value but good results are seen at approximately 0.25 so this will be the value chosen.

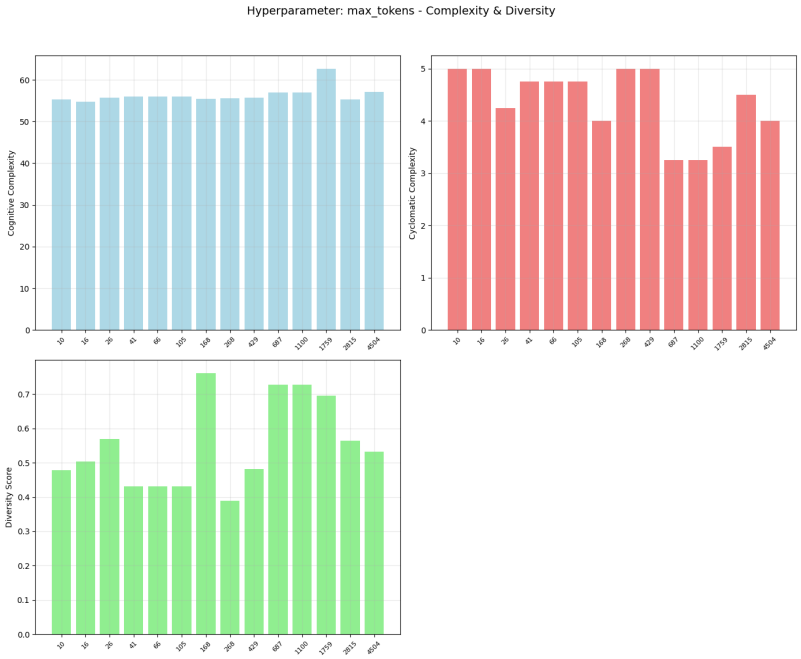


Figure 5: Complexity Comparison Threads

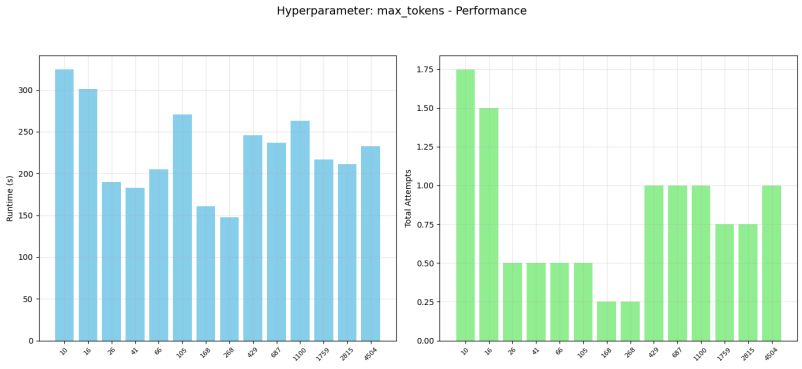


Figure 6: Complexity Comparison Threads

The max tokens hyperparameter is the maximum length of the output. For this we would expect to see very poor results while the max tokens is very low as the code will be cut off before it is complete. We would not expect to see much difference in the second graph as the diversity and complexity of the code should not be affected by the max token count. In the first graph we do see what we expect. Initially there is very poor results as the code is not able to be generated with the given max tokens. However, once the max tokens reaches an acceptable threshold the results get much better as most responses do not get cut off. This then slowly improves as the max tokens increases and the few longer responses begin to not be cut off either. Suprisingly though we do see a sudden significant increase in the attempts and run count after 400 tokens. We know from the increase in both the attempt count and the runtime that this is due to failed responses and not an increase in response

time from the LLM. It could be due to the LLM beginning to ramble and providing lower quality responses as the max tokens increases. The increase in diversity in the second graph provides some evidence for this as the responses are becoming more varied. Taking this the hyperparameter for this will be set to 300 to get the best length support for responses without a loss of accuracy.

