Retrieval-Augmented Large Code Generation and Evaluation using Large Language Models

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Bhushan Deshpande

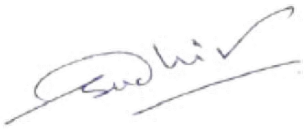
Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2025

Supervisor: Mr. Sudhir Kumar

Certificate

This is to certify that this dissertation entitled **Retrieval-Augmented Large Code Generation and Evaluation using Large Language Models** towards the partial fulfillment of the BS-MS dual degree program at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Bhushan Deshpande at Coriolis Technologies Pvt. Ltd., Pune, under the supervision of Mr. Sudhir Kumar, Member of Technical Staff, Coriolis Technologies Pvt. Ltd., Pune, during the academic year 2024-2025.



Mr. Sudhir Kumar

Committee:

Mr. Sudhir Kumar

Prof. Leelavati Narlikar

This thesis is dedicated to my mother, Manisha, my father, Mahesh, my younger brother, Vaibhav, and the countless helping hands whose support has made this journey possible.

Declaration

I hereby declare that the matter embodied in the report entitled **Retrieval-Augmented Large Code Generation and Evaluation using Large Language Models** are the results of the work carried out by me at the Coriolis Technologies Pvt. Ltd., Pune under the supervision of Mr. Sudhir Kumar, Member of Technical Staff, Coriolis Technologies Pvt. Ltd., Pune and the same has not been submitted elsewhere for any other degree.

Bhushan Deshpande

Acknowledgments

I would like to express my gratitude to Mr. Sudhir Kumar from Coriolis Technologies Pvt. Ltd., Pune, for providing me with an invaluable opportunity to work under his supervision. I am also thankful to Mr. Rohan Nandode from Coriolis for always providing critical feedback and asking intuitive questions, which played a crucial role in achieving the goals of this thesis. Both Mr. Sudhir and Mr. Rohan guided me through every step of this process, from conceptualizing the idea to providing direction at each stage. Mr. Sudhir encouraged me, especially during times when we struggled to achieve satisfactory results, and kept me motivated throughout the entire journey.

I cannot forget to thank Mr. Basant Ranjan, CEO of Coriolis Technology Pvt. Ltd., for cultivating an open and friendly work environment. This made my time at Coriolis truly enjoyable and comfortable.

I am grateful to my teammates from the AI-ML team, namely Akshunya, Isha, and Surbhi, who helped me solve technical problems during the initial phase of this project. Rajneesh helped me understand the core problem we were trying to solve in the early stages. Later, Abhishek and Saksham became a guide in solving technical challenges. Abhishek always asked, 'How is your work going on? And Is everything Okay?. I enjoy afternoon coffee with him. A special thanks to Sagar Yadav for deploying the app and to the entire team who worked tirelessly to take my research forward, transforming it into a ready-to-use application for engineers across different teams at Coriolis. I am grateful to my batchmate and teammate, Barish, for his support during this thesis, from reviewing outputs and explaining code to designing new experiments.

I would also like to thank Mr. Sagar Jagatap, Mr. Shreyashkumar Sharma, Mr. Rajanish Khushwaha, and Mr. Pranav Thube, who, apart from their busy schedule, served as human

experts in reviewing and assessing the quality of the code generated during this project. Their feedback helped in designing prompts, validating results, and improving the quality and accuracy of the generated code. The members of Coriolis are not just colleagues; they are true teammates—always supportive, encouraging, and deeply passionate about their work. I learnt a lot of things during my tenure at Coriolis.

I am very grateful to Prof. Leelavati Narlikar, Associate Professor and Deputy Chair, Data Science, IISER Pune, for being my thesis expert. She recognized my potential early on and gave me my first-semester project under her guidance during my fifth semester. Her kindness, patience, and support helped me understand new concepts, both during my semester project and throughout my coursework. I also want to thank Prof. Joy Monteiro, Assistant Professor in the Earth and Climate Science department, under whom I worked for a year on a semester project. The concepts I learned during that time laid a foundation for my MS thesis work.

Finally, I am grateful to the Data Science Department at IISER Pune for granting me permission to work on my MS thesis. I feel privileged to be a student at an institution like IISER Pune and am thankful for my friends, family, mentors, and teachers, all of whom contributed to making this journey a fulfilling experience.

Personal Acknowledgments

My journey began in a small village called Rahuri. Moving to Pune was a complete shift for me. Throughout this journey, my family, friends, teachers, and many kind-hearted people have helped me a lot.

In my early life, my mother, Manisha, taught me strong morals, while my sister, Vijeta, taught me invaluable lessons. I grew up looking at the hard work of my father, Mahesh, and my mother. My younger brother, Vaibhav, has always stood by my side and supported me in difficult times. His encouragement I can't express in words. My family has done far more for me than I could ever deserve, and mere gratitude will never be enough to repay them.

