Debugging Problems

Patrick Farmer

Supervisor: Dr. Jonathan Dukes

April 7, 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 7, 2025

Contents

Acronyms

- LLM - Language Model

- AST - Abstract Syntax Tree

- AI - Artificial Intelligence

- API - Application Programming Interface

- GPT - Generative Pre-trained Transformer

1 Introduction

1.1 Context

In recent years there have been great strides made in the ability of LLMs to generate code. These LLMs are already being integrated into employee workflows in companies all over the world. This will place a much greater emphasis on the debugging process as the code generated by these LLMs much be inspected and fixed when bugs are produced. It has also been noted in many studies that debugging is an extremely important skill as it is the most difficult part of the programming process and is also the most often overlooked when teaching programming. More will be discussed on this in the literature review. This project will aim to create a tool that will help teach debugging to students who will be the future employees of these companies. LLMs will also play another significant role in this project as they play a key role in its functionality.

1.2 Aims

The aim of this tool is quite simple. To generate debugging problems for users to develop debugging skills. The code topic and bug type should both be customisable. An example of a problem generated by this tool with the topic of Fibbonaci sequence is shown below:

```python
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)
```

```python
        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()
```

It was also a goal to create two different methods for inserting the bug which can be evaluated and compared with each other. The first method involves prompting the LLM to generate code from the working code but that contains a bug. The second method modifies

7

the code's AST to introduce either a semantic or syntax bug to create common issues such as off-by-one errors by replacing or altering nodes.

It is a requirement of the project that there is a test case which a user can use to ensure that they have solved the bug. However an aim of the project is to create multiple test cases that can validate against different scenarios or bugs.

The creation of a frontend was not a goal of this project, the meaningful work of this project is all done 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 accessability of the tool and also provide additional features such as access to a debugger.

1.3 Methodology Outline

The project was implemented using Python. The program will first collect inputs from the user or if no inputs are provided select some default inputs. These inputs are then used to generate a working piece of code for a given topic. The output of this working code for a variable number of test cases is then collected. The program will then attempt to insert a bug using either the LLM or the AST bug insertion method depending on the input provided. The test cases will then be run and ensure that the code fails. If the code fails the buggy program is then saved and the program will move onto generating another problem.

1.4 Evaluation Metrics

It was decided early on that the evaluation and measuring of the project success would be done using desktop metrics and that there would not be a user study as it would not be required for accessing the functionality of the project, it could provide an idea of how useful the tool is for students. However desktop metrics were chosen that hope to emulate that

metric. The metrics that were decided on were code complexity, code diversity, attempt count and run time. The calculation of these metrics and the reasoning behind them will be discussed 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

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. This project primarily focuses on developing the debugging skill but there will be some domain knowledge that comes with it.

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 to student how to notice and solve the bugs they they will often cause in code they write themselves. As above it was mentioned by Jadud that students will often make minor syntax tweaks and rerun the code. This is because a lot of the bugs they they write into there code are syntax errors or smaller semantic errors such as infinite loops and off by one errors. This is why the tool that will be developed will true to insert these types of bugs.

The other primary use of the tool as mentioned 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 is the overlooking of 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 and the LLM will misunderstand 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.

It was mentioned above that test cases are important to ensure that the code has been fixed but it also serves the purpose of providing feedback to the user of the tool. 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 importance 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. The multiple different test cases providing different outputs is also why it is important that there is multiple different test cases that can test different parts of the code. This can be ensured by generating useful testcases through a prompt to the LLM.

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.

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.

2.2 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 and commonly 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 very quickly. For this reason a second method of interacting with local LLMs is used too.

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. The local LLMs are also more customisable with adjustable hyperparameters 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 but for final evaluation the same code was run using the OpenAI GPT API for comparison.

2.2.4 Prompt Engineering

When interacting with an LLM significant thought also has to be put into the prompts used to generate responses from the LLM. Prompt engineering is an important aspect of working with LLMs and can have a massive affect on the performance of the model. The most important aspect of prompt engineering is ensuring to get your point across clearly but there are also more subtle aspects to prompt engineering 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.

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.