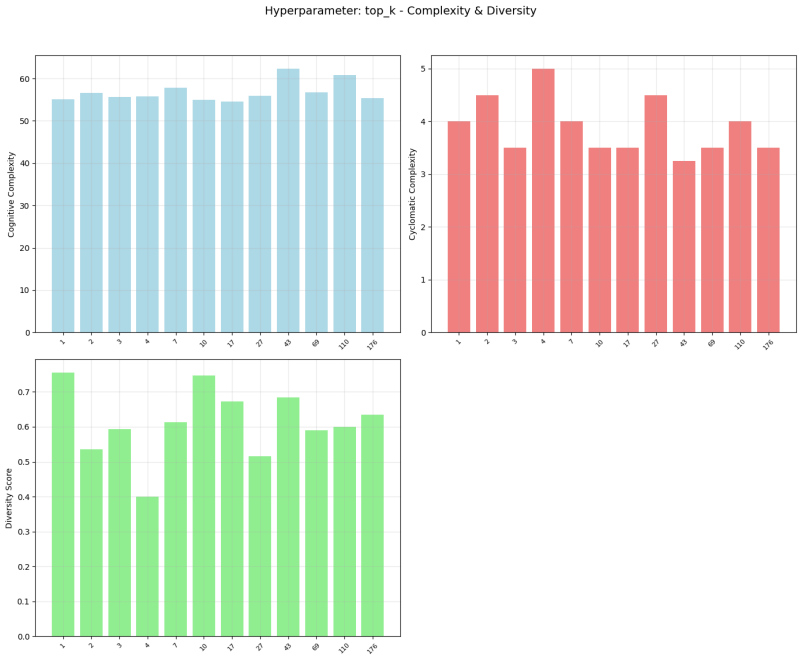


Figure 7: Complexity Comparison Threads

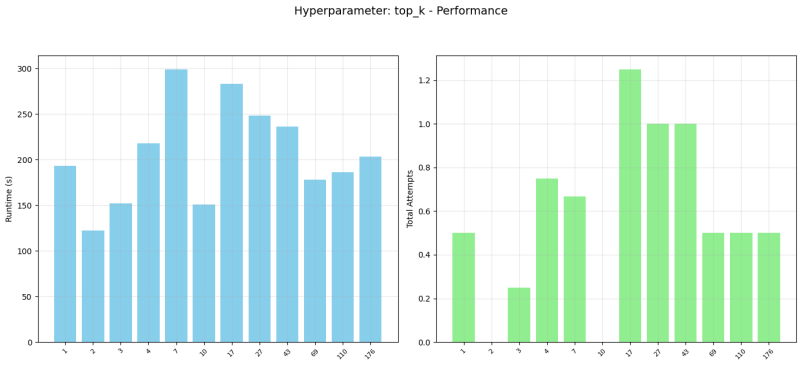


Figure 8: Complexity Comparison Threads

When the model is predicting the next token it will assign possibilities to all tokens. The top k hyperparameter will limit the number of tokens that are considered for the output to the top k. This has a similar affect to temperature. A decrease in accuracy as top k increases can be observed, once it goes above 3 we can see that the average attempts and runtimes begins to increase. The diversity score however, does not appear to be affected so for this hyperparameter it will be best to keep it at 1 or 2.

Figure 9: Complexity Comparison Threads

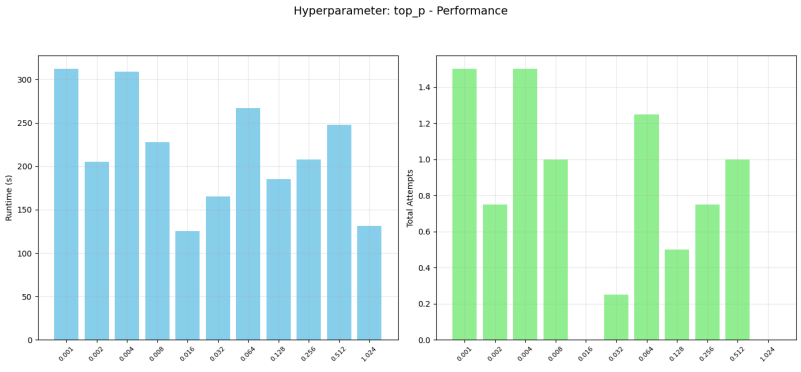
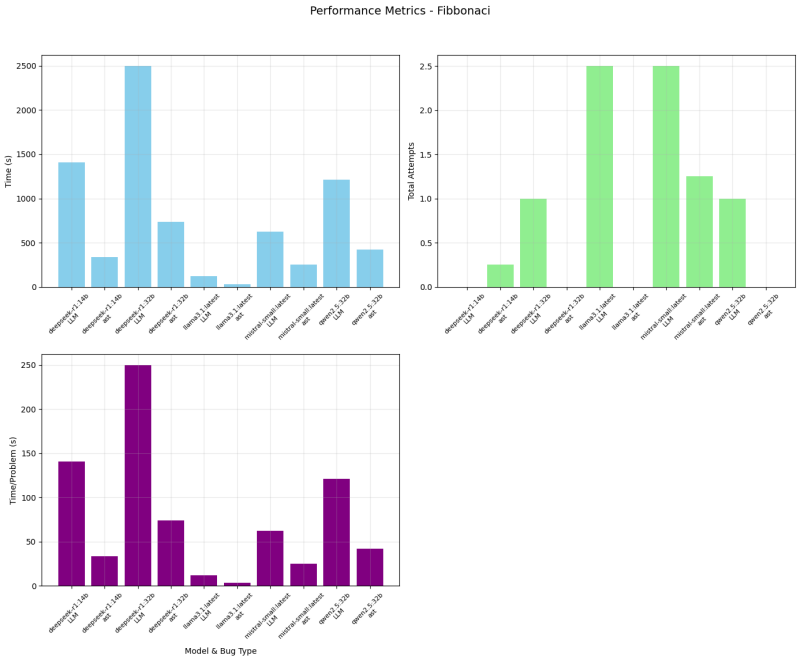


