Final Report

Level 04

Complex Code Generation Using Large Language Models

Group 29 – Loosers

Group Members

Index Number	Name
194176H	Vinoth V.
194061C	Hirushanan S.
194004G	Ajeevitha S.

Faculty of Information Technology
University of Moratuwa

2024

Complex Code Generation Using Llama Models

Group 29 – Loosers

Index Number	Name
194176H	Vinoth V.
194061C	Hirushanan S.
194004G	Ajeevitha S.

Supervised By: Dr. (Mrs.) KSD Fernando

Dissertation submitted to the Faculty of Information Technology, University of Moratuwa, Sri Lanka for the partial fulfillment of the requirements of the Honours Degree of Bachelor of Science in Information Technology.

Declaration

We declare that this thesis is our own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and the list of reference is given.

Name of the Student	Signature of the student
Vinoth V.	
Hirushanan S.	
Ajeevitha S.	

Date: ……..

Name of the Supervisor	Signature of the Supervisor
Dr. (Mrs.) KSD Fernando	

Date: …….

Acknowledgement

Our deepest appreciation goes to our supervisor, Dr. Subha Fernando, whose diligent guidance and invaluable advice were pivotal to the successful completion of our work. We sincerely thank the evaluators, Dr. T.M Thanthriwaththa and Ms. RGC Upeksha, for their insightful comments and feedback at each stage of the evaluation process.

We are profoundly grateful to the faculty of Information Technology, University of Moratuwa, for their expertise and knowledge imparted throughout our academic journey.

Lastly, we would like to express our heartfelt gratitude towards the free education system in Sri Lanka and all those who tirelessly contribute to its sustainability. Their efforts have been instrumental in shaping our academic and personal growth.

Abstract

The development of advanced language models such as LLaMA and GPT has significantly impacted various domains, including software engineering. These models have demonstrated potential in automating code generation, handling repetitive tasks, and creating complex algorithms. However, the generation of sophisticated Java code remains a complex challenge due to the inherent intricacies of such tasks. This project proposes a comprehensive solution to leverage the capabilities of LLaMA models for generating, validating, and optimizing complex Java code. The proposed system encompasses a multi-phase approach: problem decomposition, code generation, code integration, test case generation, and code optimization. By decomposing complex problems into manageable sub-problems, fine-tuning LLaMA models for code generation, and implementing robust validation and optimization mechanisms, this project aims to streamline the software engineering process. The ultimate goal is to create a cohesive system capable of handling the entire software development lifecycle, enhancing efficiency, accuracy, and robustness in the generation of complex Java applications.

Table of Contents

Chapter 1 – Introduction **9**

1.1 Introduction *9*

1.2 Background and Motivation *9*

1.3 Problem in Brief *10*

1.4 Aim and Objectives of the Research *10*

1.4.1 Aim 10

1.4.2 Objectives 11

1.5 Proposed Solution *11*

1.6 Resource Requirements *11*

1.7 Summary *12*

Chapter 2 – Literature Review **13**

2.1 Introduction *13*

2.2 Module 01: Complex Code Generation *13*

2.2.1 Introduction 13

2.2.2 Research Papers 13

2.2.3 Future Gaps 14

2.2.4 Conclusion 15

2.3 Test Case Generation *15*

2.3.1 Introduction 15

2.3.2 Research Papers 16

2.3.3 Conclusion 17

2.4 Code Optimization *17*

2.4.1 Introduction 17

2.4.2 Research Papers 18

2.4.3 Future works 19

2.5 Summary *19*

Chapter 3 – Technology Adapted **20**

3.1 Introduction *20*

3.2 Technology Adapted *20*

3.2.1 Llama 2 Model 20

3.2.2 Code Llama 20

3.2.3 GPT 3.5 20

3.2.4 Python programming language 20

3.2.5 Version Control System 21

3.2.6 Reinforcement Learning 21

3.3 Summary *21*

Chapter 4 – Our Approach **23**

4.1 Introduction *23*

4.2 Proposed System *23*

4.3 Users *23*

4.4 Module 1 - Complex Code Generation *24*
4.4.1 Input 24
4.4.2 Process 24
4.4.3 Output 24
4.4.4 Evaluation Techniques 25

4.5 Module 2 - Test Case Generation *25*
4.5.1 Input 25
4.5.2 Process 25
4.5.3 Output 26

4.6 Module 3 - Code Optimization *26*
4.6.1 Input 26
4.6.2 Process 26
4.6.3 Output 28

4.7 Summary *28*

Chapter 5 – Analysis and Design **29**

5.1 Introduction *29*

5.2 Modules of the System *29*
5.2.1 Complex Code Generation 29
5.2.2 Test Case Generation 32
5.2.3 Code Optimization 36

5.3 Summary *39*

Chapter 6 – Implementation **40**

6.1 Introduction *40*

6.2 Modules of the System *40*
6.2.1 Complex Code Generation 40
6.2.2 Test Case Generation 47
6.2.3 Code Optimization 50

6.3 Summary *54*

Chapter 7 – Discussion **55**

7.1 Introduction *55*

7.2 Discussion *55*

7.3 Summary *56*

References **57**

Appendixes **63**

Appendix A – Individual contribution *63*

194176H – Vinoth V. 63

194176H – Ajeevitha S. 63

194061C – Hirushanan S. 64

Chapter 1 – Introduction

1.1 Introduction

In the fast-paced realm of modern software development, the demand for efficient and creative algorithm design has become paramount. This project delves into an innovative paradigm of Java code generation, aiming to transcend the boundaries of traditional methods. The primary objective is to transition from manually crafting intricate algorithms to harnessing the capabilities of pre-trained generative models, specifically the Llama model fine-tuned for Java code. By doing so, the project aspires to cater to the escalating need for advanced algorithmic solutions, thereby automating the generation of complex and logically intricate Java code.

In essence, the project positions itself as a pioneering force in revolutionizing the landscape of algorithm design by incorporating cutting-edge generative models into the process of Java code creation.

1.2 Background and Motivation

As software development undergoes rapid evolution, the demand for sophisticated algorithms has witnessed an unprecedented surge. Despite this surge, the conventional approach to crafting complex algorithms remains a time-consuming and arduous task. This project is grounded in recent breakthroughs in natural language processing and generative models, with a specific focus on the Llama model, which has been adeptly fine-tuned for Java code.

The motivation behind this endeavor lies in addressing the inherent challenges faced by developers in creating intricate code structures from scratch. By leveraging the power of generative models, the project aims to provide a foundational tool that streamlines the process of creating complex Java code. This, in turn, is anticipated to enhance developer productivity and foster innovation in algorithmic design. The synthesis of advanced language processing and algorithmic creativity is poised to reshape the way developers approach complex problem-solving in the ever-evolving landscape of software engineering.

1.3 Problem in Brief

Current language models face significant challenges when generating complex Java code due to the non-deterministic nature of such tasks and the partially observable environments in which they operate. Complexity in code generation is not determined by the number of sentences a model can produce, but rather by the ability of the model to handle intricate tasks and environments. For instance, models like LLaMA struggle with code generation in non-deterministic environments where outcomes are not always predictable. In such settings, partial observability means the model lacks complete descriptions of the entire problem, complicating the generation process.

When addressing complex problems, especially those requiring sophisticated models like predictive algorithms, the limitations of LLaMA become evident.

Complexity arises not from the number of sentences but from the need to formulate an answer given an objective function. For example, while LlaMA can write code to train a neural network model, it requires precise contextual input to function correctly. If the context is unclear, the model will fail.

The true complexity of a problem lies not in code generation but in the problem itself. It involves the volume of data, clarity of instructions, and the level of abstraction needed. When a problem is presented in an abstract form, the model struggles to break it down into smaller, understandable steps. This is where LLaMA often fails—when the problem requires composite actions that need detailed instructions.

1.4 Aim and Objectives of the Research

Define the overarching goal (aim) and the specific objectives that contribute to achieving that goal. In this project:

1.4.1 Aim

The primary aim of this research is to develop a system for generating complex Java algorithms using pre-trained generative models.

1.4.2 Objectives

1. To implement a module for complex Java code generation based on the Llama model fine-tuned on Java code.
2. To design a test case generation and validation module using a fine-tuned Llama model to ensure the correctness and robustness of the generated Java code.
3. To develop a function optimization module that ranks generated code based on efficiency and style metrics, suggesting potential improvements.
4. To carefully select and evaluate Llama models for optimal performance in Java code generation tasks.
5. To experiment with different fine-tuning strategies for improved results in each module.
6. To ensure the quality and diversity of datasets used for training and fine-tuning.
7. To establish appropriate metrics for evaluating the quality of generated code, test cases, and code optimization/comments.
8. To integrate the modules into a cohesive system, providing a comprehensive solution for automated Java code generation.

1.5 Proposed Solution

The proposed solution involves the development of three interconnected modules. The first module focuses on leveraging a pre-trained Llama model for Java code generation, transforming simple code snippets into complex algorithms. The second module addresses the validation and correctness of the generated code by utilizing the same generative model for test case generation. The third module introduces a function optimization aspect, ranking the generated code based on efficiency and style metrics. By carefully selecting and evaluating Llama models, experimenting with fine-tuning strategies, and ensuring dataset quality, the proposed solution aims to provide a comprehensive, efficient, and reliable system for automated Java code generation.

1.6 Resource Requirements

<u>Hardware:</u>

High-performance computing resources (GPUs or TPUs)

Sufficient storage capacity for datasets and models

Software:

Colab Notebook

Llama 2 Model (7B)

CodeLlama

GPT 3.5

Deep learning libraries (TensorFlow, PyTorch)

Natural language processing libraries

Datasets:

Datasets of Complex problem with its subproblems

Datasets of problems with their codes (Code snippets)

Datasets of Java code with test cases

Datasets of Java code with varying levels of quality and optimization

1.7 Summary

In summary, this research project aims to address the challenge of automating the generation of complex Java algorithms. By leveraging the capabilities of pre-trained generative models, specifically the Llama model fine-tuned on Java code, the project proposes a multi-module solution. These modules encompass complex Java code generation, test case validation, and function optimization. The project outlines specific objectives, emphasizing the importance of model selection, fine-tuning strategies, dataset quality, and comprehensive evaluation metrics. The integration of these modules into a cohesive system is expected to contribute to the advancement of automated Java code generation, enhancing efficiency and creativity in algorithm design within the software development landscape.

Chapter 2 – Literature Review

2.1 Introduction

Overall project ideas together with the background, aim, objectives and proposed solution were described in detail in the previous chapter. To accomplish the implementation of they decided solution successfully, familiarity of the other existing products and research work related to it is important.

This chapter reviews the related work to our research area. The related work is illustrated under three research areas, the project's main modules.

- Complex Code Generation
- Test Case Generation
- Code Optimization

2.2 Module 01: Complex Code Generation

2.2.1 Introduction

In recent years, the advent of large language models (LLMs) has transformed various domains, including natural language processing, image generation, and most notably, code generation. Traditional code generation techniques typically rely on supervised learning, which initially treats code as natural language and gradually incorporates more code-specific features, such as abstract syntax trees and API calls. However, these methods often face challenges in data efficiency and scalability. The concept of self-planning in code generation has emerged as a promising approach to address these issues by teaching LLMs to plan their actions before generating code. This review explores the current state of the art in complex Java code generation using LLM models, identifying key research papers and highlighting future gaps that deserve further investigation.

2.2.2 Research Papers

Code Llama: Llama 2 learns to code (Touvron et al., 2023): This foundational paper introduces Code Llama, a family of Llama models specializing in code generation. They trained Code Llama on massive code datasets and fine-tuned them for several languages, including Java. The results showcase impressive performance on code completion, infilling, and natural language instruction-based generation. [1]

A comprehensive review of State-of-The-Art methods for Java code generation from Natural Language Text (Yang et al., 2023): This review compares Code Llama with other code generation models like StructCoder and REDCODER. They conclude that Code Llama achieves superior performance on benchmark datasets, demonstrating its effectiveness in handling complex Java tasks. [2]

On the Use of Abstract Syntax Trees in Deep Learning for Program Synthesis (Almanza et al., 2023): This paper explores incorporating Abstract Syntax Trees (ASTs) into code generation models. They demonstrate that integrating ASTs with the transformer architecture used in Code Llama improves the model's ability to generate syntactically correct and semantically consistent Java code.

Language Models are Few-Shot Learners - Brown et al. (2020) introduced the GPT-3 model, demonstrating its ability to perform various tasks, including code generation, with minimal examples (few-shot learning). The model's performance in code generation tasks highlighted the potential of LLMs in this domain but also underscored the need for more structured planning in complex tasks.

Program Synthesis with Large Language Models - Austin et al. (2021) explored the capabilities of LLMs in program synthesis, emphasizing the importance of understanding and planning the steps required to generate accurate and functional code. Their work laid the groundwork for integrating planning strategies into LLMs.

CodeT: Code Generation with Generated **Tests** - Chen et al. (2023) proposed an approach where LLMs generate both code and corresponding tests, improving the robustness and correctness of the generated code. This research highlighted the benefits of a more structured approach to code generation, aligning with the principles of self-planning.

2.2.3 Future Gaps

1) Handling Intricate Program Logic: Current AI models excel at syntax and basic logic but struggle with generating code involving complex decision-making and control flow. Future research needs to explore incorporating advanced reasoning and logic verification techniques into code generation models to address these challenges.

2) Real-World Application Feasibility: Scaling AI models to large-scale, real-world projects requires addressing integration with existing software development pipelines and ensuring compatibility with current codebases. Research focusing on code quality, safety, and security in generated code is crucial for practical implementation.

3) Explainability and Debugging: Understanding how AI models make decisions and generate code is important for debugging and building trust in AI-generated code. Research on interpretable AI techniques can significantly improve model explainability and foster greater acceptance among developers.

2.2.4 Conclusion

Llama models show significant promise in complex Java code generation, pushing the boundaries of AI-powered software development. Addressing the identified future gaps through further research will unlock the full potential of these models and pave the way for a future where AI plays a pivotal role in writing reliable and efficient Java code.

2.3 Test Case Generation

2.3.1 Introduction

Automated test case generation (ATCG) is a crucial aspect of software engineering aimed at improving software reliability and efficiency by automating the creation of test cases. Traditionally, test-driven development (TDD) methodologies advocate writing tests before implementing code, ensuring that software functions correctly and meets specified requirements. However, manual test case generation is labor-intensive and time-consuming, prompting the exploration of automated approaches.

Recent advancements in natural language processing (NLP) and machine learning, particularly with large language models (LLMs), have opened new avenues for automated test case generation. LLMs like GPT-3.5 have demonstrated the ability to understand and generate human-like text, making them promising candidates for generating test cases directly from natural language descriptions of software functionality.

Test-Driven Development (TDD) stands as a pivotal agile methodology in software development, advocating for the creation of test cases prior to code implementation. This iterative process involves iteratively defining tests based on requirements, implementing code to fulfill these tests, and then refining the codebase to improve its quality. TDD has been shown to significantly enhance code reliability and facilitate better requirement handling compared to traditional methods.

Research indicates that systems developed with TDD methods can experience a reduction in defects by approximately 50% when contrasted with those using ad-hoc testing approaches. However, a primary challenge of TDD lies in the substantial time and effort required initially to craft comprehensive and effective test cases—a crucial but resource-intensive endeavor.

In response to this challenge, various automated test case generation approaches have emerged. These methodologies leverage diverse techniques such as random testing, evolutionary algorithms, and more recently, deep learning to automate the generation of test cases. Despite their advancements, many existing methods necessitate source code as input (Code → Test), rendering them less compatible with TDD's ideal workflow, where tests precede code.