The below diagram shows a pipeline diagram of the code. There is three main blocks to the code. The code generator which takes no input except for simple parameters such as model type, bug tpe, code topic and options for overriding default prompts. It will then produce functional code based on the topic mentioned by querying the LLM then parsing and saving the code to a newly created python file provided upon compilation it will work without errors. The output and input to this python script is also saved. The bug inserter then reads this same file. Depending on the method provided in the inputs it will either use LLM bug insertion or AST bug insertion. For LLM bug insertion it will follow the same process of quering the LLM, parsing and saving the response. The code will then be run to ensure that it fails. The AST method will follow the same process but insert bugs based on a number of pairs of dictionaries of nodes or values of nodes that should be replaced with other specific nodes or values. The buggy script is then saved over the original file (Indicated as separate on diagram for clarity). This will all be loops for however many problems the user wants to generate. The evaluation section will then read all of those files. It will first get the diversity metric by one by one comparing each of the files, this is inefficient but due to the LLM runtime being many orders of magnitude larger this is not significant. The complexity metric is then calculated using the radon library. This consists of both a cyclomatic complexity and a cognitive complexity metric that will be discussed in more detail later. The attempts to generate the original code and the attempts to insert the bug were also saved for each. Each of these metrics and the averages across the debugging problems are then saved to a CSV.
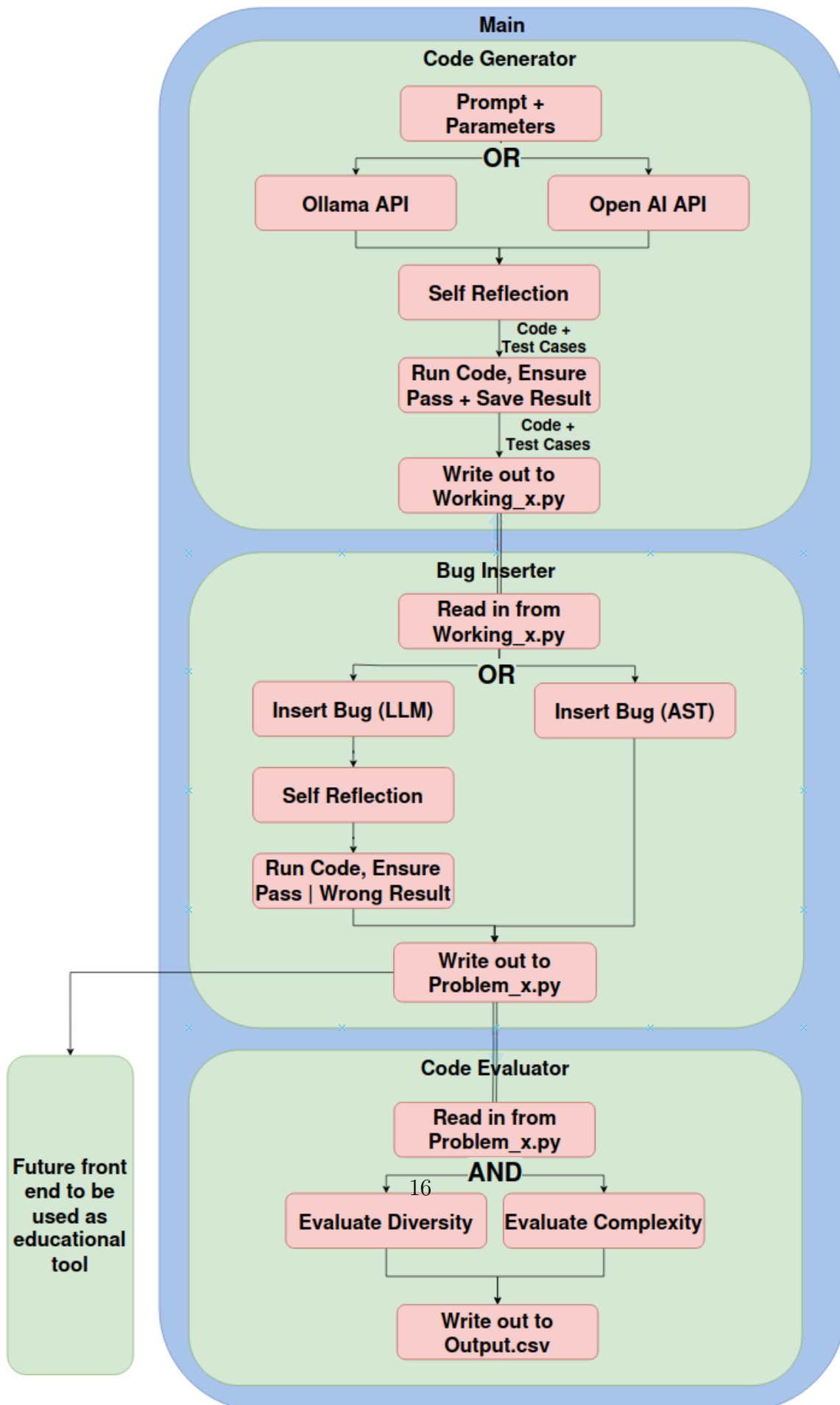
3.1 Design Overview

The bug inserter will then take this working code and insert a bug into it, this is done by either the use of 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 liable 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.

3.2 Tools

3.2.1 Ollama

The first of the two major tools that was used in this project is 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 but like this through some shell scripting language such as bash would not be the ideal solution. Fortunately the ollama API is much better suited to this task. The constructor for the class is called with some intialising hyperparameters and then a prompt can simple be passes 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.

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 is 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 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 was 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.

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. 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.

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   Environments

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. 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. The final 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.