Figure 10: Complexity Comparison Threads

Similarly to top k the top p hyperparameter will limit the number of tokens that are considered. However, the top p will limit the tokens to that percentage of probabilities. That is, for a top p of 0.3 and if the first token has 0.2 probability and the second token has 0.1 probability these will be the only tokens considered. Opposite effects to that of top k are seen here. The lower the top p the more constrained the output is and this would lead one to this the responses would be more likely to be successful which would lead to shorter runtimes and less attempts needed. Though here we see high runtimes and retry counts with a low p. Once we get beyond the initial few values we see an increase in the accuracy of the responses which appears to hit a best accuracy at a top p of 0.15 - 0.3. The diversity score much like the top k is not affected by the top p. For this hyperparameter the value chosen will be 0.2.

5.3 Model Benchmark

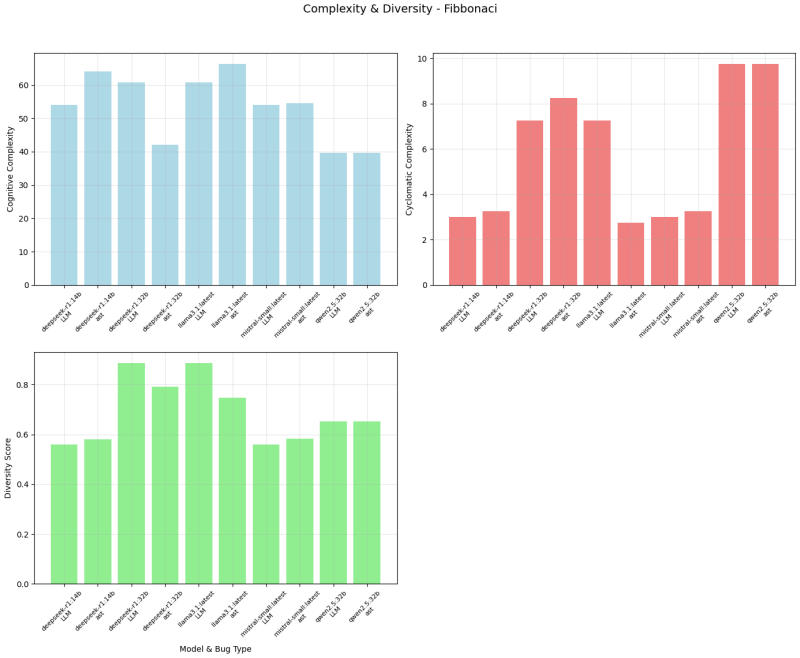
From the result of the model benchmark script the below graph was generate which compares the time taken, attempts taken and diversity score for each model with both LLM bug insertion and AST bug insertion.



From the graph above we see the very predictable outcome that in general the larger models will have larger runtimes. We also as expected see that the reasoning models will have substantially longer runtimes then equally sized non reasoning models. This is because the reasoning models will re-prompt a number of times over to try and improve the response as much as possible before providing a final output. The shortest runtime is from the llama3.1 model with AST bug insertion. As this is the smallest model this is expected.

Across the board we can see a very large difference between the AST bug insertion and

LLM bug insertion, for each model we can see that the runtime of the LLM bug insertion is at least two times larger, in some cases as much as four times larger. The same is seen for the attempts needed as the vast majority of the re-prompts are during the bug insertion, there is rarely a need for a retry in original code generation. We can see from this that the AST bug insertion is much more reliable and much faster than the LLM bug insertion. In the below graph we can see the metrics for cognitive complexity, cyclomatic complexity and diversity for each model which are the best indicator of the quality of the code generated.



The size of the model and the type of model has little effect on the cognitive complexity, this is because the task given to all of them was the same and there was little scope for creativity, it is possible that there would be further difference between the models with a more open ended topic which will be discussed in the next section. The cyclomatic complexity does seem to increase with the size of the model, this is likely because the larger

models are generating more parameterised code that will have greater levels of nesting than the smaller models that will be more likely to generate simpler code. The diversity score also is larger with the larger models which can be attributed to the same reason as the cyclomatic complexity, the increased complexities of the code generated by the larger models will lead to a greater diversity in the code generated.

We can see that the LLM bug insertion method tends to have a higher cognitive complexity. This is likely because the bugs that it is inserting are adding more complexity to the code, whereas the AST bug insertion method is only altering working parts of the code to cause errors. The opposite result is seen for the cyclomatic complexity, this will again be because of the types of bug that are inserted. Two of the possible bug types for AST can affect the functioning of a for loop and there is also a bug type that will affect function calls which can also affect cylomatic complexity. The diversity score for both methods is quite similar,it will change from model to model which has higher diversity which is well within margin of error for the test which is not as precise as the two complexity measurements.