2.3.2 Research Papers

Here's a concise description of each research paper mentioned, focusing on their contributions to automated test case generation:

Random Testing (Pacheco and Ernst, 2007):

Description: Pacheco and Ernst introduced Randoop, a seminal tool that utilizes feedback-directed random testing. Randoop dynamically explores software executions using neural network architectures, generating test cases that aim to cover diverse paths within the codebase.

Contribution: This approach pioneered the use of neural networks in automated testing, providing a basis for generating test cases without prior assumptions about program behavior.

Evolutionary Algorithms (Fraser and Arcuri, 2011):

Description: Fraser and Arcuri developed EvoSuite, leveraging evolutionary algorithms for automated test case generation. EvoSuite employs genetic algorithms to evolve test cases iteratively, optimizing for code coverage and ensuring comprehensive software validation.

Contribution: EvoSuite introduced a systematic approach to evolve test suites automatically, addressing the challenge of generating tests that cover critical paths in complex software systems.

Deep Learning (Tufano et al., 2020):

Description: Tufano et al. explored deep learning techniques for automated test case generation directly from source code. Their approach involved training deep neural networks to synthesize test cases that verify the functionality of software components, enhancing testing efficacy and efficiency.

Contribution: This research extended automated test generation capabilities to leverage deep learning models, enabling the creation of tests that validate software functionality across various scenarios.

2.3.3 Conclusion

In conclusion, automated test case generation using LLMs represents a transformative advancement in software engineering practices, offering potential benefits in terms of efficiency, reliability, and scalability. The reviewed literature underscores the evolution and challenges in harnessing neural networks, evolutionary algorithms, and deep learning for automated testing. Leveraging LLMs for generating test cases directly from natural language descriptions addresses fundamental limitations of conventional approaches, making it a pivotal tool in modern software development workflows. Future research opportunities in this domain could focus on refining LLM architectures, advancing evaluation methodologies, and seamlessly integrating automated test case generation with agile methodologies like TDD.

2.4 Code Optimization

2.4.1 Introduction

We will also explore how performance relates to code metrics like execution time and cyclomatic complexity, which measures how complex the code is. In the past, these metrics were often calculated manually, but now many programming languages use large language models (LLMs) to help with code optimization. We'll start by discussing well-known research on different ways to optimize Java code, how effective these methods are, and common performance issues. We will also explore how performance relates to code metrics like time complexity and cyclomatic complexity, which measures how complex the code is. In the past, these metrics were often calculated manually, but now many programming languages use large language models (LLMs) to help with code optimization. One such LLM is Code Llama, which has some limitations. Few studies have focused on optimizing Java code generated by Code

Llama, as there are many ways to achieve the same result, each with different metric values. These values need to be prioritized in the optimization model.
We'll look for existing datasets that help develop recommendation engines or train models for optimizing Java code. Since there are no comprehensive datasets with evaluation metrics for Java code, we decided to gather Java code from the internet, evaluate its complexity measures, and create a new dataset. By incorporating human feedback, we aim to improve the model's accuracy, ultimately achieving better Java code optimization.

2.4.2 Research Papers

LangProp: A Code Optimization Framework Using Large Language Models Applied to Driving (Ishida et al., 2024): This paper proposes LangProp, a framework for iteratively optimizing code generated by large language models (LLMs) in both supervised and reinforcement learning settings. The framework evaluates code performance on a dataset of input-output pairs, catches exceptions, and feeds results back to the LLM in the training loop for iterative improvement. The research demonstrates the application of LangProp in driving scenarios, providing valuable insights into the potential of LLMs for code optimization.

Romero Rosas, M., Barakhshan, B., & Eigenmann, R. (2024). "Should AI Optimize Your Code? A Comparative Study of Current Large Language Models Versus Classical Optimizing Compilers." arXiv:2406.12146. This paper presents a comparative analysis between state-of-the-art Large Language Models, such as GPT-4.0 and CodeLlama-70B, and traditional optimizing compilers like CETUS, PLUTO, *and* ROSE. It assesses their abilities and limitations in optimizing code for maximum efficiency, finding that while LLMs show potential to outperform traditional compilers, they often generate incorrect code on larger sizes, highlighting the need for automated verification methods.

CODECOMPLEX: A Time-Complexity Dataset for Multi-Language Source Codes (Maddern et al., 2024): This foundational paper introduces CODECOMPLEX, a comprehensive dataset designed to evaluate the time complexity of source codes across multiple programming *Code Llama: Llama 2 learns to code (Touvron et al., 2023):* The Code Llama family of Llama models, which specialises in code creation, is introduced in this seminal study. Large code datasets were used to train Code Llama, and they optimised it for a number of languages,

including Java. The outcomes demonstrate remarkable achievement in terms of code completion, infilling, and creation based on natural language teaching. The complete document is available at.

2.4.3 Future works

1) Handling logic for Using RL based Algorithms for code optimization: Define Software Metrics like Computational cyclometric Complexity, Time Complexity which can vary based on factors like input size, hardware, and implementation details. integrate these metrics into the training and feedback loop of code generation models. Run the generated code through a profiling tool or instrumentation to measure its computational and time complexity metrics. Compare with Optimized Code. then Calculate Errors or Scores. Provide this score or error as feedback to the code generation model. The model can then learn to minimize these scores during future iterations.

2)Dynamic Code Optimization: two prominent applications of reinforcement learning (RL) emerge as pivotal strategies. Firstly, online learning integrates RL agents that continually adapt to evolving workload conditions and performance demands. These agents learn from real-time feedback, adjusting code execution strategies to optimize efficiency and responsiveness dynamically. Secondly, adaptive compilation techniques leverage RL to guide the compilation process based on runtime behavior and observed performance metrics. By dynamically optimizing code generation and execution paths, adaptive compilation strategies enhance overall system efficiency and adaptability, making them invaluable in environments where performance optimization is crucial.

2.5 Summary

This chapter summarizes the recent approaches used to solve the identified problem. Through this chapter the recent approaches and the pitfalls of them and the gaps that exists in the current research are identified. The next chapter will describe the technologies that we hope to use in implementing the solution.

Chapter 3 – Technology Adapted

3.1 Introduction

The evolution of technology has significantly impacted the field of software development, offering new avenues for innovation and efficiency. This section explores the technologies adapted for the three key modules of the project: Complex Java Code Generation, Test Case Generation and Validation, and Code Optimization.

3.2 Technology Adapted

3.2.1 Llama 2 Model

Llama 2, a collection of pretrained and fine-tuned large language models (LLMs) ranging in scale from 7 billion to 70 billion parameters. We have focused on using Llama 2 model with 7 billion parameters that will be easy to run and fine-tune with our local machine GPU.

3.2.2 Code Llama

Code Llama is a code-specialized version of Llama 2 that was created by further training Llama 2 on its code-specific datasets, sampling more data from that same dataset for longer. Essentially, Code Llama features enhanced coding capabilities. It can generate code and natural language about code, from both code and natural language prompts.

3.2.3 GPT 3.5

ChatGPT is primarily used for natural language understanding and generation, making it valuable for tasks like content creation, chatbot development, language translation, and more.

3.2.4 Python programming language

Python is a high level, dynamically typed, powerful programming language which consists of a huge set of libraries and tools needed for handling large amounts of data sets. It is an object oriented, interpreted, flexible language which provides data interaction, exploration, and reusability. Python usually comes with extensive standard library, numerical processing, and web connectivity. Some of the main libraries needed for the implementation are as follows.

I. Scikit-learn: Python machine learning library.

An open-source library that implements a range of machine-learning, pre-processing, cross-validation, and visualization algorithms. It is efficient and simple for data analysis. It includes various classification, regression, and clustering algorithms such as support vector machine, k-means, random forests etc.

II. NumPy and Pandas

These two libraries interoperate with each other. NumPy adds support for large, multidimensional arrays and matrices along with high-level mathematical functions whereas Pandas depends on NumPy. Both makes the importing and analysis of data easier.

III. Natural Language Toolkit (NLTK)

It contains text processing libraries for tokenization, parsing, classification, and semantic reasoning. It is an open-source library for python programming language. It contains text processing libraries for tokenization, parsing, classification, and semantic reasoning. It is an open-source library for python programming language. This is important for the task of review analysis using twitter in the project.

3.2.5 Version Control System

Git-Lab is used as the source code management system. It tracks changes in the development process and provide a collaborative work environment to the developers.

3.2.6 Reinforcement Learning

Using reinforcement learning with software metrics as feedback scores, it's possible to create a smart system that continuously gets better at optimizing code. This approach not only makes the code run faster and use less memory, but also keeps it easy to read and maintain, resulting in higher quality software development.

3.3 Summary

In summary, the technology adapted for the project revolves around the integration of generative models, machine learning, and advanced testing methodologies. The adoption of

technologies like Llama signifies a paradigm shift in how complex Java code is generated, emphasizing the synergy between human creativity and machine intelligence. The incorporation of these technologies not only addresses the challenges in algorithmic design but also opens avenues for continuous improvement and innovation in software development practices. The subsequent chapters delve into the details of each module, showcasing how these technologies are harnessed to achieve the project's overarching goals.

Chapter 4 – Our Approach

4.1 Introduction

In this chapter, we outline our approach to address the challenges posed by complex Java code generation. Our methodology involves breaking down the process into distinct modules, each serving a specific purpose in the overall code generation and refinement pipeline. The key modules include Complex Java Code Generation, Test Case Generation and Validation, and Function Optimization.

4.2 Proposed System

The proposed system integrates advanced language models, specifically fine-tuned Llama models, to enhance the efficiency and reliability of Java code generation. Each module plays a crucial role in the overall process, contributing to the generation of complex yet optimized Java code.

4.3 Users

The users of our system include developers and programmers seeking to expedite the code development process while ensuring the generated code is correct, robust, and optimized.

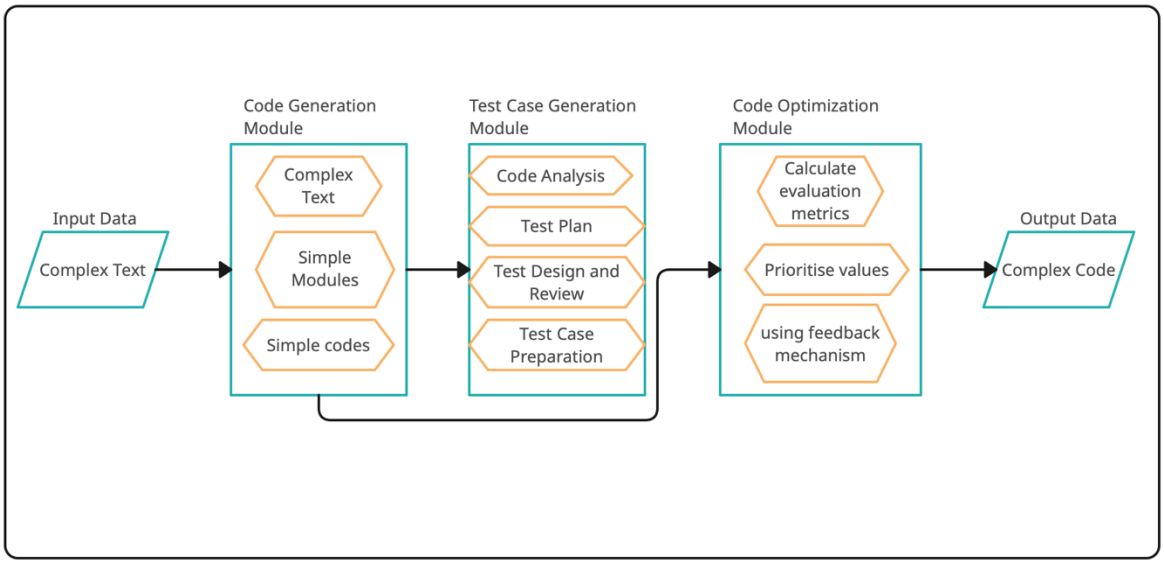


Figure 1: Overall Approach and Modules of the Project

4.4 Module 1 - Complex Code Generation

4.4.1 Input

The core idea behind this module is to take a very complex text-based code and decompose it into smaller, more manageable modules. This involves training a small Llama language model to identify and extract these smaller tasks from the complex code.

Input of this module is very complex text code.

4.4.2 Process

4.4.2.1 Process Steps

1) Planning Phase: The self-planning approach involves generating concise solution steps from the user's intent before code implementation. This planning phase helps the model understand complex intents and reduces the difficulty of problem-solving. The steps include:

1. **Solution Steps Generation**: The LLM generates solution steps from the given intent using few-shot prompting techniques or finetuned LLM models. This phase is crucial for breaking down complex tasks into simpler sub-tasks.

2) Implementation Phase: Once the planning phase is complete, the implementation phase begins. This involves generating code step-by-step, guided by the solution steps from the planning phase. The steps include:

1. **Step-by-Step Code Generation**: The LLM generates the code for each solution step, ensuring that each part of the complex code is handled individually and correctly.
2. **Integration Algorithm/model**: An algorithm or used prompting techniques integrates the generated code modules, ensuring that the overall logic and functionality of the original complex code are preserved.

4.4.3 Output

The output is integrated complex code generated from the smaller modules. This code retains the functionality and logic of the original complex input but is produced in a more modular and manageable form.

4.4.4 Evaluation Techniques

Pass@1: This is a metric that evaluates the percentage of correctly generated code snippets on the first attempt.

Human Evaluation

1. **Correctness**: Human evaluators assess whether the generated code correctly implements the intended functionality.
2. **Readability**: The readability of the generated code is judged, considering aspects like code structure, naming conventions, and comments.
3. **Robustness**: Evaluators check if the code is robust, meaning it can handle edge cases and unexpected inputs without failing.

4.5 Module 2 - Test Case Generation

4.5.1 Input

This module aims to generate test cases from natural language descriptions of software methods. Using a fine-tuned language model, it parses and translates method descriptions into executable test cases, ensuring alignment with functional requirements and comprehensive code coverage.

4.5.2 Process

Planning Phase: This phase involves generating concise solution steps from the method descriptions before generating test cases. This planning helps the model understand the method's functionality and reduces the complexity of generating relevant test cases.

Solution Steps Generation: The LLM generates solution steps from the method descriptions using few-shot prompting techniques or fine-tuned LLM models. This phase is crucial for breaking down the method's functionality into simpler sub-tasks, aiding in the creation of effective test cases.

Implementation Phase: Once the planning phase is complete, the implementation phase begins. This involves generating test cases step-by-step, guided by the solution steps from the planning phase. The steps include:

Step-by-Step Test Case Generation: The LLM generates the test cases for each solution step, ensuring that each part of the method's functionality is tested individually and correctly.
Integration and Verification: This step involves comparing the generated test cases with the previously generated code. The algorithm verifies that the test cases align with the method descriptions and thoroughly test the original code's logic and functionality. This ensures consistency and accuracy in the generated test cases.

4.5.3 Output

The output of this model is a comprehensive suite of test cases specifically tailored to evaluate the generated Java code.

4.6 Module 3 - Code Optimization

4.6.1 Input

The core idea is that there are matrices to evaluate computational time and the number of steps required to execute the code. Then, once the code generation or GPT models run, you should integrate feedback mechanisms to train it that are based on those scores.
The input to this module is the complex Java code that has been generated by the Code-LLMs model. It is generated from the previous module.

4.6.2 Process

4.6.2.1 Process Steps

Evaluation matrices and algorithms are essential tools in the process of optimizing and assessing the performance and complexity of generated code. These matrices serve to quantify various aspects of code quality and efficiency. Here's how they are utilized:

Evaluation Matrices:

Computational Time Matrix

The Computational Time Matrix focuses on measuring the efficiency of code execution, using the following metrics:

1. **Execution Time:** This metric measures the actual time taken by the code to complete its execution, typically in milliseconds or seconds.
2. **Time Complexity (Big O Notation):** Time complexity indicates how the runtime of the code scales with respect to the size of the input. It provides a theoretical estimate of execution time under different input sizes.
3. **Number of Iterations or Recursive Calls:** This metric counts how many times loops are executed or recursive functions are called during the execution of the code. It gives insight into the iterative or recursive behavior of the code.