4.2   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 chosen too, this is the amount of times the LLM had to be re-prompted because it produced an incorrect or unexpected output, this was mainly a good metric to compare against different models and also to compare the self relection 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.
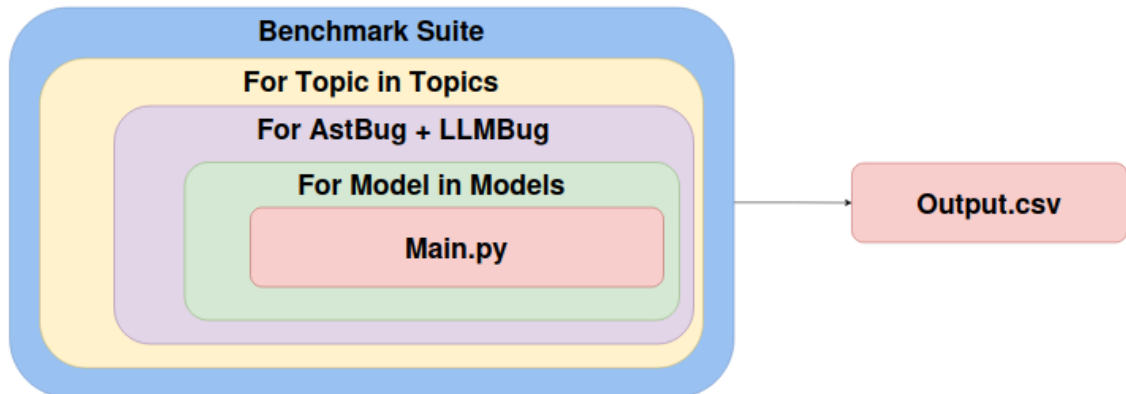

Figure 1: Benchmark Results

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 of course be much faster than the other tests as it is only one iteration of the script being 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 these, 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.

4.3 Metrics

4.3.1 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. The same is done for the cognitive complexity. 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. [1]

4.3.2 Code Diversity

The code diversity score is calculated individually for each file. 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.

4.3.3 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

---

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

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 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

4.3.4   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 hyperparameters 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 hyperparameters 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:
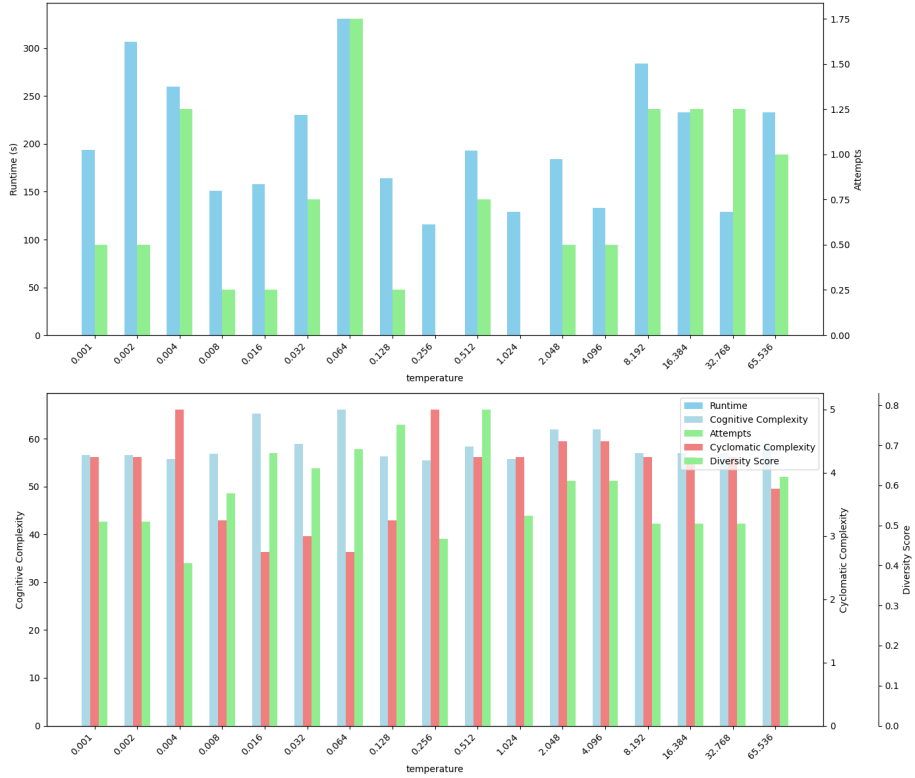