5.4 Topic Benchmark

The results of the Models with different bug insertion methods above were run for one topic. The same test was run a number more times with different topics and bug prompts. Some topics will have more or less scope for creativity and complexity which might provide an interesting comparison between the bug insertion methods under different circumstances.

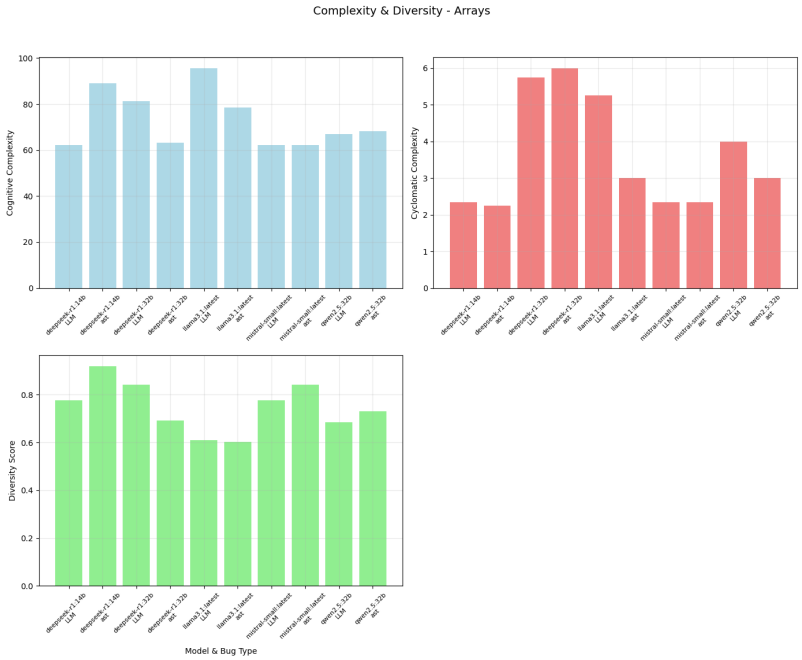


Figure 11: Complexity Comparison Arrays

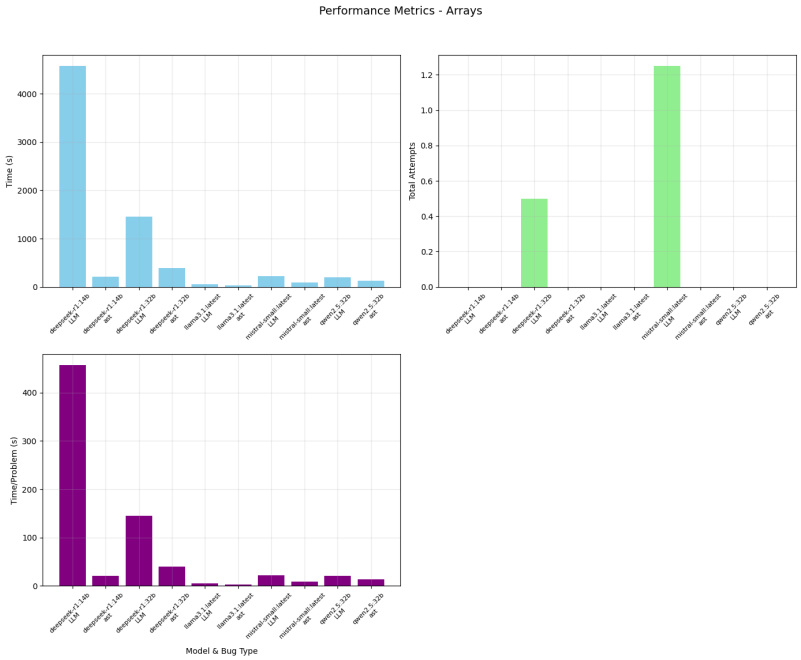


Figure 12: Model Comparison Arrays

The results of both graphs in this case are very similar to that of the baseline. They follow the exact same trend but the scale is different. The results of the array topic take longer to generate, take more retries and provide more complex code, the diversity however is not improved from the baseline.

The next topic that was tested was the Djikstra algorithm. From the broad phrasing of the topic we can expect to see more meaningful results in this case.