Computational Steps Matrix

The Computational Steps Matrix evaluates the structural characteristics of the code, focusing on its complexity and readability:

1. **Number of Statements:** This metric counts the total number of individual statements or operations in the code. It provides a measure of the code's overall length and complexity.
2. **Number of Loops and Conditions:** This metric quantifies the number of loop constructs (like for, while) and conditional statements (if, else) present in the code. It assesses the branching and looping behavior of the code.
3. **Nested Structure Analysis:** Nested structure analysis evaluates how deeply structures such as loops and conditions are nested within each other. It indicates the level of complexity in control flow within the code.

Model Integration

To integrate these evaluation matrices into the code optimization process using AI models like GPT:

1. **Generate Code Snippets:** Use GPT models to generate code snippets based on given tasks or requirements.
2. **Evaluate Generated Code:** Apply the Computational Time Matrix and Computational Steps Matrix to the generated code to obtain quantitative scores for each metric.

Feedback Mechanisms and Model Training

To refine the optimization process and improve code quality:

1. **Calculate Error:** Determine the error by comparing the expected (ideal) scores with the actual scores obtained from the evaluation matrices.
2. **Train the GPT Model:** Utilize the feedback mechanism to train the GPT model. Adjust the model parameters and training data to minimize errors in the generated code.

By systematically applying these evaluation matrices and algorithms, developers can iteratively improve the efficiency, readability, and performance of generated code, leveraging AI-driven approaches to enhance software development practices.

4.6.3 Output

Optimization output

- Use the trained GPT model to generate optimized Java code.
- Run the optimized code through the evaluation matrices to validate improvements.
- Rank-generated code segments based on efficiency and style metrics.
- Provide suggestions for potential improvements for lower-ranked code segments.

4.7 Summary

This chapter explain about our approach on our proposed solution. We describe about inputs, outputs, and process of the system what we going to implement. In next chapter we describe on analysis and design part of our proposed solution.

Chapter 5 – Analysis and Design

5.1 Introduction

In the previous chapter, our approach of the project in terms of users, inputs, outputs, features, and process were discussed. This chapter presents the top-level architecture diagram of the system and describes each module of the diagram indicating how each module interacts with other modules.

5.2 Modules of the System

5.2.1 Complex Code Generation

Objective: To enhance the quality and efficiency of code generation by incorporating a self-planning phase before the actual code generation.

Analysis

Challenges in Direct Code Generation

1. **Complex Intent Understanding**: Direct code generation models struggle to understand and implement complex human intents.
2. **Error Propagation**: Mistakes in initial steps can propagate, leading to incorrect final outputs.
3. **Scalability Issues**: Handling large-scale, complex problems directly is challenging for both humans and models.

Benefits of Self-Planning

1. **Decomposition of Complex Problems**: Breaking down complex tasks into manageable sub-tasks.
2. **Improved Guidance**: Clear plans provide better guidance for the implementation phase.
3. **Enhanced Code Quality**: Results in code that is more correct, readable, and robust.

Design

Phases of Self-Planning Code Generation

1. **Planning Phase**:
    - **Input**: Natural language intent.
    - **Output**: A detailed plan decomposing the intent into sequential steps.
    - **Process**: Utilize few-shot prompting or LLM model to generate the plan.
2. **Implementation Phase**:
    - **Input**: Detailed plan from the planning phase.
    - **Output**: Generated code that fulfills the initial intent.
    - **Process**: Generate code step-by-step following the detailed plan.

Diagram

Below is a simplified diagram illustrating the self-planning approach:

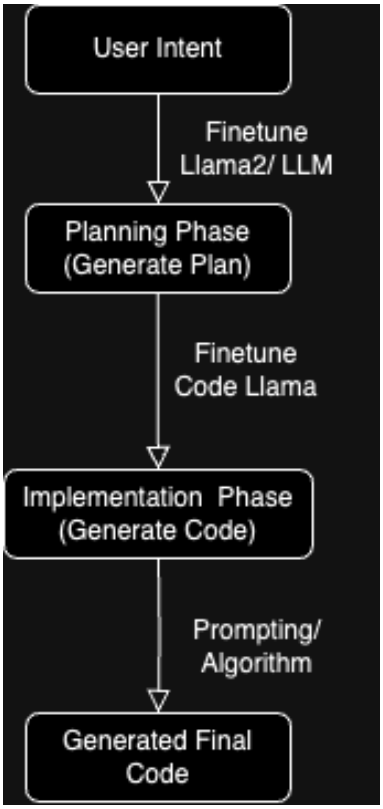


Figure 2: Self planning approach

The self-planning approach for code generation is a two-phase process designed to improve the quality and reliability of generated code by incorporating a planning stage before the actual code generation. The diagram illustrates the flow of this process, starting from the user's intent and culminating in the generation of the final code.

1) **User Intent**

The process begins with the **User Intent**, where the user provides a natural language description of the functionality or task, they want the code to accomplish. This intent serves as the foundational input for the entire self-planning approach. It must be clearly articulated to ensure the subsequent phases can effectively interpret and act on it.

2) **Planning Phase**

Next, the **Planning Phase** comes into play. During this phase, the model takes the user's intent and generates a detailed plan. This plan is created using few-shot prompting or finetuned LLM, where the model is given examples of how to break down similar intents into structured, step-by-step plans. The output of this phase is a comprehensive plan that decomposes the user's intent into smaller, manageable tasks. This decomposition is crucial as it transforms complex, high-level objectives into a series of executable steps, providing clear guidance for the next phase.

3) **Implementation Phase**

Following the planning phase is the **Implementation Phase**. In this phase, the model uses the detailed plan generated earlier to guide the actual code generation process. The implementation phase involves following the plan step-by-step, ensuring each task is addressed systematically. By adhering closely to the plan, the model can generate code that aligns more accurately with the user’s original intent. This structured approach helps mitigate the risk of errors and ensures that the generated code is more robust and easier to understand.

4) **Generated Code**

The final output of the self-planning approach is the **Generated Code**. This code is the direct result of the structured and methodical process outlined in the previous phases. Because the code is generated based on a detailed and well-thought-out plan, it is likely to be more correct, readable, and robust compared to code generated through direct methods. The generated code effectively fulfills the user’s initial intent, demonstrating the advantages of incorporating a planning phase in the code generation process.

5.2.2 Test Case Generation

Objective:

To enhance the quality and efficiency of test case generation by incorporating a self-planning phase before the actual test case generation.

Analysis

Challenges in Test Case Generation

- Complex Requirement Understanding: Direct test case generation models often struggle to fully comprehend and translate complex human requirements into accurate test cases. This can lead to incomplete or incorrect test coverage.

- Error Propagation: Mistakes in early stages of test case generation can propagate, leading to compounded errors in the final test cases. This makes it difficult to identify and correct issues efficiently.

- Scalability Issues: Directly handling large-scale, intricate testing scenarios is challenging for both developers and models. The complexity and volume of test cases required for comprehensive testing can overwhelm standard generation techniques.
- Syntax and Semantic Errors: Generated test cases may contain syntax errors or lack proper structure, making them invalid or ineffective. Ensuring syntactical correctness and meaningful assertions is crucial for test case reliability.

- Alignment with Requirements: Ensuring that generated test cases align with the specified requirements and correctly test the intended functionality is a significant challenge. Misalignment can result in test cases that do not validate the correct behavior of the code.

- Code Coverage: Achieving high code coverage with generated test cases is difficult. Direct generation methods may miss edge cases or complex scenarios, leading to inadequate testing.

- Integration with Generated Code: Ensuring that generated test cases integrate seamlessly with the existing codebase and testing framework can be problematic. Compatibility issues can arise, requiring additional manual intervention.

Direct test case generation can be computationally expensive and time-consuming, especially for large and complex codebases. Optimizing performance while maintaining accuracy is a significant challenge.

Benefits of early and post processing stages

Prompt Design ensures that the LLM accurately understands and generates relevant test cases, improving the initial quality of the output.

Iteration and Feedback Mechanism allow for continuous improvement, enabling the model to learn from its mistakes and generate better test cases over time.

Check the Generated Code with Test Cases validating the generated test cases against the existing code ensures they are functionally correct and meet the specified requirements.

Phases of Test Case Generation

1. Planning Phase:

- Input: Method descriptions in natural language.
- Output: A structured plan outlining the steps to generate corresponding test cases.
- Process: Utilize few-shot prompting or a fine-tuned LLM model to create the plan.

2. Implementation Phase:

- Input: Structured plan from the planning phase.
- Output: Generated test cases that align with the method descriptions and requirements.
- Process: Use the fine-tuned LLM to generate detailed test cases based on the plan, followed by post-processing for syntax validation and correction.

Diagram

Here is the Simplified Architectural diagram of this module

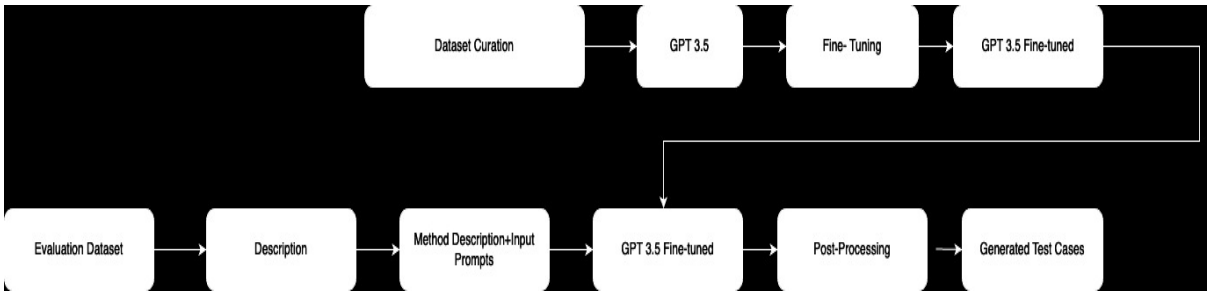


Figure 3: Architectural diagram

Here is the Simplified workflow diagram of this module

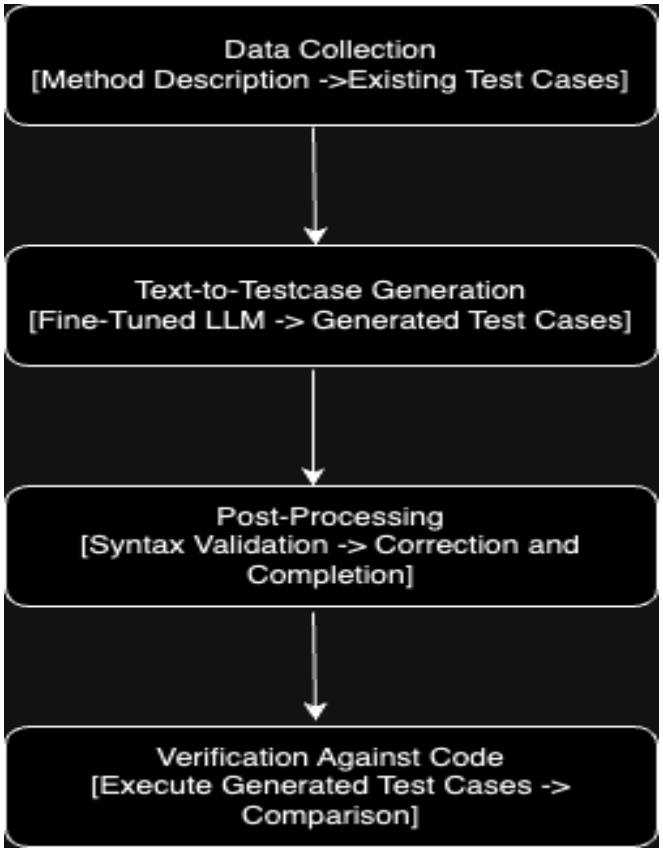


Figure 4: Workflow of Testcase Generation

1. Data Collection

Purpose: Gather a dataset comprising method descriptions paired with their corresponding test cases

Importance: Provides the foundation for training and validating the fine-tuned LLM. This dataset ensures that the LLM learns to generate test cases that accurately reflect the intended functionality described by method descriptions.

Process: Involves collecting method descriptions from various sources and extracting associated test cases, ensuring data quality and relevance to the intended application domain.

2. Text-to-Testcase Generation

Purpose: Use a fine-tuned LLM to automatically generate test cases from given method descriptions in natural language.

Functionality: The LLM processes method descriptions as prompts and generates corresponding test cases based on learned patterns and examples from the training dataset.

Methodology: Leveraging the LLM's ability to understand and generate human-like text, ensuring generated test cases align with the functional requirements described in the method descriptions.

3. Post-Processing

Purpose: Validate, correct, and enhance the syntactic correctness and completeness of generated test cases.

Validation: Check generated test cases for syntax errors or incomplete constructs that may prevent successful compilation or execution.

Correction: Address identified issues by adding missing keywords, balancing parentheses, or modifying generated code to ensure it meets syntactic requirements.

Completion: Ensure that each generated test case fully covers the intended functionality described by the method descriptions, enhancing the reliability and effectiveness of the generated tests.

4. Verification Against Generated Code

Purpose: Execute the generated test cases against the implemented Java code to verify correctness and functional alignment.

Execution: Automate the process of running generated test cases within the development environment to compare expected versus actual outcomes.

Comparison: Evaluate the test results to ensure that generated test cases adequately cover and validate the behavior of the implemented Java code.
Feedback Loop: Provides insights into the quality of generated test cases, guiding iterative improvements in the LLM fine-tuning process or adjustments in the test case generation strategy.

These steps collectively form a structured approach to automate the generation and validation of test cases from method descriptions, leveraging the capabilities of LLMs to enhance efficiency and accuracy in software testing practices.

5.2.3 Code Optimization

Objective: Enhance the prediction of the model using reinforcement learning techniques. Before developing the original model, use a sample model to improve the code optimization methods

Analysis

Challenges Faced when direct approach

1. **Many types of human knowledge are needed: because** complex code is generated across various domains. Experts from different fields are required to implement an effective feedback mechanism.

2. **Increase wrong predictions:** because many resources-based developments occur under one approach. The first time can't confirm that.

Benefits Of Sample Model

1. **Improved Guidance**: Clear plans provide better feedback for identifying and implementing optimization techniques.

2. **Enhanced Prediction Quality**: Results in code that is more efficient, readable, and maintainable.

Design.

The first step in the process involves identifying the functionalities present in the complex code. Subsequently, the code is broken down into smaller, more manageable functionalities, each serving a distinct purpose. Following this decomposition, a collection of sample outputs is generated for each individual function, ensuring that each one produces the same output as the preceding function. This meticulous approach aids in understanding the intricacies of the code and enables a focused optimization effort.

Once the individual functionalities are isolated and their corresponding outputs validated, the next phase involves calculating evaluation metrics. These metrics serve as quantifiable indicators of the code's performance and efficiency. The entire process is aimed at transforming the initially complex code into a more streamlined and optimized version, enhancing its functionality, and ensuring a robust and efficient outcome.