My 10th-grade mathematics teacher, Ms. Manisha Kulkarni, is like a second mother to me. She played a significant role in shaping my life. She is a symbol of kindness and love. Whatever teaching skills I have today, I owe it to her. "Thank you" feels inadequate to express the depth of my gratitude for her.

Throughout my academic journey, I received financial support from several organizations, including the Maharashtra Times, Seva Sahayog Foundation, Infosys Foundation, and many generous donors. I am truly grateful for their assistance. I was also awarded the INSPIRE scholarship from the Department of Science and Technology (DST) and received financial aid from IISER Pune. Their support has been invaluable, and I sincerely appreciate it.

My dear friend, Maitreyee, provided me with financial assistance multiple times during my time at IISER. However, her contribution was not limited to just that. She stood by me through every high and low, offered unwavering emotional support, and always listened when I needed someone. I owe her a debt that I may never be able to repay. I cannot even imagine my life at IISER without her. Her family, too, has been incredibly generous.

"Thank you" does not do justice to my gratitude for them.

Among my school friends, Pankaj and Prerna have always believed in me. Pankaj is no less than a brother to me, while Prerna was there during my time of confusion. Prerna and Vijeta guided me when I didn't know what to do. From IISER, Rajat, Pratiksha, and Vishakha were my fun partners and constant supporters who helped me stay on track. My junior, Aakash, a multitalented individual, always motivated me to maintain good health and pursue my passion for art. I am immensely grateful to all of them who made this journey enjoyable and living.

Abstract

Large Language Models (LLMs) have been shown to capture the syntax, semantics, and structure of programming languages, enabling the generation of accurate code for similar test cases through Few Shot Learning (FSL) and prompt engineering. Although LLMs perform exceptionally well with small context-length inputs, they struggle to produce accurate results with large context-length inputs and out-of-distribution-dataset, thereby limiting their applicability for large-scale code generation tasks. Our work focuses on large code generation based on the custom dataset using LLMs. This research explored a number of small-sized, open-source, state-of-the-art LLMs and various configurations of LLMs, with temperature and other hyper-parameter settings. With pre-trained LLMs, code generation accuracy was less than **20%** without tuning any hyper-parameters. By implementing **Retrieval-Augmented Generation (RAG)** to retrieve contextually relevant examples, the initial accuracy of the generated code was improved, reaching **65% to 70%** correctness based on expert evaluations. A framework for reviewing the generated code called **'LLM Judge'**, was developed to identify correctness, issues, and places of improvement. By iteratively generating and refining code based on feedback from the 'LLM Judge', the accuracy of the generated code improved to **75%–80%** at the end of the second iteration. These results highlight the potential of LLM to automate the test code generation. This work reduces the time required to write custom code to automate test cases from, on average, two days to a few hours, thereby simplifying the development process for engineers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automating product test cases is essential in modern software development because it helps to check if everything works correctly and reduces manual effort. Software testing is necessary to make sure the product is stable and secure. It also imagines the edge-case scenarios before the release of software and handles those, making it more robust and thereby improving performance (Wang et al., 2024). However, manually writing each test case and its code is time-consuming. It requires expert knowledge and a lot of human hours based on the complexity of the test case, as well as the complexity of the product. Therefore, writing code for such test cases, especially when dealing with unique libraries and complex requirements, is a bottleneck for the software development process. This increases the time and cost of development of software (Sherifi et al., 2024). Large Language Models (LLMs) (Vaswani et al., 2017) have shown potential in automating various software development tasks. However, existing LLMs are trained on open-source data (Jiang et al., 2024) and are not well suited for generating accurate code for domain-specific private software or datasets. Additionally, LLMs perform well in generating short-length code blocks with high accuracy, but their performance declines sharply when generating large and complex codes. This research aims to use the ability of LLMs to learn through Few-Shot Learning and Retrieval Augmented Generation (RAG) and utilize contextual information from similar test cases to generate code for test cases of proprietary software. Our goal is to achieve an accuracy of generated code 80% or higher based on expert verification.

1.1 Large Language Models for Code Generation: An Overview

Previously, there have been numerous try at understanding and modeling language. Research in this field started with early statistical models such as N-grams (Shannon, 1948) and Hidden Markov Models(HMMs) (Rabiner, 1989). Later, advanced natural language modeling using Feedforward Neural Language Model (Bengio et al., 2003) and Word Embeddings (Word2Vec) (Mikolov et al., 2013) has been done, which improved word representation learning. Recurrent Neural Networks (RNNs) (Elman, 1990) and Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) were developed to handle sequential dependencies in text. From 2014 to 2017, there was a rise in sequence-to-sequence models such as Seq2Seq with Attention (Bahdanau et al., 2014) and Neural Machine Translation (NMT) (Sutskever et al., 2014). However, these models had difficulties with capturing long-range dependencies and required huge computational power. The paper "Attention Is All You Need" introduces the transformer architecture, which introduced self-attention, which solved the problem of parallel text processing and effectively handling long-range dependencies (Vaswani et al., 2017).