Figure 2: Complexity Comparison Threads

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.
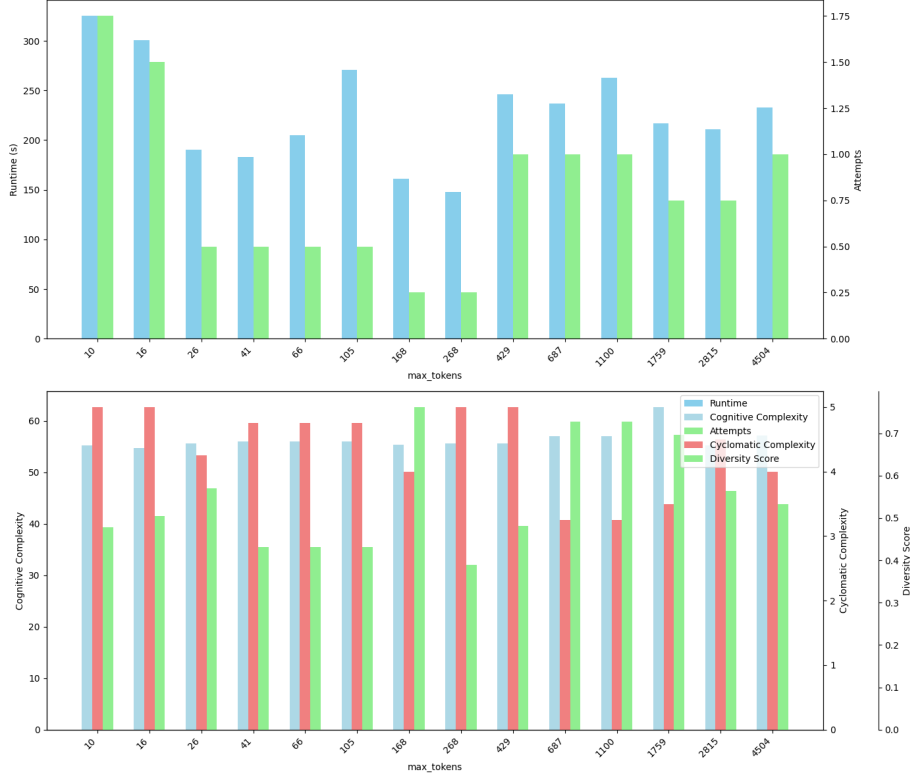

Figure 3: Complexity Comparison Threads

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

29

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.

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.

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
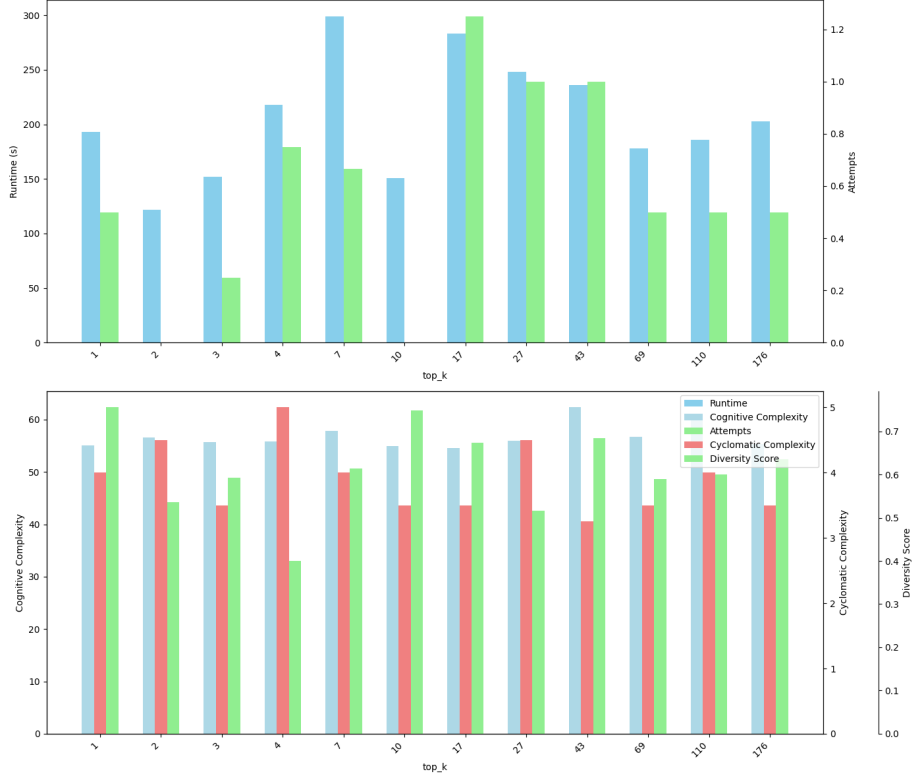
Figure 4: Complexity Comparison Threads