Training Phase

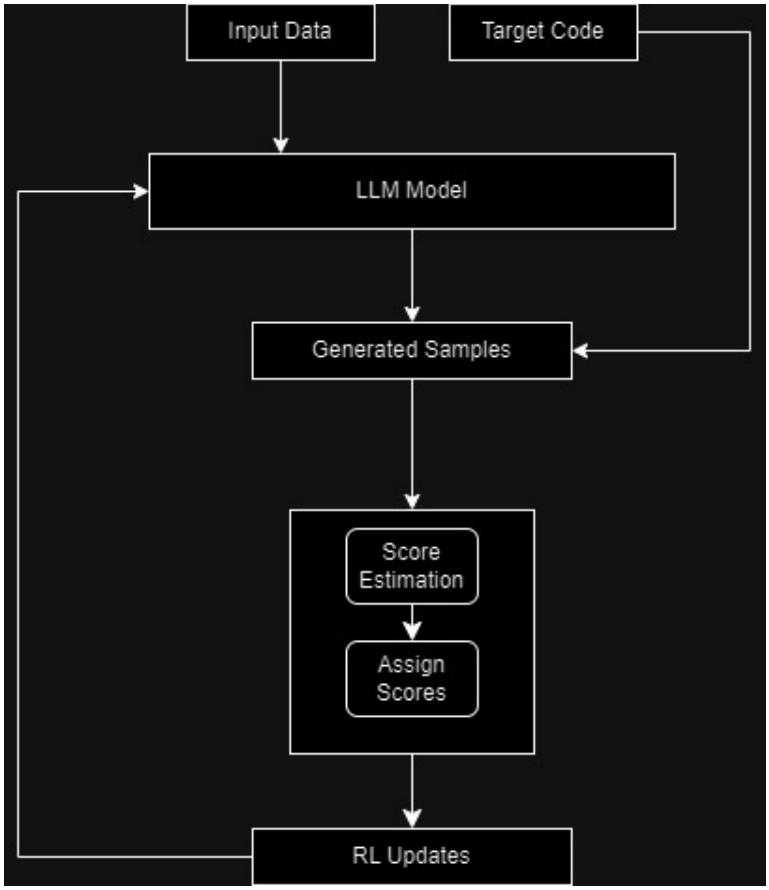


Figure 5: Diagram for training phase

1. Complex Code as Input:

In the initial step, a complex code is taken as input, characterized by a diverse range of functionalities. The primary objective is to identify these functionalities for subsequent score estimation. Each functionality within the code is carefully delineated to facilitate a comprehensive understanding. Subsequently, scores are assigned based on evaluation metrics, providing a quantitative measure of performance and efficiency. This systematic process lays the groundwork for the subsequent optimization and enhancement of the code's overall functionality.

2. Generate Samples:

Utilizing a pre-trained Llama model, multiple sample functionalities are generated for each function, ensuring correct expected outputs. Score values are then assigned to each code sample based on evaluation metrics. The optimization process involves selecting the sample with the minimum score value as the optimized code, indicative of superior performance and efficiency. This method enables the systematic refinement of the original code, aligning it with established benchmarks and enhancing overall functionality.

3. Reinforcement Learning and human feedback:

Incorporating reinforcement learning and human feedback, experts calculate score values for input functionalities. Deviations from optimal performance are then provided as feedback to the model, initiating a continuous feedback loop. This iterative process guides the model towards improved optimization by incorporating human insights. The cycle persists until the model consistently produces the best scores for functions, leveraging both automated learning and expert input to enhance overall performance and align with desired benchmarks.

Generate Optimize Code

The final output of this model provides the optimized version of the given complex code. We use a sample model for training with a limited dataset. This increases domain knowledge and properly handles the feedback mechanism. After the model is well-predicted, we expand the dataset range.

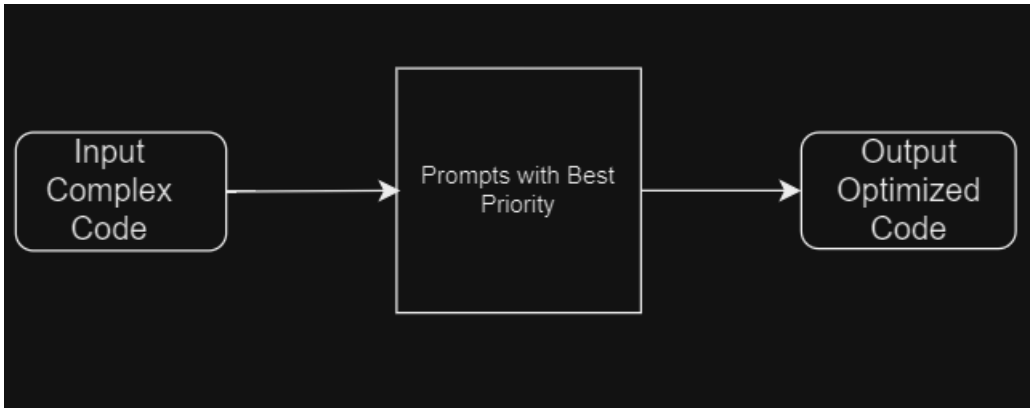


Figure 6: Work diagram

5.3 Summary

In this chapter we discussed about the analysis and design of the system under our study. In the next chapter we will discuss about the implementation of the system under our study.

Chapter 6 – Implementation

6.1 Introduction

Analysis and design of the solution was discussed in the previous chapter. Implementation details of approaches used, techniques, tools and models are discussed in this section based on each module.

6.2 Modules of the System

6.2.1 Complex Code Generation

6.2.1.1 Preprocessing the dataset

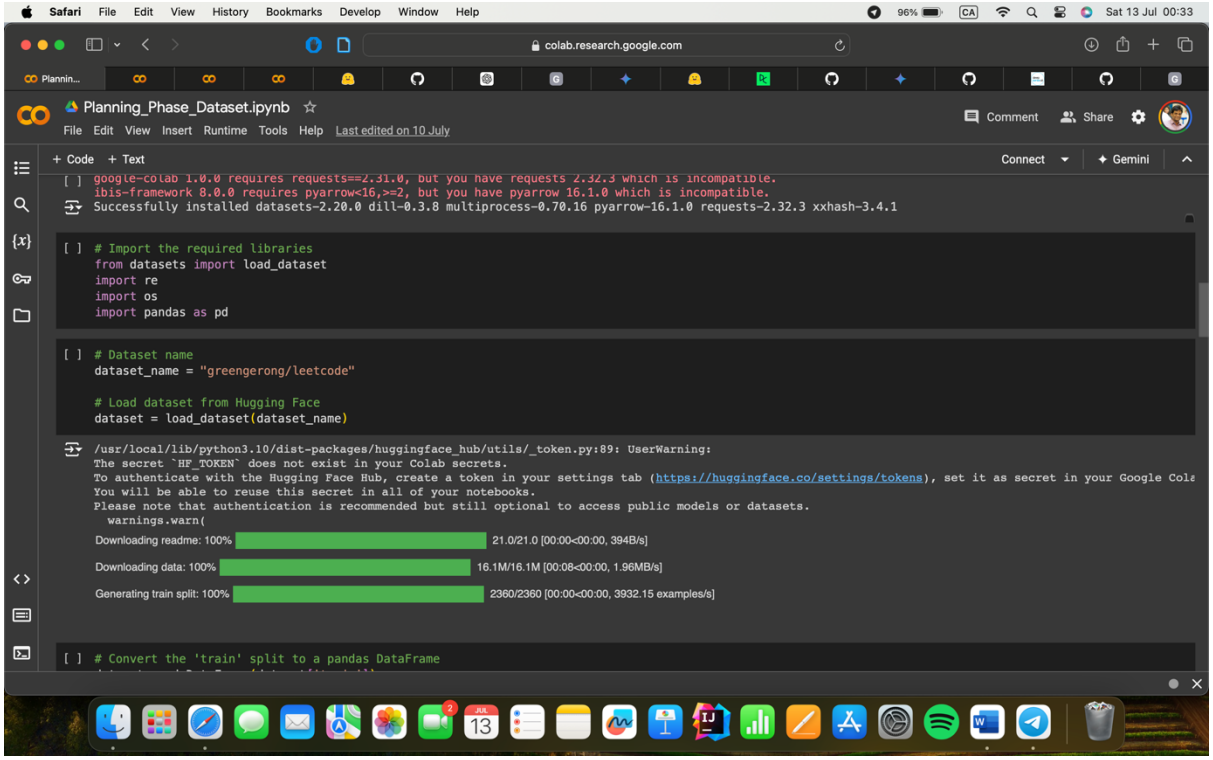


Figure 7: Import dataset from Huggingface.

Planning_Phase_Dataset.ipynb

```python
# Convert the 'train' split to a pandas DataFrame
dataset = pd.DataFrame(dataset['train'])
```

```python
dataset
```

	id	slug	title	difficulty	content	java	c++	python	javascript
0	1	two-sum	Two Sum	Easy	Given an array of integers `nums` and an integ...	\n ```java\nimport java.util.HashMap;\nimpo...	\n ```cpp\n#include <vector>\n#include <uno...	\n ```python\ndef twoSum(nums, target):\n ...	\n ```javascript\nfunction twoSum(nums, tar...
1	2	add-two-numbers	Add Two Numbers	Medium	You are given two **non-empty** linked lists r...	\n ```java\npublic class ListNode {\n in...	\n ```cpp\nstruct ListNode {\n int val;\...	\n ```python\nclass ListNode:\n def __in...	\n ```javascript\nfunction ListNode(val, ne...
2	3	longest-substring-without-repeating-characters	Longest Substring Without Repeating Characters	Medium	Given a string `s`, find the length of the **l...	\n ```java\nimport java.util.HashSet;\nimpo...	\n ```cpp\n#include <string>\n#include <uno...	\n ```python\ndef length_of_longest_substri...	\n ```javascript\nfunction lengthOfLongestS...
3	4	median-of-two-sorted-arrays	Median of Two Sorted Arrays	Hard	Given two sorted arrays `nums1` and `nums2` of...	\n ```java\npublic double findMedianSortedA...	\n ```cpp\ndouble findMedianSortedArrays(ve...	\n ```python\ndef findMedianSortedArrays(nu...	\n ```javascript\nfunction findMedianSorted...
4	5	longest-palindromic-substring	Longest Palindromic Substring	Medium	Given a string `s`, return _the longest_ _pali...	\n ```java\npublic String longestPalindromi...	\n ```cpp\n#include <string>\n\nstd::string...	\n ```python\ndef longest_palindromic_subst...	\n ```javascript\nfunction longestPalindrom...
...	...	...	...	...	...	...	...	...	...
2355	2608	shortest-cycle-in-a-graph	Shortest Cycle in a Graph	Hard	There is a **bi-directional** graph with `n` v...	\n ```java\nimport java.util.ArrayList;\nim...	\n ```cpp\n#include <vector>\n#include <que...	\n ```python\nfrom collections import deque...	\n ```javascript\nfunction shortestCycleLen...

Figure 8: Convert the dataset into dataframe

Planning_Phase_Dataset.ipynb

```python
# Function to check if text is already numbered
def is_numbered(text):
    # Pattern to detect numbering
    numbered_pattern = re.compile(r'^\d+\.\s', re.MULTILINE)
    return bool(numbered_pattern.search(text))
```

```python
# Function to number the text
def number_text(text):
    # Split text into sentences based on periods
    sentences = re.split(r'(?<=\.)\s+', text)
    # Number each sentence
    numbered_sentences = [f"{i+1}) {sentence.strip()}" for i, sentence in enumerate(sentences) if sentence.strip()]
    return ' '.join(numbered_sentences)
```

```python
# Function to extract and process description after Java code block
def extract_and_number(java_content):
    # Pattern to identify code blocks
    code_block_pattern = r'```.*?```'
    match = re.search(code_block_pattern, java_content, re.DOTALL)
    if match:
        # Get text after the code block
        after_code_text = java_content[match.end():].strip()
    else:
        after_code_text = java_content.strip()

    # Number the text if it's not already numbered
    if not is_numbered(after_code_text):
        return number_text(after_code_text)
    return after_code_text
```

Figure 9: Define functions for extract the relevant data from the dataset.

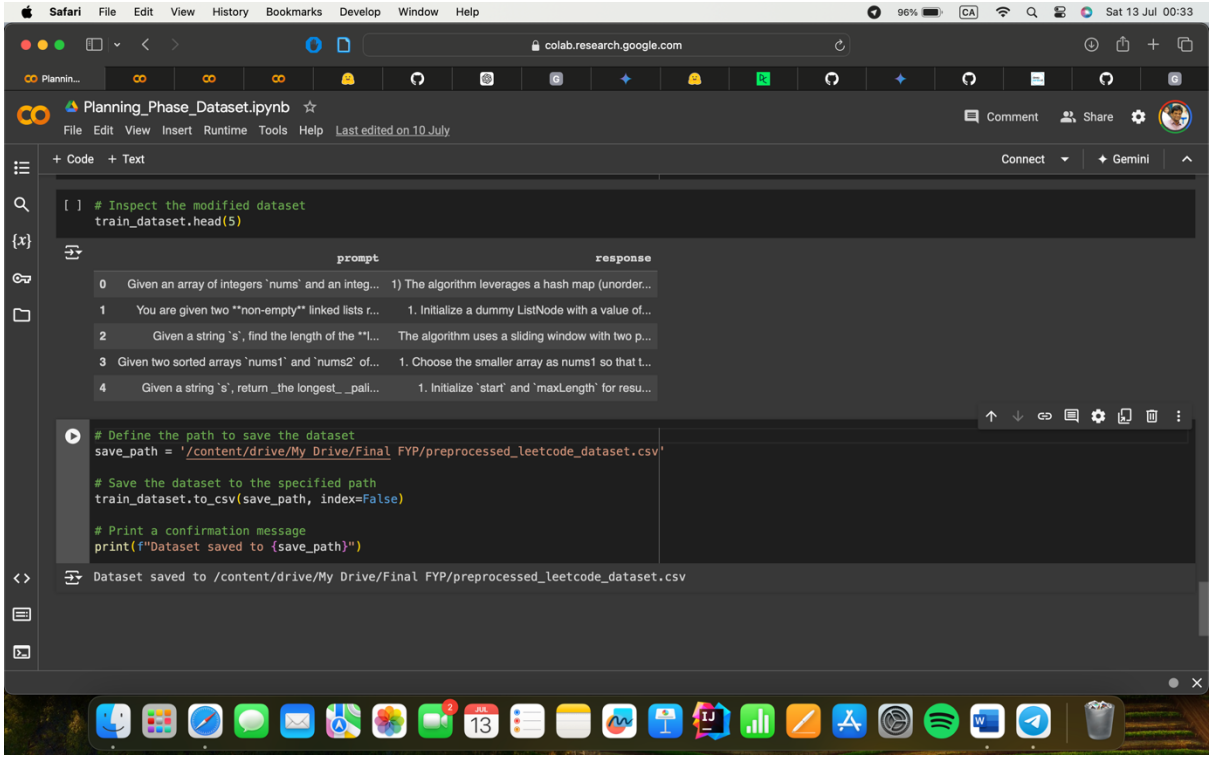


Figure 10: Save the preprocessed dataset in the drive.