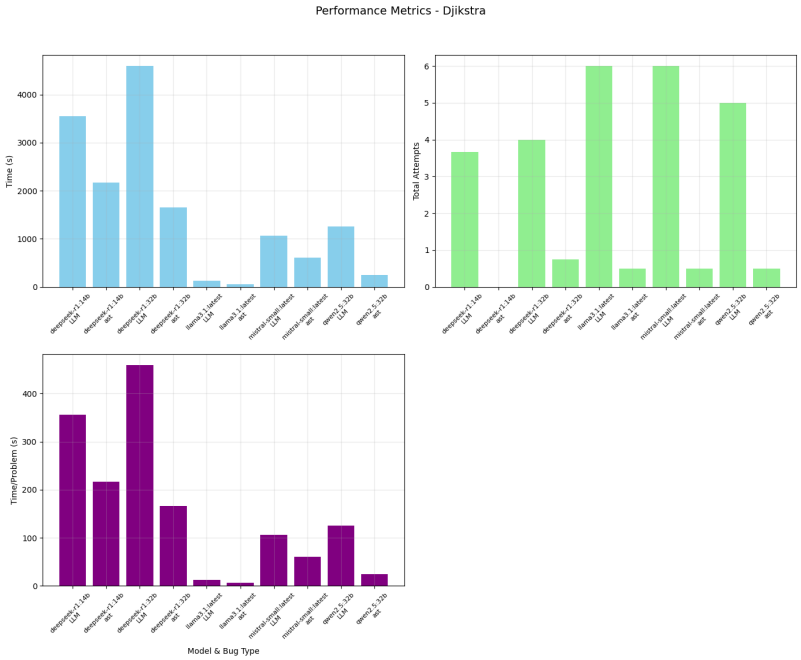


Figure 13: Model Comparison Djikstra

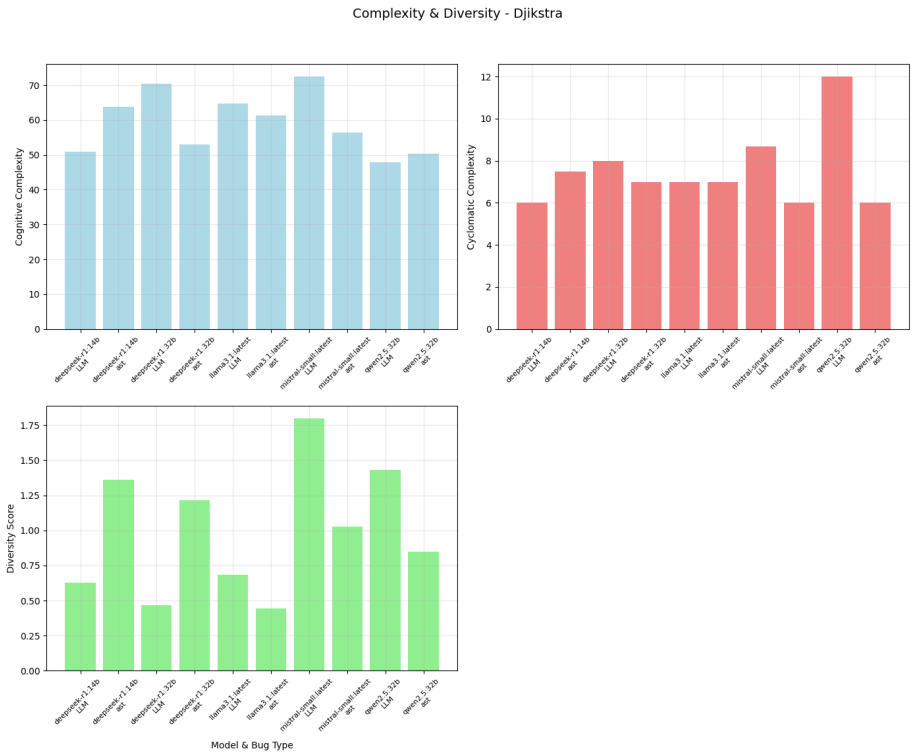


Figure 14: Complexity Comparison Djikstra

Much like above these graphs too follow the same trend across models but the scale is different again. There runtime and attempts required are both much higher than the baseline. this is due to the task being more complex, this is clearly only the case in the LLM bug insertion as there is no increase in attempts for the AST bug insertion, from this we can take that the LLM bug insertion becomes much less feasible for larger more complex code while the AST bug insertion will perform consistently in any setting. From the second graph we also see the expected increase in complexity and diversity from the baseline.