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
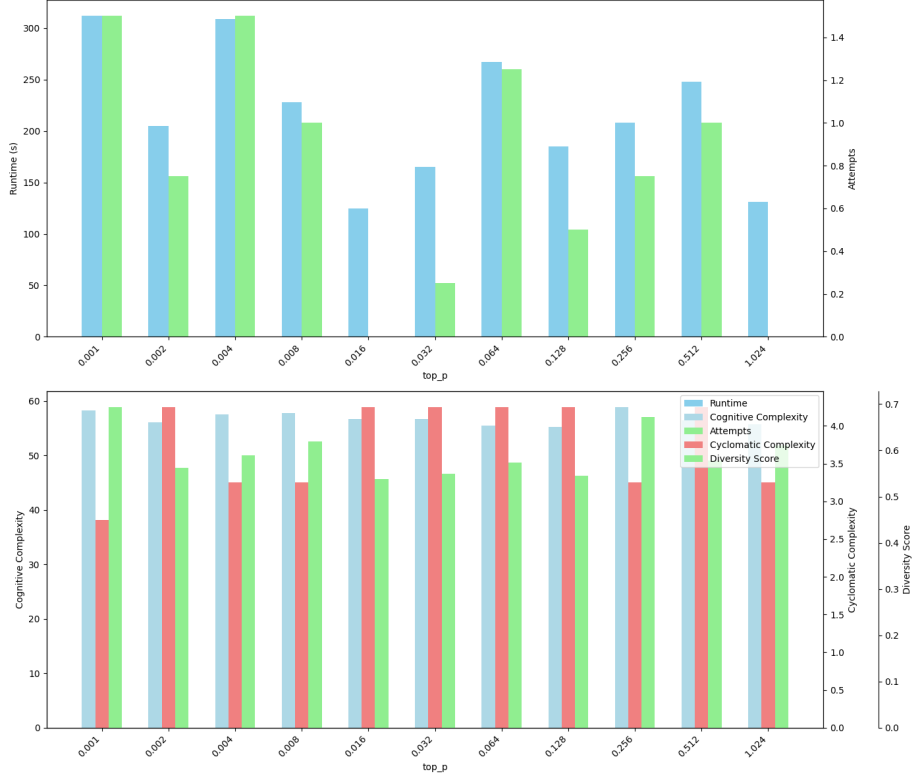
Figure 5: Complexity Comparison Threads

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.
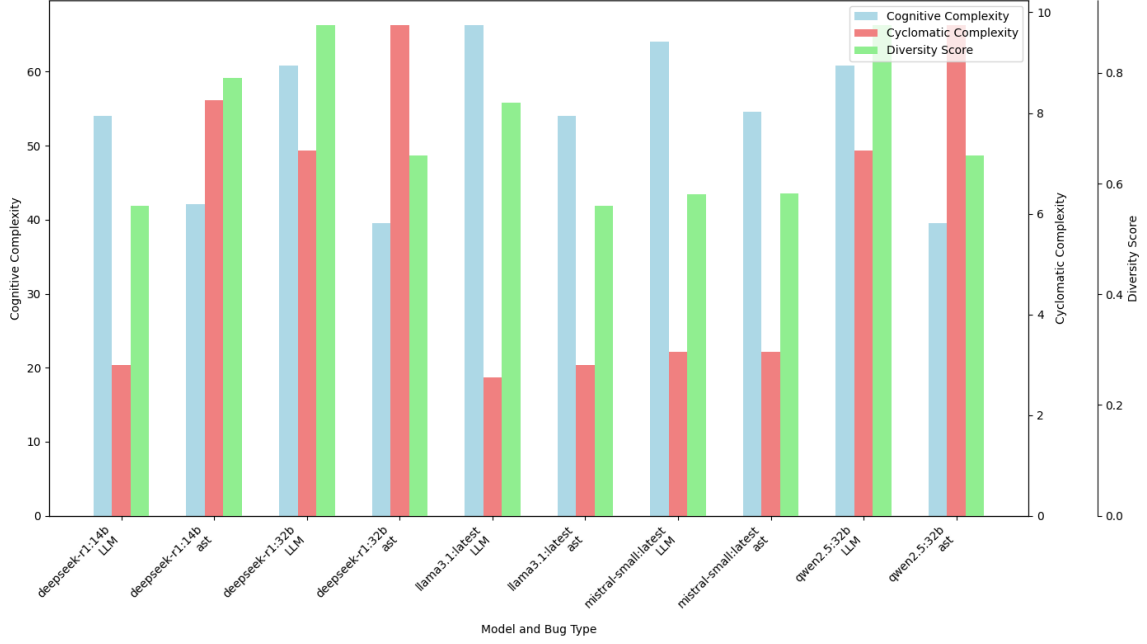
Across the board we can see a very large difference between the AST bug insertion and LLM bug insertion, for each model we can see that the runtime of the LLM bug insertion is at least two times larger, in some cases as much as four times larger. The same is seen for the attempts needed as the vast majority 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
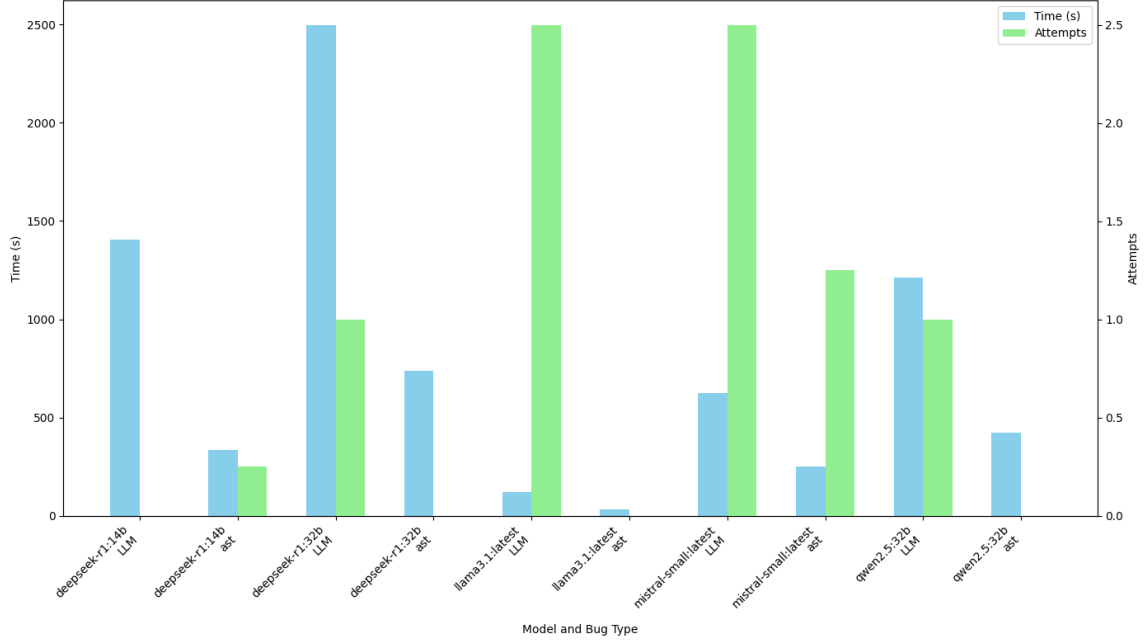
the below graph we can see the metrics for cognitive complexity, cyclomatic complexity and diversity for each model which are the best indicator of the quality of the code generated.