6.2.1.2 Finetuned Llama 2 using the dataset

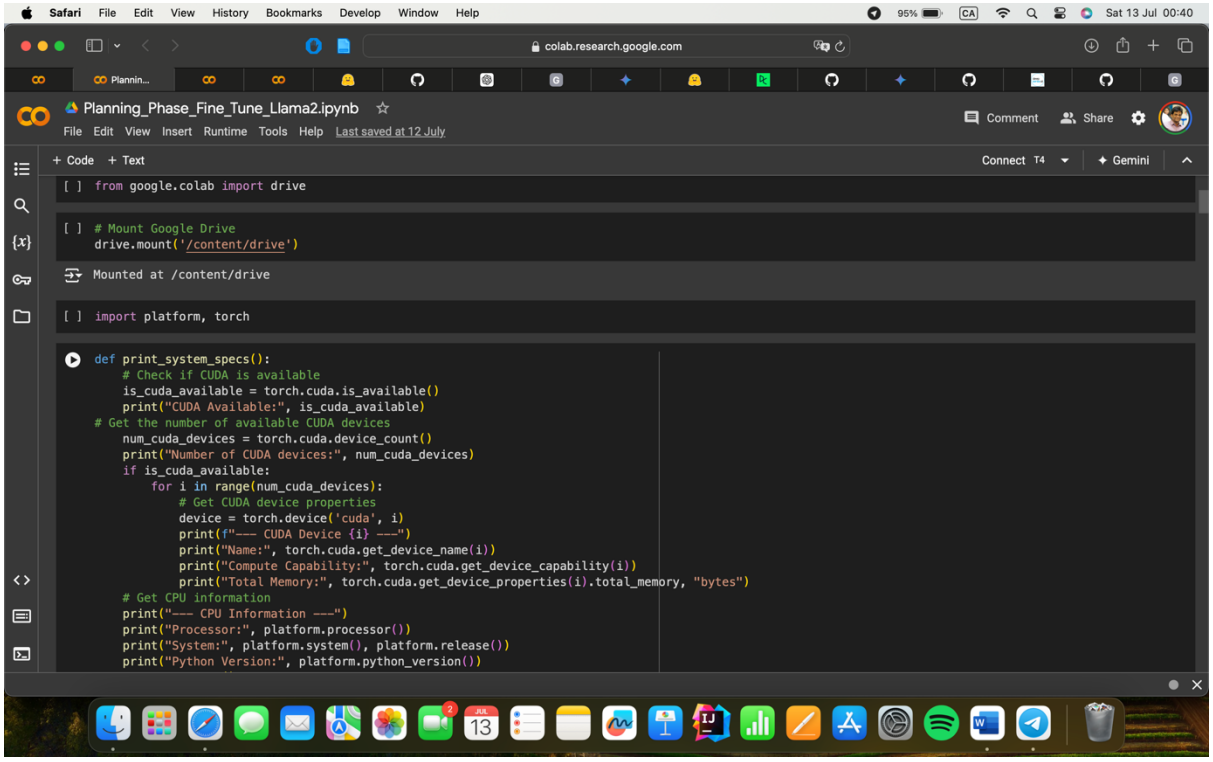


Figure 11: Check the CUDA available or not.

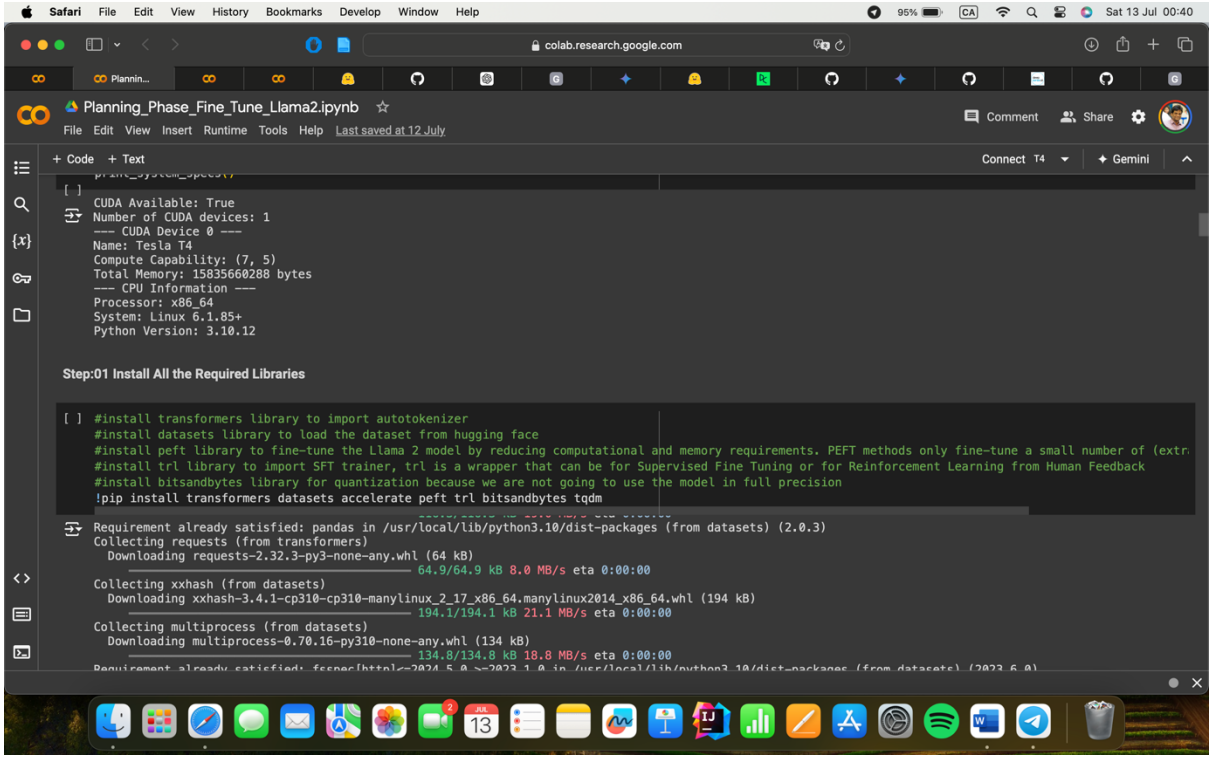


Figure 12: Install necessary libraries.

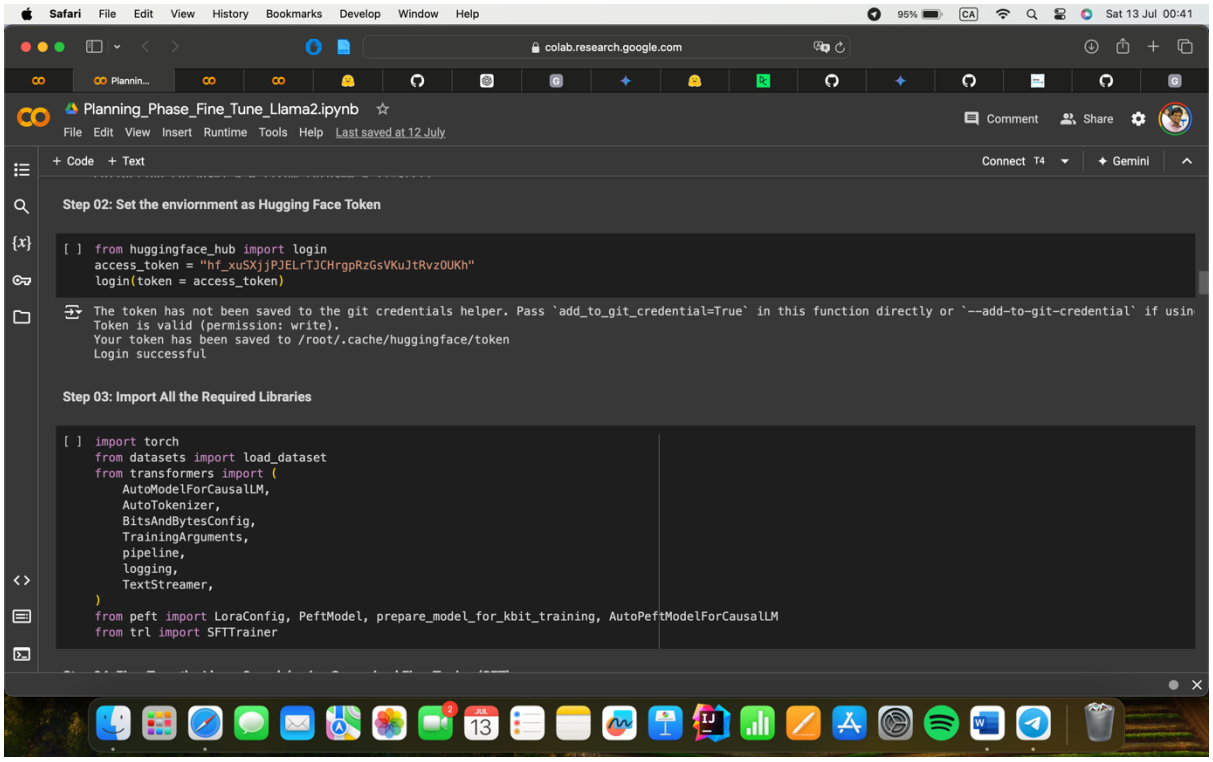


Figure 13: Set Hugging Face Token.

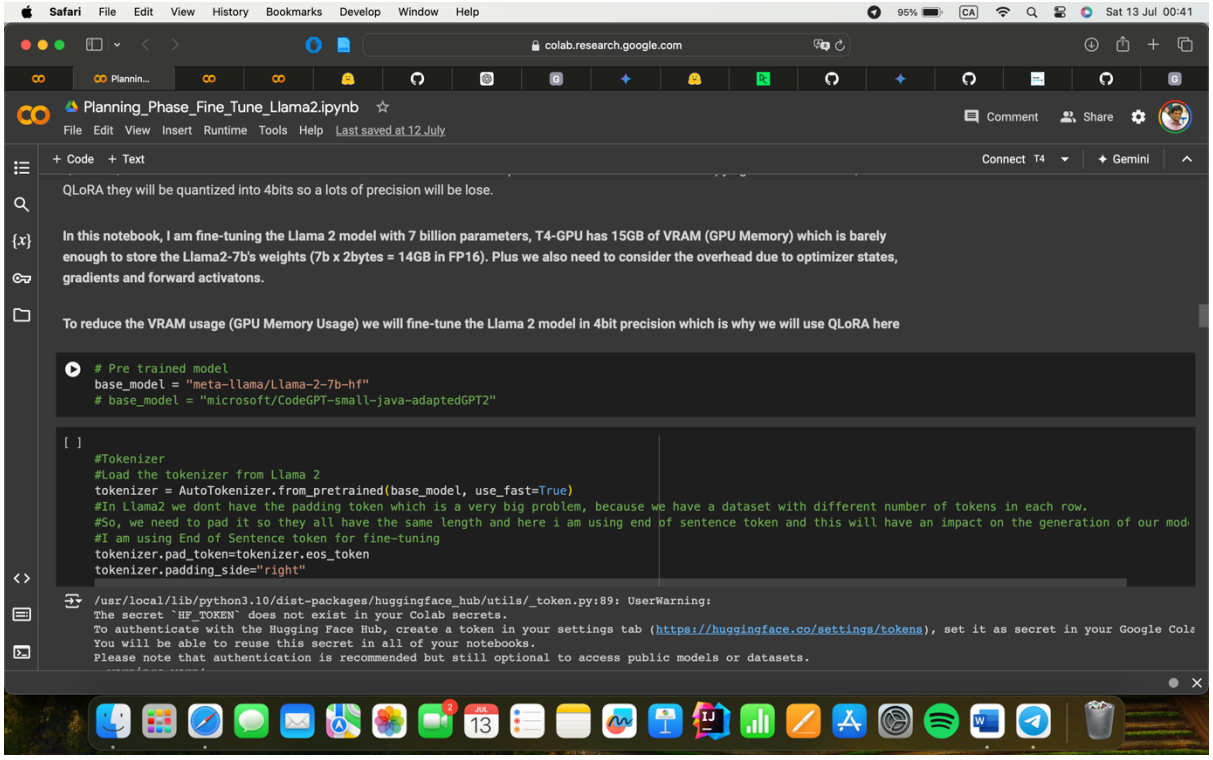


Figure 14: Import model and tokenizer.

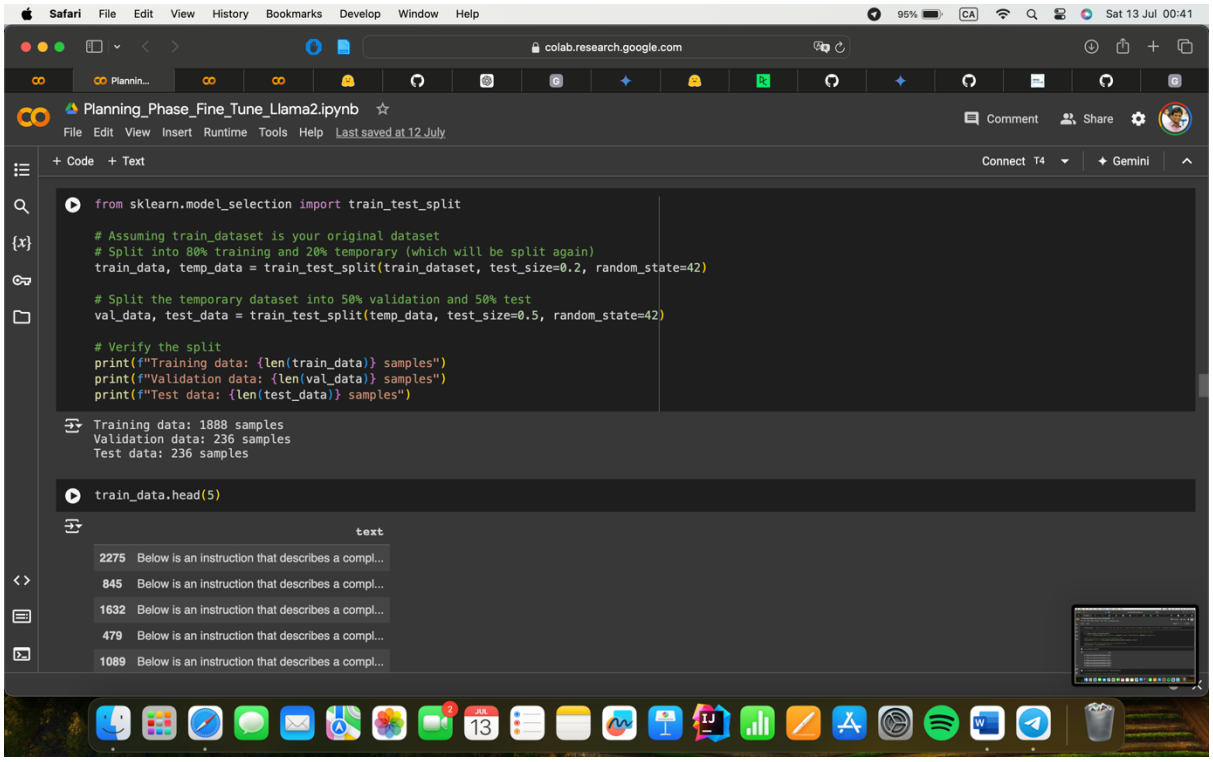


Figure 15: Dataset Split for train, test and validation.

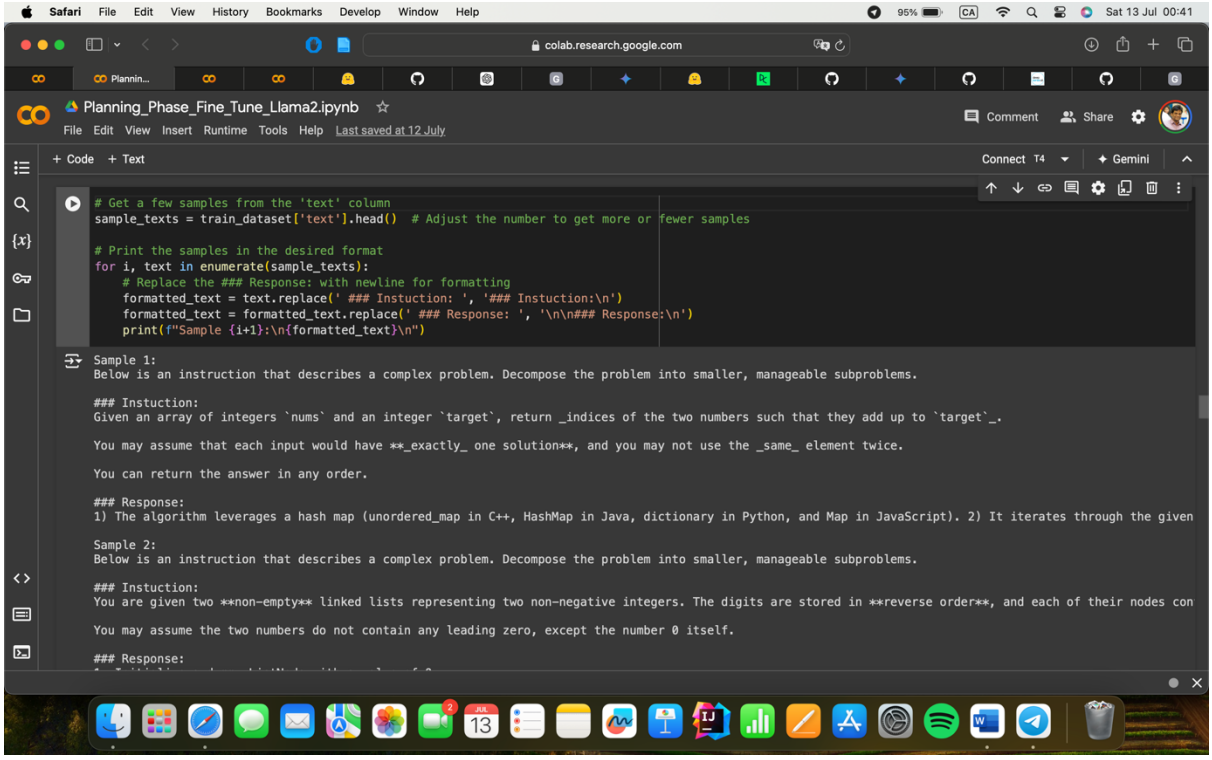


Figure 16: Print samples of a dataset.