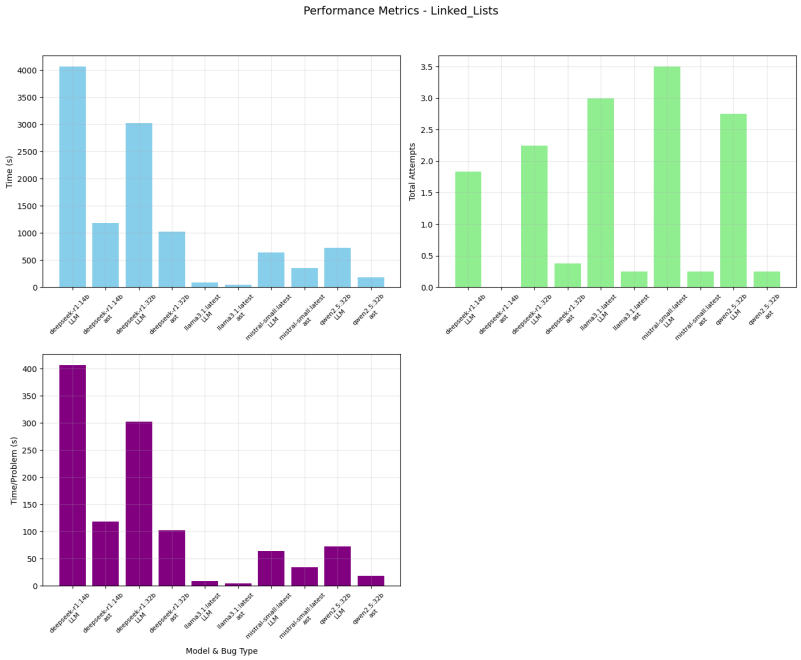


Figure 15: Model Comparison Linked Lists

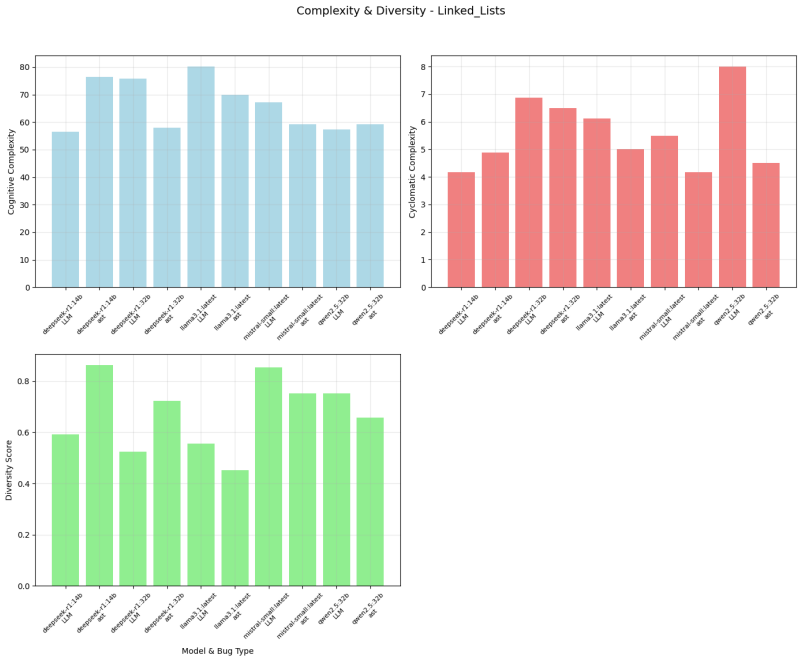


Figure 16: Complexity Comparison Linked Lists

The linked lists topic had performance between the baseline and the djikstra topic. The runtime and attempts required were comfortably higher than the baseline but not quite as high as djikstra, particularly for the attempts required. The complexity and diversity scores were very similar to the baseline and were not increased significantly.