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 com-

plexity. 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.

The first topic that was tested was Arrays. This is a very simple topic which tests how the models and methods performed with a more constrained topic.
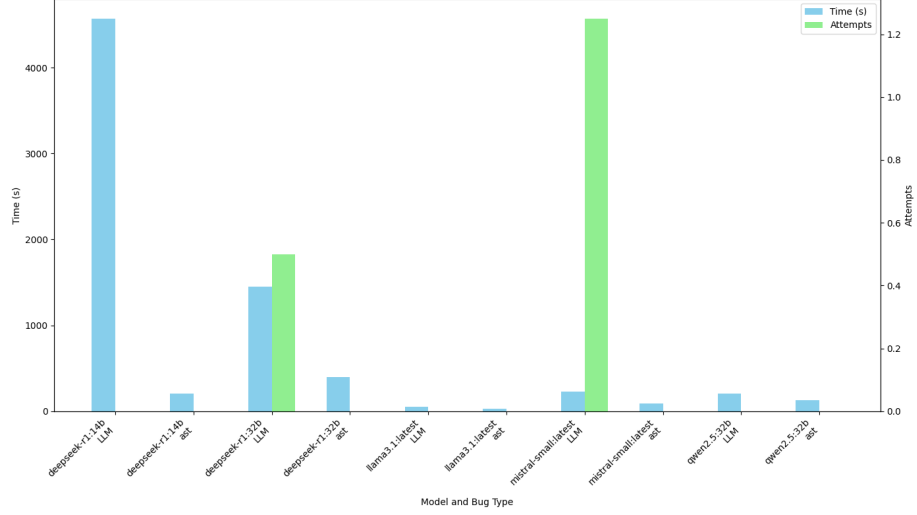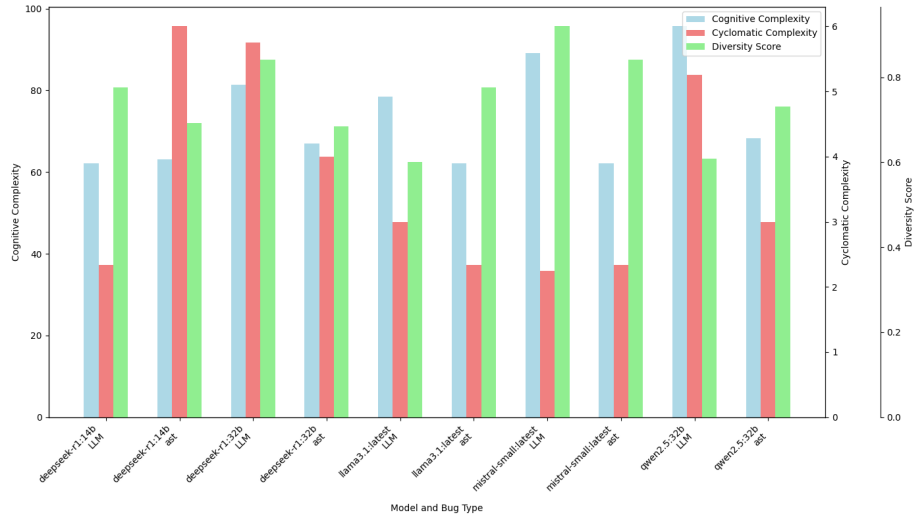