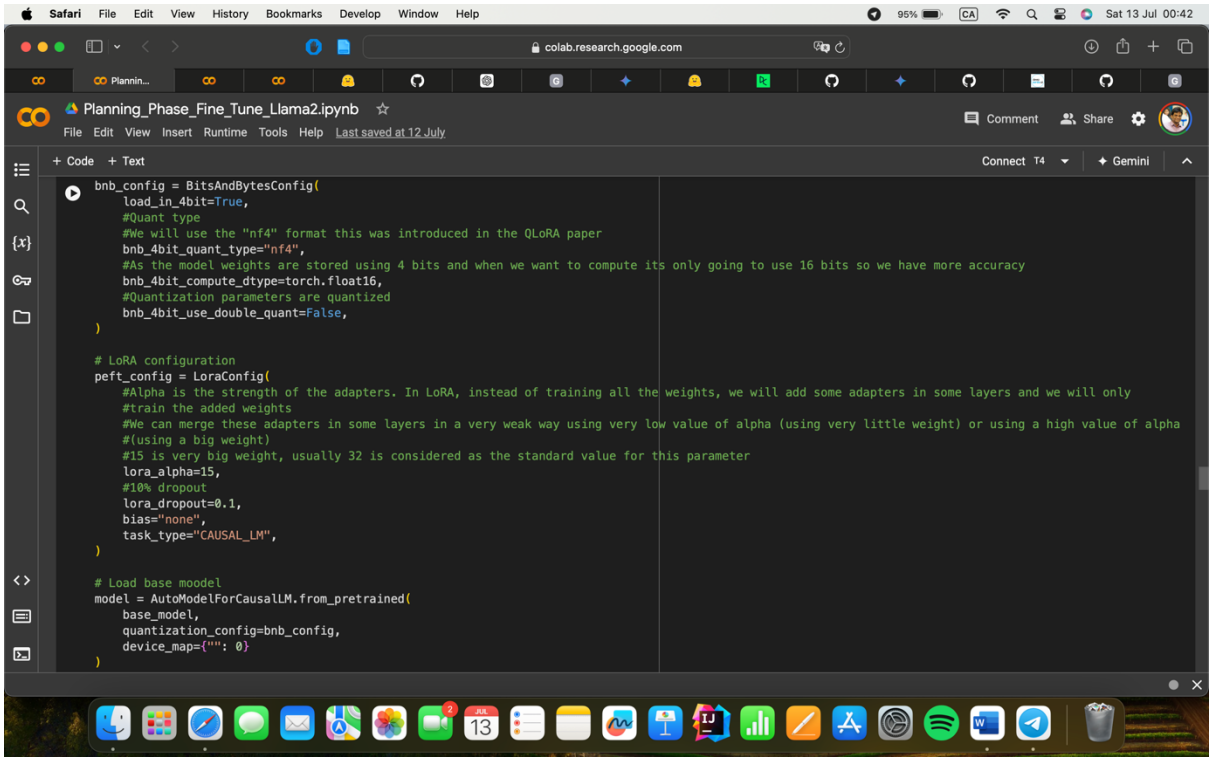


Figure 17: Define LoRA and QloRA for the Llama2 finetuning.

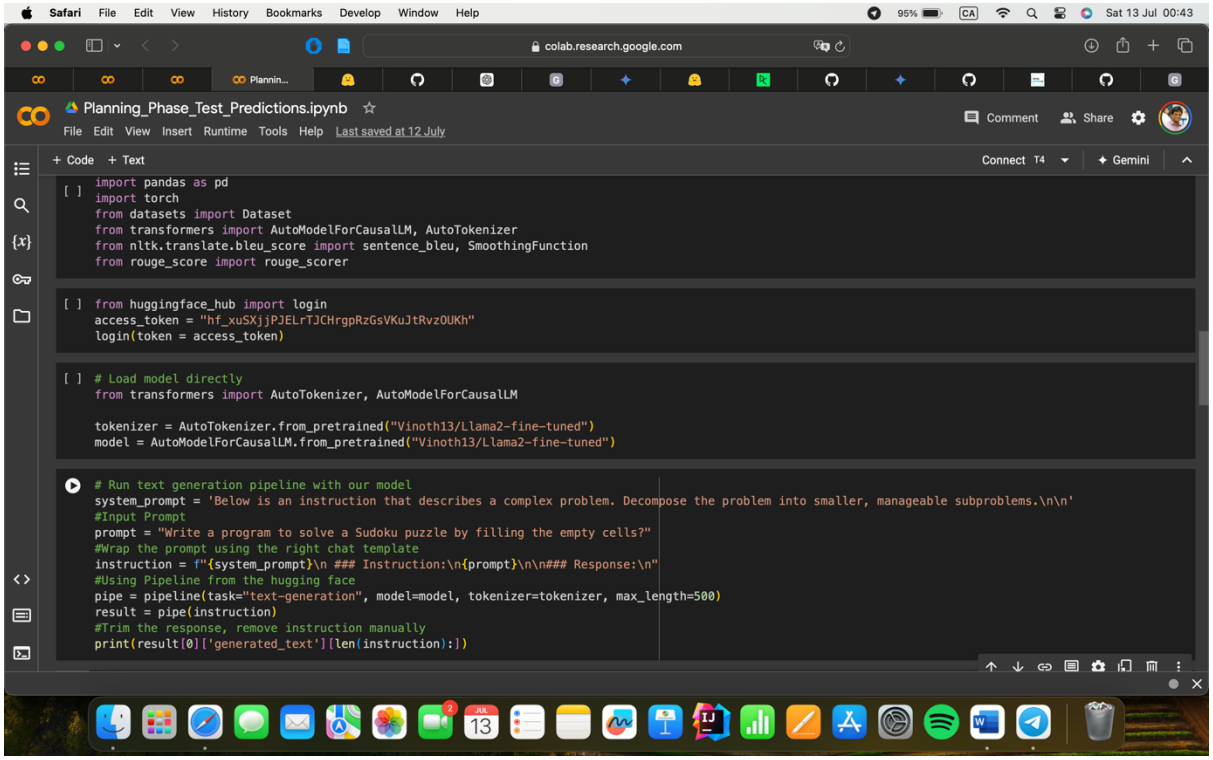


Figure 18: Test the model.

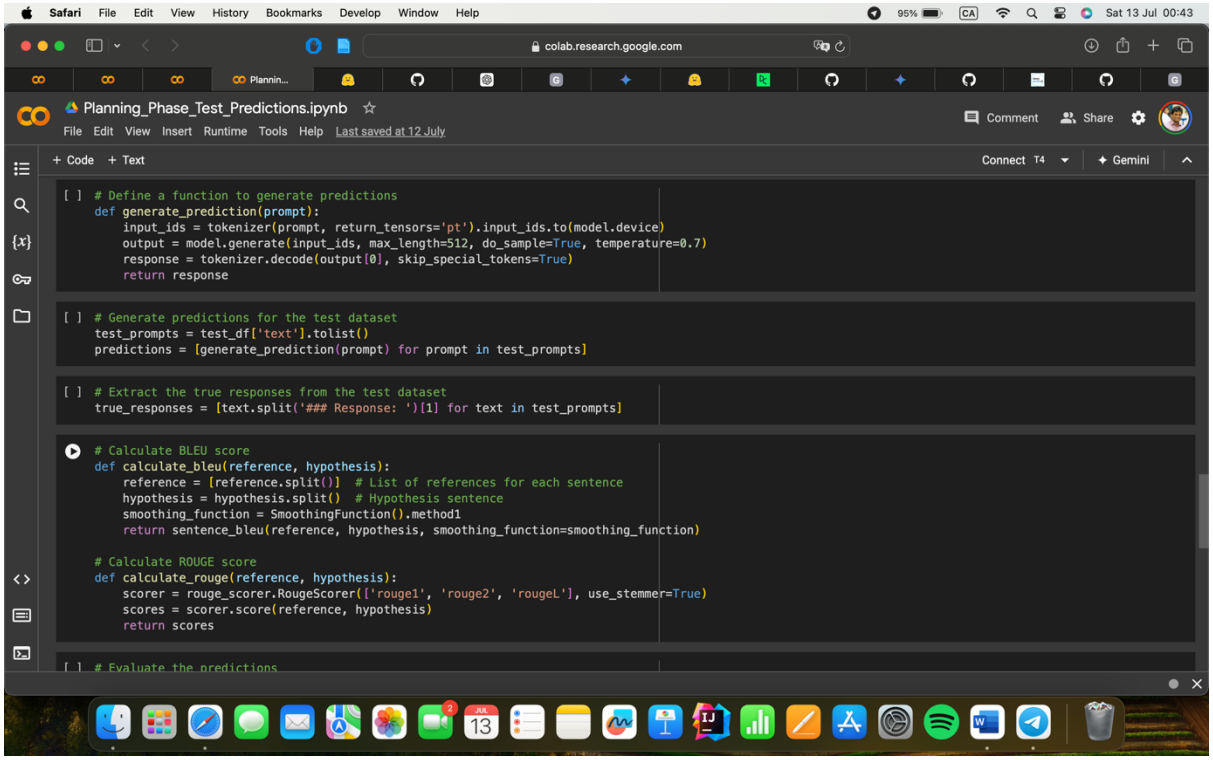


Figure 19: Define the function for calculate the BLEU and ROUGE score.

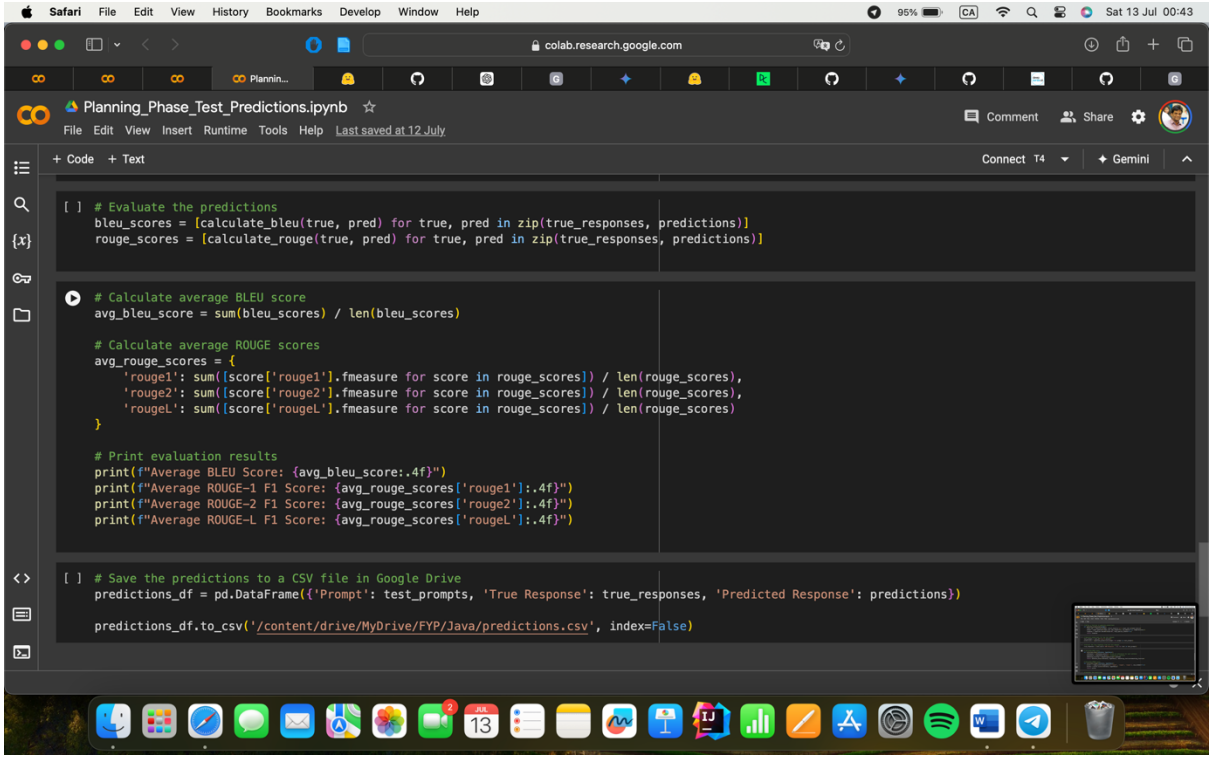
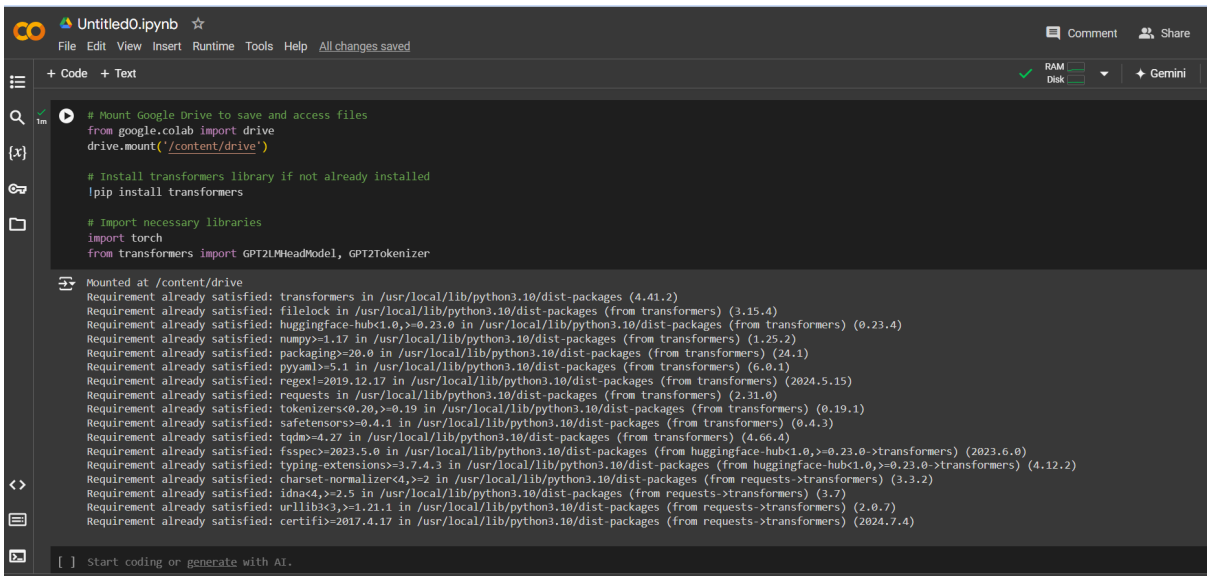


Figure 20: Calculate the average score of BLEU and ROUGE.