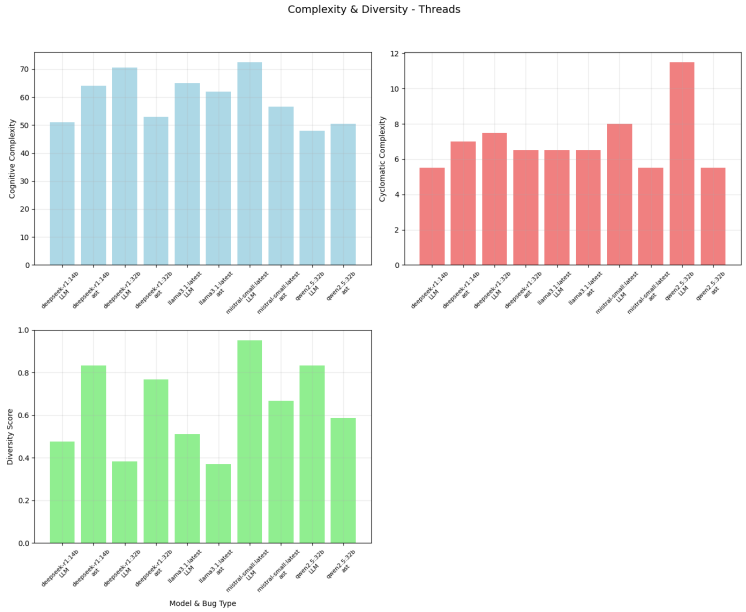


Figure 17: Complexity Comparison Threads

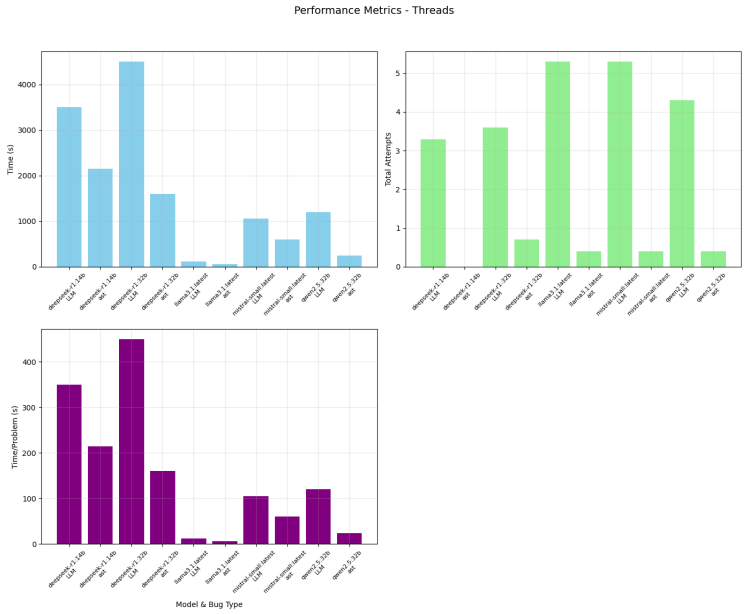


Figure 18: Model Comparison Threads

The multi-threading topic had very similar results to djikstra as both have high levels of nesting and subsequent complexity. However where it does differ is the diversity score, the multi-threading does not see nearly as diverse results as djikstra. Examing the output code this appears to be because each case of multi-threading defaulted to the same few fundamental examples of a deadlock while the djikstra topic had no clear bug to insert and so the results were much more varied.

5.5 Test Case Coverage

As mentioned in the design and implementation section, multiple test cases are generated for each problem. There is no method to actually figure out the code coverage of the test cases. It could be some future work to implement a tool to measure the coverage of the test cases and then use this to improve or create more test cases until full code coverage is acheived.

6 Conclusion

This project managed to achieve its goals of first creating a tool that is able to generate problems for students with customisable topic, bug type and partially customisable language. The project also investigated the performance of bug insertion using a LLM compared to a more targeted approach using Abstract Syntax Trees. A variety of metrics were implemented to compare the performance of the model with different parameters given. An extensive test suite was created to compare the program with different models, different bug insertion types, different topics and different hyper-parameters for the model. The results of these tests showed that the AST bug insertion method was faster and more reliable than the LLM approach. However, the AST bug insertion method cannot provide the same customisability to the end user as the LLM bug insertion method.

6.1 Potential Use Cases

This project was mainly targeted at novice programmers and professors who can use it to generate problems for their students. However, there is a lot of potential for this project to be used in more advanced settings. For example, for computer graphics, with the LLM bug insertion method and the correct prompt a user would be able to insert a visual bug into a graphics program. This would be useful for allowing graphics students to practice debugging in a more realistic setting. The existing implementation would not be able to check for visual bugs. This would mean there would have to be either a manual check of the users potential fix or some further work from this project to check if the user has fixed the bug. The extension could be very simple for a deterministic program where the correct

output is just one image or video, the correct output could be saved and compared against the users fix. Once there is more than one correct output the problem becomes much more complex.

6.2 Managing Debugging Difficulty

This project was targeted towards novice programmers and professors who can use it to generate problems for their students. This section will discuss the potential issues with managing the difficulty of the problems generated.

6.2.1 Comparing Bug Complexity

The complexity metrics are able to capture a good idea of the complexity of the code. The problem with it is that it is not able to capture the complexity of the bug inserted. If this tool was used to generate a set of problems for a class there is no way to be sure the students are all getting a program of the same level of difficulty. The tool will generally be reasonably consistent with the level of complexity of the bug inserted but there is no way to be sure and this makes it difficult to recommend this use case without some screening of the output.

6.2.2 Matching Difficulty to Learner

The debugging problems will become much more beneficial for the learner when the difficulty of the problem is correctly set for the user. The difficulty of the problem can be specified in the prompt, the difficulty of the bug can then also be specified in the prompt. Since

the code allows an override of the prompt this can be done easily. The difficulty here is to achieve the right balance of leaving the prompt open ended enough to achieve diversity but also outlining the difficulty of the problem clearly. This is not a complex task and with some trial and error can be done well, however, it would be an increase in complexity for using the tool.

6.3 Future Work

Some future work to expand upon this project could include two parts. The first to develop an interactive frontend that a user could then interact with whether that user be a student or a professor. There is also the potential to then collect user feedback of the tool as although the metrics give a good idea of the quality of the code they only attempt to recreate the opinions of a developer.