Figure 6: Model Comparison Arrays

Figure 7: Complexity 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.
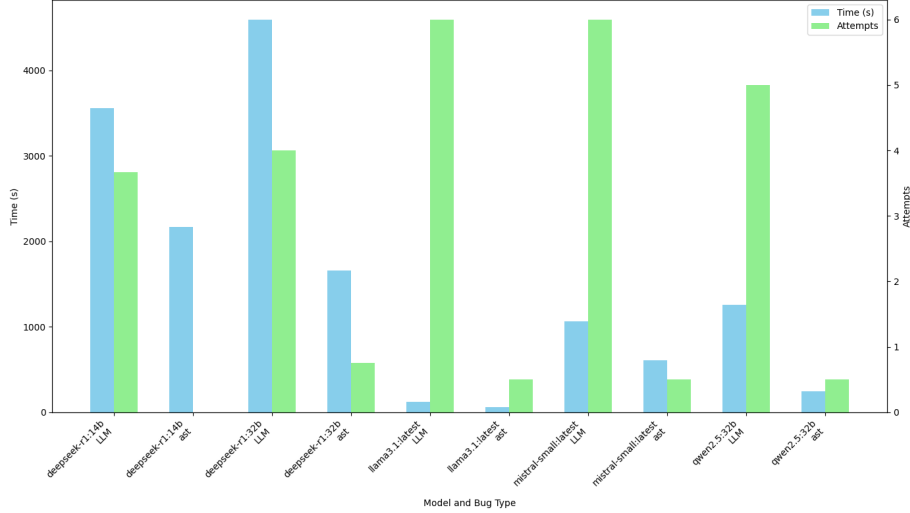

Figure 8: Model Comparison Djikstra

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.

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
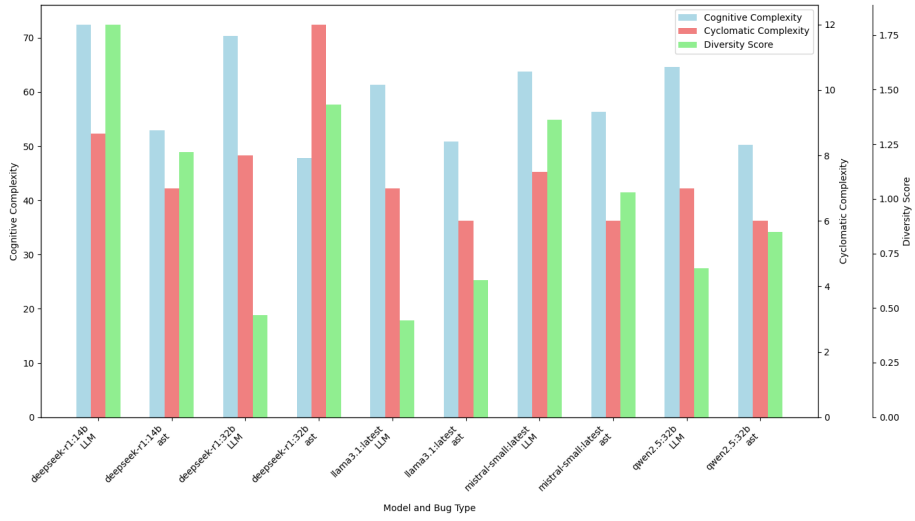
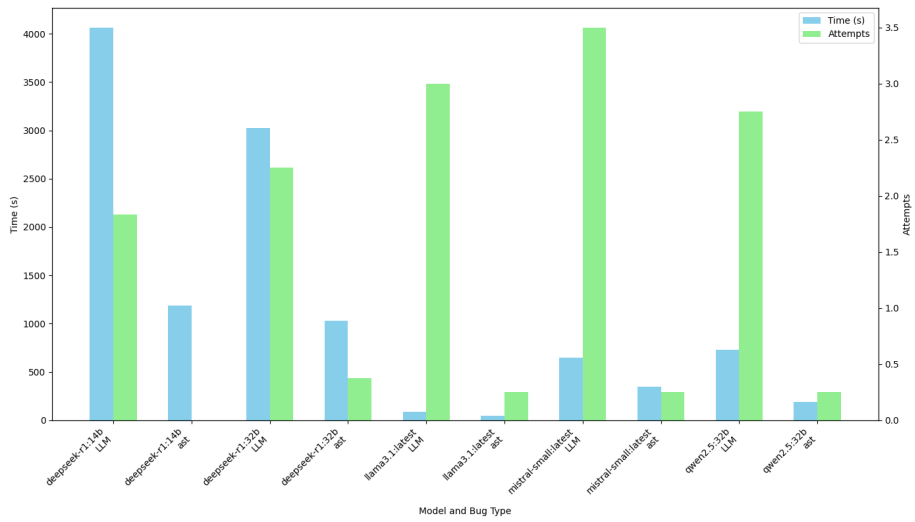Figure 9: Complexity Comparison Djikstra

Figure 10: Model Comparison Linked Lists

were very similar to the baseline and were not increased significantly.