6.2.2 Test Case Generation

6.2.2.1 Dataset Generation



```
model_name = 'gpt2'
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)


method_description = "Write a JUnit test case for a method that calculates the factorial of a number."


input_ids = tokenizer.encode(method_description, return_tensors='pt')


output = model.generate(input_ids, max_length=100, num_return_sequences=1, no_repeat_ngram_size=2, repetition_penalty=1.5, length_penalty=1.0, early_stopping=True)
generated_test_case = tokenizer.decode(output[0], skip_special_tokens=True)

print("Generated Test Case:")
print(generated_test_case)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  26.0/26.0 [00:00<00:00, 489B/s]
vocab.json: 100%  1.04M/1.04M [00:00<00:00, 3.11MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 1.86MB/s]
```

6.2.2.2 Testcase generation

```
model_name = 'gpt2'
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)


method_description = "Write a JUnit test case for a method that calculates the factorial of a number."


input_ids = tokenizer.encode(method_description, return_tensors='pt')


output = model.generate(input_ids, max_length=100, num_return_sequences=1, no_repeat_ngram_size=2, repetition_penalty=1.5, length_penalty=1.0, early_stopping=True)
generated_test_case = tokenizer.decode(output[0], skip_special_tokens=True)

print("Generated Test Case:")
print(generated_test_case)
```

```
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `attention_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Generated Test Case:
Write a JUnit test case for a method that calculates the factorial of a number.
The following example shows how to use this technique in your code:

 ( defn sum [ n ] {... } ) ; #=> "1" 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

6.2.2.3 Post processing

```
output = model.generate(input_ids, max_length=100, num_return_sequences=1, no_repeat_ngram_size=2, repetition_penalty=1.5, length_penalty=1.0, early_stopping=True)
generated_test_case = tokenizer.decode(output[0], skip_special_tokens=True)

print("Generated Test Case:")
print(generated_test_case)
```

```
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `attention_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Generated Test Case:
Write a JUnit test case for a method that calculates the factorial of a number.
The following example shows how to use this technique in your code:

 ( defn sum [ n ] {... } ) ; #=> "1" 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

```
# Example post-processing steps (syntax validation)
# Validate test case syntax (example only)
if 'public void testFactorialMethod() {' in generated_test_case:
    print("Test case syntax is correct.")
else:
    print("Test case syntax needs correction.")
```

```
Test case syntax needs correction.
```

```
[9] # Simulate verification against existing Java code
existing_java_code = """
public class MathUtils {
    public int factorial(int n) {
```

6.2.2.4 Verification with Java Code

```
existing_java_code = """
public class MathUtils {
    public int factorial(int n) {
        if (n == 0) {
            return 1;
        }
        return n * factorial(n - 1);
    }
}
"""

expected_output = "Passed"

print("Verification Result:")
print(expected_output)
```

```
Verification Result:
Passed
```

```
[10] # Reporting and analysis (example)
print("Generated Test Case:")
print(generated_test_case)
print("Verification Result:")
print(expected_output)
```

```
Generated Test Case:
Write a JUnit test case for a method that calculates the factorial of a number.
The following example shows how to use this technique in your code:
```

6.2.3 Code Optimization

6.2.3.1 Dataset Generation

```
import os
os.environ["HF_TOKEN"] = "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"

# Install required libraries
!pip install transformers accelerate bitsandbytes

# Import necessary libraries
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
from accelerate import dispatch_model, infer_auto_device_map
import torch
import time
import re
```

```
Collecting accelerate
  Downloading accelerate-0.32.1-py3-none-any.whl (314 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 314.1/314.1 kB 3.3 MB/s eta 0:00:00
Collecting bitsandbytes
  Downloading bitsandbytes-0.43.1-py3-none-manylinux_2_24_x86_64.whl (119.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 119.8/119.8 MB 3.7 MB/s eta 0:00:00
```

Figure 21: Import Colab Libraries

```
# Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")

# Load the model with 8-bit precision
quantization_config = BitsAndBytesConfig(load_in_8bit=True)
model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    quantization_config=quantization_config,
    device_map="auto"
)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  776/776 [00:00<00:00, 28.5kB/s]
tokenizer.model: 100%  500k/500k [00:00<00:00, 6.47MB/s]
tokenizer.json: 100%  1.84M/1.84M [00:00<00:00, 6.11MB/s]
special_tokens_map.json: 100%  414/414 [00:00<00:00, 20.4kB/s]
config.json: 100%  609/609 [00:00<00:00, 19.9kB/s]
```

Figure 22: Load Tokenizer

```python
[ ] # Function to calculate cyclomatic complexity
    def calculate_cyclomatic_complexity(java_code):
        decision_points = len(re.findall(r'\b(if|for|while|case|catch)\b', java_code))
        conditions = len(re.findall(r'\|\||&&', java_code))
        cyclomatic_complexity = decision_points + conditions + 1
        return cyclomatic_complexity
```

```python
# Function to calculate time complexity
def calculate_time_complexity(java_code):
    lines = java_code.split("\n")
    max_depth = 0
    current_depth = 0
    in_loop = False

    for line in lines:
        stripped_line = line.strip()
        if re.match(r'\b(for|while)\b', stripped_line):
            current_depth += 1
            in_loop = True
            if current_depth > max_depth:
                max_depth = current_depth
        elif "{" in stripped_line:
            if in_loop:
                current_depth += stripped_line.count("{")
                if current_depth > max_depth:
                    max_depth = current_depth
        elif "}" in stripped_line:
            current_depth -= stripped_line.count("}")
            if current_depth <= 0:
                in_loop = False

    if max_depth == 0:
        time_complexity = "O(1)"
    else:
        time_complexity = f"O(n^{max_depth})"

    return time_complexity
```

Figure 23: Code for time complexity

```python
# Function to analyze Java code
def analyze_java_code(java_code):
    num_loops = java_code.count("for") + java_code.count("while")
    num_conditionals = java_code.count("if") + java_code.count("else")
    num_functions = java_code.count("public") + java_code.count("private") + java_code.count("protected")
    cyclomatic_complexity = calculate_cyclomatic_complexity(java_code)
    time_complexity = calculate_time_complexity(java_code)

    input_text = (
        f"Analyze the following Java code:\n\n{java_code}\n\n"
        f"Number of loops: {num_loops}, Number of conditionals: {num_conditionals}, "
        f"Number of functions: {num_functions}, Cyclomatic complexity: {cyclomatic_complexity}, "
        f"Time complexity: {time_complexity}.\n"
    )

    # Tokenize the input text
    input_ids = tokenizer(input_text, return_tensors="pt").to("cuda")

    # Measure the time taken for model inference
    start_time = time.time()
    with torch.no_grad():
        outputs = model.generate(**input_ids, max_new_tokens=150, num_return_sequences=1)
    end_time = time.time()

    output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return output_text, end_time - start_time
```

```
java_code = """
public class Example {
    public static void main(String[] args) {
        int n = 10;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.println(i * j);
            }
        }
    }
}
"""

# Analyze the Java code and print the results
output_text, inference_time = analyze_java_code(java_code)
print("Model Output:\n", output_text)
print("Inference Time: {:.2f} seconds".format(inference_time))
```

Figure 24: Analyze the code

```
Model Output:
 Analyze the following Java code:


public class Example {
    public static void main(String[] args) {
        int n = 10;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.println(i * j);
            }
        }
    }
}


Number of loops: 2, Number of conditionals: 0, Number of functions: 2, Cyclomatic complexity: 3, Time complexity: O(n^2).
```

Figure 25: Output java code

6.2.3.2 Preprocessing the dataset phase

```
import numpy as np
import pandas as pd
import torch
from datasets import ClassLabel, DatasetDict, load_dataset
from evaluate import load

from transformers import (
    AutoModelForSequenceClassification,
    AutoTokenizer,
    DataCollatorWithPadding,
    Trainer,
    TrainerCallback,
    TrainingArguments,
    set_seed,
)
```

Figure 26: Import Colab libraries

Load Train data

```
[18] train_ds = load_dataset("giganticode/java-cmpx", split="train")
     print(next(iter(train_ds)))
     print("Train dataset size : ",len(train_ds))
```

```
{'method_id': '8905281e-c3a3-4c34-8be7-feaac26638c1', 'cyclomatic_complexity': 0, 'method_text': 'public int getNowYear() {\n\t\treturn Integer.parseInt(sysNowTime[5]);\n\t}'}
Train dataset size :  200000
```

Load Test data

```
[19] test_ds = load_dataset("giganticode/java-cmpx", split="test")
     print(next(iter(test_ds)))
     print("Test dataset size : ",len(test_ds))
```

```
{'method_id': '72d208df-414f-4e56-a115-bac2a56596fa', 'cyclomatic_complexity': 1, 'method_text': 'public void setEnabled(boolean enabled) {\n\t\tsuper.setEnabled(enabled);\n\n\t\tif (
Test dataset size :  50000
```

Figure 27: Load data

```
[20] test_validation_ds = test_ds.train_test_split(test_size=0.2)
```

```
test_validation_ds
```

Show hidden output

```
[26] test_validation_df=pd.DataFrame(test_validation_ds['test'])
     test_validation_df
```

	method_id	cyclomatic_complexity	method_text
0	ccd1ea98-3d31-453b-b8cb-28823c9797ad	3	public int removeDuplicates(int[] A) {\n ...
1	539abe94-6ae0-4bc9-8a6a-e3ecfc9b6165	7	private void cftbsub(int n, double[] a, int[] ...
2	927782c4-9200-46d1-9cfe-6c81bdc3f73f	5	public String toString(ArrayList<Integer> resu...
3	066b78aa-c746-4dc1-b680-6e0e562c02d4	3	private final void processArgs(String[] args) ...
4	63eee262-f07c-42cb-8900-b94db1d70e3d	6	public static void main(final String[] args) t...
...	...	...	...
9995	22cf8586-0ed2-40c0-afc4-9718d2fd0a60	4	public ArrayList<MT950> Query(String Vdate,Str...
9996	91df7066-368d-4bc2-a138-f78b917f93e5	0	public void visitBlock(JCBlock tree) {\n ...
9997	32024a0c-041d-45dc-befb-93a8693ee4d2	6	@Override\n\tpublic boolean okMessage(final En...

Figure 28: Split data for validation test

Organize Train, Validation and Test data

```
[22] train_validation_test = DatasetDict(
            {
                "train": train_ds,
                "valid": test_validation_ds["test"],
                "test": test_validation_ds["train"],
            }
        )
```

Loading tokenizer and model

```
[9] tokenizer = AutoTokenizer.from_pretrained("microsoft/unixcoder-base-nine")
    tokenizer.pad_token = tokenizer.eos_token
    model = AutoModelForSequenceClassification.from_pretrained("microsoft/unixcoder-base-nine", num_labels=10)
    model.config.pad_token_id = model.config.eos_token_id
```

```
tokenizer_config.json: 100%  1.11k/1.11k [00:00<00:00, 70.1kB/s]
vocab.json: 100%  938k/938k [00:00<00:00, 3.57MB/s]
merges.txt: 100%  444k/444k [00:00<00:00, 22.4MB/s]
special_tokens_map.json: 100%  772/772 [00:00<00:00, 54.3kB/s]
```

```
labels = ClassLabel(num_classes=10, names=list(set(train_validation_test["train"]["cyclomatic_complexity"])))

def tokenize(example):
    inputs = tokenizer(example["method_text"], truncation=True, max_length=1024)
    label = labels.str2int(example["cyclomatic_complexity"])
    return {
        "input_ids": inputs["input_ids"],
        "attention_mask": inputs["attention_mask"],
        "label": label,
    }

tokenized_datasets = train_validation_test.map(
    tokenize,
    batched=True,
    remove_columns=train_validation_test["train"].column_names,
)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