6.3.1 Interactive Frontend

An interactive frontend could be created to allow a user that would not be comfortable with a CLI to use the tool. This would be appropriate as the target audience is novice programmers or professors generating programs for novices programmers. This would also allow for implementing a debugger that could be used by the user to develop their skills using a debugger.

6.3.2 User Feedback

The user feedback could be collected in some future work based on the tool. This would likely require the previous mentioned point of an interactive frontend to be implemented. This would allow for the collection of user feedback on the generated problems. This could be used to improve the tool based on user feedback.

6.3.3 LLM Bug Insertion

Another potential piece of future work which may be of less importance is inspired by the poor consistency of the LLM bug insertion method. This could be improved upon by fine-tuning a model for bug insertion, this could provide a perfect balance of the two methods provided where it could have the adaptability and customisability of the LLM method but the reliability of the AST method.

In conclusion it is possible to generate useable problems to teach novice programmers how to debug using an LLM to generate the original code and then inserting bugs.

References

[1] Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. Proceedings of the Second International Workshop on Computing Education Research, 73–84. https://doi.org/10.1145/1151588.1151600

[2] Li, C., Chan, E., Denny, P., Luxton-Reilly, A., & Tempero, E. (2019). Towards a Framework for Teaching Debugging. Proceedings of the Twenty-First Australasian Computing Education Conference, 79–86. https://doi.org/10.1145/3286960.3286970

[3] O'Dell, D. H. (2017). The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. Queue, 15(1), 71–90. https://doi.org/10.1145/3055301.3068754

[4] Parkinson, M. M., Hermans, S., Gijbels, D., & Dinsmore, D. L. (2024). Exploring debugging processes and regulation strategies during collaborative coding tasks among elementary and secondary students. Computer Science Education, 0(0), 1–28. https://doi.org/10.1080/08993408.2024.2305026

[5] Whalley, J., Settle, A., & Luxton-Reilly, A. (2021). Analysis of a Process for Introductory Debugging. Proceedings of the 23rd Australasian Computing Education Conference, 11–20. https://doi.org/10.1145/3441636.3442300

[6] Whalley, J., Settle, A., & Luxton-Reilly, A. (2023). A Think-Aloud Study of Novice Debugging. ACM Transactions on Computing Education, 23(2), 1–38. https://doi.org/10.1145/3589004

[7] Denny, P., Leinonen, J., Prather, J., Luxton-Reilly, A., Amarouche, T., Becker, B. A., & Reeves, B. N. (2023). Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. https://doi.org/10.48550/ARXIV.2307.16364

[8] Denny, P., Leinonen, J., Prather, J., Luxton-Reilly, A., Amarouche, T., Becker, B. A., & Reeves, B. N. (2024). Prompt Problems: A New Programming Exercise for the Generative AI Era. Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, 296–302. https://doi.org/10.1145/3626252.3630909

[9] Nguyen, S., Babe, H. M., Zi, Y., Guha, A., Anderson, C. J., & Feldman, M. Q. (2024). How Beginning Programmers and Code LLMs (Mis)read Each Other. Proceedings of the CHI Conference on Human Factors in Computing Systems, 1–26. https://doi.org/10.1145/3613904.3642706

[10] David Janzen and Hossein Saiedian. 2008. Test-driven learning in early programming courses. SIGCSE Bull. 40, 1 (March 2008), 532–536. https://doi.org/10.1145/1352322.1352315

[11] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv. https://arxiv.org/abs/2303.11366

[12] Esposito, M., Janes, A., Kilamo, T., & Lenarduzzi, V. (2024). Early Career Developers' Perceptions of Code Understandability. A Study of Complexity Metrics. arXiv. https://arxiv.org/abs/2303.07722

[13] Pădurean, V.-A., Denny, P., & Singla, A. (2025). BugSpotter: Automated Generation of Code Debugging Exercises. Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1, 896–902. https://doi.org/10.1145/3641554.3701974

[14] Koutcheme, C., Dainese, N., Sarsa, S., Hellas, A., Leinonen, J., Ashraf, S., & Denny, P. (2025). Evaluating Language Models for Generating and Judging Programming Feedback. Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1, 624–630. https://doi.org/10.1145/3641554.3701791

[15] del Carpio Gutierrez, A., Denny, P., & Luxton-Reilly, A. (2024). Automating Personalized Parsons Problems with Customized Contexts and Concepts. Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, 688–694. https://doi.org/10.1145/3649217.3653568

[16] Brown, N. C. C., & Altadmri, A. (2014). Investigating novice programming mistakes: educator beliefs vs. student data. Proceedings of the Tenth Annual Conference on International Computing Education Research, 43–50. https://doi.org/10.1145/2632320.2632343

[17] Effenberger, T., & Pelánek, R. (2022). Code Quality Defects Across Introductory Programming Topics. Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1, 941–947. https://doi.org/10.1145/3478431.3499415