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
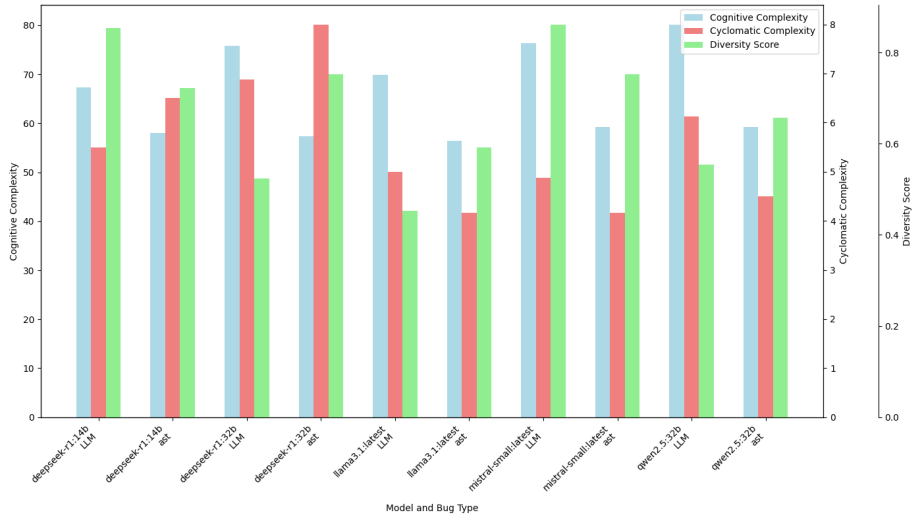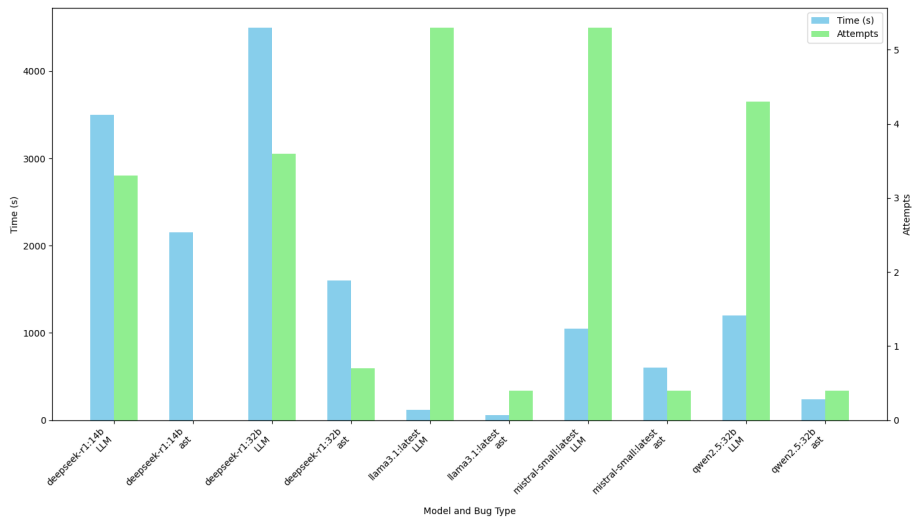
Figure 11: Complexity Comparison Linked Lists

Figure 12: Model Comparison Threads

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.
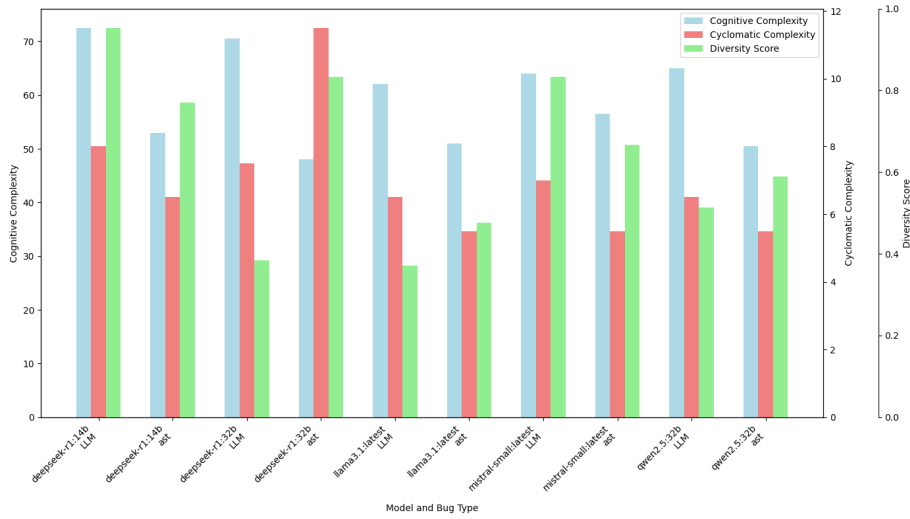
Figure 13: Complexity Comparison Threads

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 hyperparameters 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.

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. Another potential piece of future work which would definitely be less important 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 in a couple of methods.

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