```
Map: 100%  200000/200000 [02:13<00:00, 1228.95 examples/s]
Map: 100%  10000/10000 [00:06<00:00, 1744.45 examples/s]
Map: 100%  40000/40000 [00:25<00:00, 1728.52 examples/s]
```

Figure 29: Load tokenizer and model.

6.3 Summary

This chapter discussed about the ongoing process of implementation of the research. Next chapter will give an overall discussion about the research, its inputs, outputs, limitations, and further work.

Chapter 7 – Discussion

7.1 Introduction

The previous chapter summarized the implementation details of the system. This section gives a brief description about the summary of work, limitations, significance from similar approaches and further work. Finally, this chapter states the suggestions and distinguishes further improvements.

7.2 Discussion

The project's foundation lies in recognizing the increasing demand for sophisticated algorithms in software development and the limitations of traditional manual approaches. By harnessing the power of generative models, the research aims to streamline the complex Java code generation process. The proposed solution integrates the capabilities of pre-trained LLaMA models into three key modules.

The first module focuses on decomposing complex Java code into smaller modules using a specialized Llama model. These modules are then processed by Java-generating GPT models to reconstruct the intricate code structure. The second module addresses the validation and correctness of the generated code by employing the same generative model for test case generation. It aims to create a robust suite of test cases tailored to evaluate the reliability and correctness of the generated Java code. The third module introduces a code optimization aspect, evaluating computational time and the number of steps required to execute the code. Feedback mechanisms are integrated to train the models based on these scores, ensuring optimized and efficient code.

The technology adapted for the project includes Python programming language, machine learning libraries like Scikit-learn, NumPy, and Pandas, natural language processing libraries like NLTK, and development tools such as Git-Lab for version control. The proposed system aims to cater to developers and programmers seeking to expedite code development while ensuring correctness, robustness, and optimization.

The top-level architecture diagram illustrates the interaction of modules, emphasizing the seamless flow of data from complex code input to integrated complex Java code output. The Code Generation module employs Llama models and GPT models, the Test Case Generation module analyzes and generates test cases based on the Code-Llama model output, and the Code Optimization module refines the generated code based on computational metrics.

The analysis and design chapter provides a detailed breakdown of each module, outlining inputs, processes, and outputs. It emphasizes the significance of advanced language models in achieving the project's goals and highlights the interconnectedness of the modules.

Overall, the project presents a pioneering approach to automated Java code generation, combining the strengths of generative models, machine learning, and advanced testing methodologies. The integration of these technologies addresses the challenges in algorithmic design and opens avenues for continuous improvement and innovation in software development practices.

7.3 Summary

This chapter presents a discussion about the work done, assumptions, inputs, outputs, and limitations of the approach. In addition, further work to be done and the scope of the research is stated clearly. The next chapter is a list of references used in the research.

References

[1] "Code Llama: Llama 2 learns to code," *huggingface.co*. https://huggingface.co/blog/codellama (accessed Jan. 13, 2024).

[2] J. López Espejel, M. S. Yahaya Alassan, E. M. Chouham, W. Dahhane, and E. H. Ettifouri, "A comprehensive review of State-of-The-Art methods for Java code generation from Natural Language Text," *Natural Language Processing Journal*, vol. 3, p. 100013, Jun. 2023, doi: 10.1016/j.nlp.2023.100013.

[3] S. Manning, P. Mishkin, G. Hadfield, and E. Eisner, "A Research Agenda for Assessing the Economic Impacts of Code Generation Models".

[4] C. Zhang *et al.*, "A Survey of Automatic Source Code Summarization," *Symmetry*, vol. 14, no. 3, Art. no. 3, Mar. 2022, doi: 10.3390/sym14030471.

[5] E. Al-Hossami and S. Shaikh, "A Survey on Artificial Intelligence for Source Code: A Dialogue Systems Perspective," no. arXiv:2202.04847. arXiv, Feb. 10, 2022. doi: 10.48550/arXiv.2202.04847.

[6] A. H. Mohammadkhani, N. S. Bommi, M. Daboussi, O. Sabnis, C. Tantithamthavorn, and H. Hemmati, "A Systematic Literature Review of Explainable AI for Software Engineering," no. arXiv:2302.06065. arXiv, Feb. 12, 2023. doi: 10.48550/arXiv.2302.06065.

[7] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, vol. 204, p. 111796, Oct. 2023, doi: 10.1016/j.jss.2023.111796.

[8] S. Rai, R. C. Belwal, and A. Gupta, "A Review on Source Code Documentation," *ACM Trans. Intell. Syst. Technol.*, vol. 13, no. 5, p. 84:1-84:44, Jun. 2022, doi: 10.1145/3519312.

[9] N. Fatima, A. S. Imran, Z. Kastrati, S. M. Daudpota, and A. Soomro, "A Systematic Literature Review on Text Generation Using Deep Neural Network Models," *IEEE Access*, vol. 10, pp. 53490–53503, 2022, doi: 10.1109/ACCESS.2022.3174108.

[10] "A Systematic Literature Review on Text Generation Using Deep Neural Network Models | IEEE Journals & Magazine | IEEE Xplore." Accessed: Aug. 25, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9771452

[11] G. C. Velasco, N. A. P. Vaz, and S. T. Carvalho, "Challenges and Opportunities in Smart Contract Development on the Ethereum Virtual Machine: A Systematic Literature Review," in *Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain)*, SBC, May 2023, pp. 15–28. doi: 10.5753/wblockchain.2023.756.

[12] S. B. Pandi, S. A. Binta, and S. Kaushal, "Artificial Intelligence for Technical Debt Management in Software Development." arXiv, Jun. 16, 2023. Accessed: Aug. 18, 2023. [Online]. Available: http://arxiv.org/abs/2306.10194

[13] J. G. Meyer *et al.*, "ChatGPT and large language models in academia: opportunities and challenges," *BioData Mining*, vol. 16, no. 1, p. 20, Jul. 2023, doi: 10.1186/s13040-023-00339-9.

[14] L. Ayinde, M. P. Wibowo, B. Ravuri, and F. B. Emdad, "ChatGPT as an important tool in organizational management: A review of the literature," *Business Information Review*, p. 02663821231187991, Jul. 2023, doi: 10.1177/02663821231187991.

[15] R. Gozalo-Brizuela and E. C. Garrido-Merchan, "ChatGPT is not all you need. A State-of-the-Art Review of large Generative AI models," no. arXiv:2301.04655. arXiv, Jan. 11, 2023. doi: 10.48550/arXiv.2301.04655.

[16] M. Sallam, "ChatGPT Utility in Healthcare Education, Research, and Practice: Systematic Review on the Promising Perspectives and Valid Concerns," *Healthcare*, vol. 11, no. 6, Art. no. 6, Jan. 2023, doi: 10.3390/healthcare11060887.

[17] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 121–154, Jan. 2023, doi: 10.1016/j.iotcps.2023.04.003.

[18] S. Z. Hassan, A. Irshad, J. Mir, A. Aslam, K. Ayaz, and M. A. Bawazir, "Code Comment Analysis–A Review Paper," *Journal of Management Practices, Humanities and Social Sciences*, vol. 6, no. 1, Art. no. 1, Jan. 2022, doi: 10.33152/jmphss-6.1.9.

[19] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," *IEEE Access*, vol. 10, pp. 82434–82455, 2022, doi: 10.1109/ACCESS.2022.3196347.

[20] G. Sebastián, J. A. Gallud, and R. Tesoriero, "Code generation using model driven architecture: A systematic mapping study," *Journal of Computer Languages*, vol. 56, p. 100935, Feb. 2020, doi: 10.1016/j.cola.2019.100935.

[21] L. Wang, X. Ge, L. Liu, and G. Hu, "Code Interpreter for Bioinformatics: Are We There Yet?," *Ann Biomed Eng*, Jul. 2023, doi: 10.1007/s10439-023-03324-9.

[22] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, in ICPC '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 200–210. doi: 10.1145/3196321.3196334.

[23] Z. Yang, S. Chen, C. Gao, Z. Li, G. Li, and R. Lv, "Deep Learning Based Code Generation Methods: A Literature Review," no. arXiv:2303.01056. arXiv, Mar. 02, 2023. doi: 10.48550/arXiv.2303.01056.

[24] T. H. M. Le, H. Chen, and M. A. Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," *ACM Comput. Surv.*, vol. 53, no. 3, p. 62:1-62:38, Jun. 2020, doi: 10.1145/3383458.

[25] "Dimensionality Reduction and Visualization of Bharatanatyam Mudras | International Journal of Image and Graphics." Accessed: Sep. 03, 2023. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0219467823500018

[26] N. A. Parikh, "Empowering Business Transformation: The Positive Impact and Ethical Considerations of Generative AI in Software Product Management -- A Systematic Literature Review." arXiv, Jun. 05, 2023. doi: 10.48550/arXiv.2306.04605.

[27] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "ExploitGen: Template-augmented exploit code generation based on CodeBERT," *Journal of Systems and Software*, vol. 197, p. 111577, Mar. 2023, doi: 10.1016/j.jss.2022.111577.

[28] M. M. Rahman, Y. Watanobe, A. Shirafuji, and M. Hamada, "Exploring Automated Code Evaluation Systems and Resources for Code Analysis: A Comprehensive Survey," no. arXiv:2307.08705. arXiv, Jul. 08, 2023. doi: 10.48550/arXiv.2307.08705.

[29] S. Paul and P. P. Das, "Fast Detection and Rule Based Classification of Bharatanatyam hasta mudra," in *Computer Vision and Image Processing*, D. Gupta, K. Bhurchandi, S. Murala, B. Raman, and S. Kumar, Eds., in Communications in Computer and Information Science. Cham: Springer Nature Switzerland, 2023, pp. 589–603. doi: 10.1007/978-3-031-31407-0_44.

[30] A. Pathak, "Exploring Chatgpt: An Extensive Examination of its Background, Applications, Key Challenges, Bias, Ethics, Limitations, and Future Prospects." Rochester, NY, Jul. 03, 2023. doi: 10.2139/ssrn.4499278.

[31] "Fast Detection and Rule Based Classification of Bharatanatyam hasta mudra | SpringerLink." Accessed: Sep. 03, 2023. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-31407-0_44

[32] V. Rajapakse, I. Karunanayake, and N. Ahmed, "Intelligence at the Extreme Edge: A Survey on Reformable TinyML," *ACM Comput. Surv.*, vol. 55, no. 13s, p. 282:1-282:30, Jul. 2023, doi: 10.1145/3583683.

[33] X. Hou *et al.*, "Large Language Models for Software Engineering: A Systematic Literature Review." arXiv, Aug. 21, 2023. doi: 10.48550/arXiv.2308.10620.

[34] E. N. Crothers, N. Japkowicz, and H. L. Viktor, "Machine-Generated Text: A Comprehensive Survey of Threat Models and Detection Methods," *IEEE Access*, vol. 11, pp. 70977–71002, 2023, doi: 10.1109/ACCESS.2023.3294090.

[35] A. Beganovic, M. A. Jaber, and A. A. Almisreb, "Methods and Applications of ChatGPT in Software Development: A Literature Review," *Southeast Europe Journal of Soft Computing*, vol. 12, no. 1, Art. no. 1, May 2023, doi: 10.21533/scjournal.v12i1.251.

[36] "Mobile Information Systems - Google Scholar Metrics." Accessed: Aug. 11, 2023. [Online].Available: https://scholar.google.com/citations?hl=en&view_op=search_venues&vq=Mobile+Information+Systems&btnG=

[37] N. Sanchez-Gomez, J. Torres-Valderrama, J. A. Garcia-Garcia, J. J. Gutierrez, and M. J. Escalona, "Model-Based Software Design and Testing in Blockchain Smart Contracts: A Systematic Literature Review," *IEEE Access*, vol. 8, pp. 164556–164569, 2020, doi: 10.1109/ACCESS.2020.3021502.

[38] A. Bucaioni, A. Cicchetti, and F. Ciccozzi, "Modelling in low-code development: a multi-vocal systematic review," *Softw Syst Model*, vol. 21, no. 5, Art. no. 5, Oct. 2022, doi: 10.1007/s10270-021-00964-0.

[39] E. Heisler, A. Deshmukh, S. Mazumder, P. Sadayappan, and H. Sundar, "Multi-discretization domain specific language and code generation for differential equations," *Journal of Computational Science*, vol. 68, p. 101981, Apr. 2023, doi: 10.1016/j.jocs.2023.101981.

[40] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review," *Entropy*, vol. 25, no. 6, Art. no. 6, Jun. 2023, doi: 10.3390/e25060888.

[41] G. M. Hocky and A. D. White, "Natural language processing models that automate programming will transform chemistry research and teaching," *Digital Discovery*, vol. 1, no. 2, Art. no. 2, 2022, doi: 10.1039/D1DD00009H.

[42] R. Milewicz, P. Pirkelbauer, P. Soundararajan, H. Ahmed, and T. Skjellum, "Negative Perceptions About the Applicability of Source-to-Source Compilers in HPC: A Literature Review," in *High Performance Computing*, H. Jagode, H. Anzt, H. Ltaief, and P. Luszczek, Eds., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 233–246. doi: 10.1007/978-3-030-90539-2_16.

[43] Y. M. Al-Hamzi and E. Al, "Neural Network-based Framework for understanding Machine deep learning systems' open issues and future trends: A systematic literature review,"

Turkish Journal of Computer and Mathematics Education (TURCOMAT), vol. 12, no. 12, Art. no. 12, May 2021.

[44] P. Robe, S. K. Kuttal, J. AuBuchon, and J. Hart, "Pair programming conversations with agents vs. developers: challenges and opportunities for SE community," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 319–331. doi: 10.1145/3540250.3549127.

[45] L. Yan *et al.*, "Practical and Ethical Challenges of Large Language Models in Education: A Systematic Scoping Review," no. arXiv:2303.13379. arXiv, Jul. 22, 2023. doi: 10.48550/arXiv.2303.13379.

[46] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, in SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 500–506. doi: 10.1145/3545945.3569759.

[47] C. Yang, Y. Liu, and C. Yin, "Recent Advances in Intelligent Source Code Generation: A Survey on Natural Language Based Studies," *Entropy*, vol. 23, no. 9, Art. no. 9, Sep. 2021, doi: 10.3390/e23091174.

[48] G. Novakova Nedeltcheva, A. De La Fuente Ruiz, L. Orue-Echevarria Arrieta, N. Bat, and L. Blasi, "Towards Supporting the Generation of Infrastructure as Code Through Modelling Approaches - Systematic Literature Review," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2022, pp. 210–217. doi: 10.1109/ICSA-C54293.2022.00048.

[49] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Model: Survey, Landscape, and Vision." arXiv, Jul. 14, 2023. Accessed: Oct. 23, 2023. [Online]. Available: http://arxiv.org/abs/2307.07221

[50] F. Alexander, E. A. Abdiwijaya, F. Pherry, A. A. S. Gunawan, and Anderies, "Systematic Literature Review on Solving Competitive Programming Problem with Artificial Intelligence (AI)," in *2022 1st International Conference on Software Engineering and Information Technology (ICoSEIT)*, Nov. 2022, pp. 85–90. doi: 10.1109/ICoSEIT55604.2022.10029949.

[51] A. Fontes and G. Gay, "The Integration of Machine Learning into Automated Test Generation: A Systematic Mapping Study." arXiv, Apr. 15, 2023. Accessed: Aug. 25, 2023. [Online]. Available: http://arxiv.org/abs/2206.10210

[52] H. Doi, T. Osawa, and N. Tsutsumida, "The role of large language models in ecology and biodiversity conservation: Opportunities and Challenges," Preprints, preprint, Jun. 2023. doi: 10.22541/au.168657324.49460085/v1.

[53] M. Sallam, "The Utility of ChatGPT as an Example of Large Language Models in Healthcare Education, Research and Practice: Systematic Review on the Future Perspectives and Potential Limitations." medRxiv, p. 2023.02.19.23286155, Feb. 21, 2023. doi: 10.1101/2023.02.19.23286155.

[54] C. Lee, J. Gottschlich, and D. Roth, "Toward Code Generation: A Survey and Lessons from Semantic Parsing." arXiv, Apr. 26, 2021. doi: 10.48550/arXiv.2105.03317.

[55] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Model: Survey, Landscape, and Vision," *arXiv.org*, Jul. 14, 2023. https://arxiv.org/abs/2307.07221 (accessed Oct. 19, 2023).

[56] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and M. C. Desmarais, "Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing," *arXiv (Cornell University)*, Aug. 2023, doi: https://doi.org/10.48550/arxiv.2308.16557.

[57] M. T. Ribeiro, T. Wu, C. Guestrin, and S. Singh, "Beyond Accuracy: Behavioral Testing of NLP Models with CheckList," *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, doi: *https://doi.org/10.18653/v1/2020.acl-main.442*.

[58] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, Feb. 2018, doi: *https://doi.org/10.1109/tse.2017.2663435*.

Appendixes

Appendix A – Individual contribution

194176H – Vinoth V.

In the challenging module of Complex Java Code Generation using Llama, my individual contribution lies in streamlining the intricate process of translating complex textual code into simplified tasks. I pioneered the use of a small llama model, which is trained specifically to break down elaborate code into manageable modules. Before the code undergoes the generation phase, it is internally decomposed into task-specific cores using another trained model.

Each module is then processed through pre-trained Java language generation models, such as GPT, to produce the necessary programs. My unique contribution extends to the development of an algorithm that efficiently reconnects these generated modules to reconstruct the original complex code seamlessly.

Furthermore, I actively engaged in implementing the foundational aspects of this approach in Colab, ensuring a smooth integration of the entire system. This innovative process revolutionizes the way we approach complex code generation, making it more efficient and comprehensible.

194176H – Ajeevitha S.

In the challenging realm of Test Case Generation for Java using advanced language models (LLMs), my key role centers around optimizing the intricate process of transforming descriptive method requirements into executable test cases. I spearheaded the implementation of a specialized LLM, fine-tuned to parse complex method descriptions and distill them into concise test case prototypes. Prior to the generation phase, I devised a methodology to segment these descriptions into task-specific components using another trained model.

Each segmented task undergoes transformation through pre-trained Java language generation models, leveraging tools like GPT, to meticulously craft the requisite test cases. A significant

innovation I introduced involves the development of an algorithm that seamlessly integrates these generated test cases, ensuring they accurately reflect the intended method functionalities.

Moreover, I played a pivotal role in establishing the foundational framework for this approach within Google Colab, facilitating the seamless integration and validation of the entire test case generation system. This systematic approach not only enhances efficiency but also revolutionizes the way we approach the critical task of generating and validating test cases for complex Java code.

194061C – Hirushanan S.

In the realm of complex Java code optimization, I embraced the challenge of streamlining the intricate translation process with the innovative use of a Llama model. This model, specifically trained to dissect the functionalities of complex code, played a pivotal role in my contributions. By meticulously evaluating code segments with evaluation metrics and assigning scores, I prioritized samples and designated optimized code with minimized scores. This approach led to the creation of efficient and streamlined code structures.

Beyond optimization, I ventured into uncharted territory by devising a unique evaluation method for the optimized code itself. This involved delving into the depths of software engineering metrics and RLHF cycles, ensuring the quality and efficacy of the transformed code. To solidify the practical implementation of these advancements, I actively engaged in constructing the foundational framework within Colab, paving the way for future development and utilization. Prior to this, my efforts in filtering a superior dataset for model training and crafting Colab codes for metric calculation contributed significantly to establishing a robust foundation for this project.