Transformers established the foundation for modern LLMs such as BERT (Devlin et al., 2019) and LlaMa (Touvron et al., 2023), which later transformed code generation tasks with models like CodeLlama (Kumar et al., 2023). Three main types of transformer architectures exist as described in the table 1.1.

Type	**Description**	**Example**
Encoder-Only	Processes text bi-directionally for understanding tasks like text classification, Named Entity Recognition (NER), and Question Answering (QA).	BERT (Devlin et al., 2019)
Decoder-Only	Uses causal (unidirectional) attention for text generation tasks like text completion and code generation.	GPT-4 (OpenAI, 2023)
Encoder-Decoder	Uses an encoder for input understanding and a decoder for output generation, applied in sequence-to-sequence tasks such as machine translation and summarization.	T5 (Mastropaolo et al., 2021)

Table 1.1: Different types of transformer architectures

Models used for code generation are built on Decoder-Only Transformer architecture. These models are trained on vast datasets, which enables them to learn the syntax (structure of sentences), semantics (meaning of words and phrases), and the underlying patterns in the data (Vaswani et al., 2017). Models like **CodeLlaMa** (Kumar et al., 2023), **LlaMa3.1** (Meta, 2024; Touvron et al., 2023), and **Phi4** (Abdin et al., 2024) are also trained on diverse coding datasets that include various programming languages. The data for training these models is taken from GitHub Repositories, Stack Overflow and Code Discussions, Open-Source Projects, Documentation, and API References. Each model is trained on a wide range of programming languages, such as Python, C, C++, Java, HTML, and so on, covering 20 to 30 programming languages. Along with the multiple languages, there are models like **CodeLlama-Python**, which specializes in Python, and **CodeLlama-Instruct**, which is fine-tuned with human instructions and uses self-instructed code synthesis data.

Models mentioned above are trained for multiple coding tasks such as converting natural language text to code, code completion, code review, filling in the middle, etc. Despite these capabilities, these models perform poorly on custom and domain-specific data. They are not perfect and can struggle with complex and context-specific scenarios.

1.2 The Challenges in Large Custom Test Code Generation

1.2.1 Human Level Challenges

Writing a code for the specific test case is tedious and time-consuming. The time required to write the code for test cases in our dataset varies based on the product's and test case's complexity. Based on the complexities, an average complex test case may take one to two days for an experienced engineer. For less experienced engineers, it can be higher than this. This statistic applies to pre-existing test cases, but when users interact with the product, new test scenarios that were not previously planned may arise. Therefore, to write efficient code for test cases, one should have in-depth knowledge of the product, suggesting the need for an expert in the given field. During the product's initial release, the number of test cases are less. But as the features and complexity increase, the number of test cases also

increases, making it hard to write test code for each test case and maintain it manually (Wang et al., 2024). Also, it is necessary to remember the complex product details, such as names of classes, functions, methods, and APIs, along with their arguments, making the process prone to human error (Thant and Tin, 2023). As development progresses, new functions and libraries get added, and test cases are updated, requiring continuous human efforts (Tripathy and Naik, 2014). Many organizations do not have enough experienced engineers to handle the complexity of test case generation. When new engineers start working on such complex software, they need to go through all the details, which is a time-consuming and tedious process. These problems make it hard to test the specific product within the given timeline and can delay delivering the software.

1.2.2 LLM Related Challenges

LLMs are trained on open-source data. Specifically for the coding task, models such as **CodeLlaMa**, **LlaMa3** (Meta, 2024), etc., are trained using datasets like The Stack, CodeSearchNet, GitHub Code, and BigCode. While generating code for the custom test cases designed to test private software, even models like **GPT** (OpenAI, 2023) fail to generate accurate code. The reason behind their failure is a lack of software knowledge. When generating code for out-of-distribution data or product-specific data, the models mentioned above generate code using widely used libraries and give just names of the functions and general structures. However, when software is written, engineers often define their own libraries, methods, and functions whose understanding, knowledge, and context LLMs lack.

Another challenge with LLMs is hallucination. Due to the absence of context, models generate incorrect or illogical code (Ji et al., 2023). Even when provided with similar test cases as references, the output can be unreliable. The phenomenon of hallucination is well documented in the literature. In the case of code generation, there are 19 different types of hallucination are studied. This includes starting and ending in between, overall and local semantic conflicts, inaccurate expression, inaccurate conditions, logical errors, directly copying input, repetition in generated code, defining the function and not using it in the rest of the code, using functions or methods which are not pre-defined, etc. and combination of more than one of these (Liu et al., 2024). These issues make it hard to generate code because minor hallucinations can lead to the failure of the test case.

Every LLM has a parameter called the context window, which defines the number of

input tokens the model can process in a single pass. Open-source models have varying context window sizes. For example, **CodeLlaMa** supports 8,000 tokens, **LLaMA2** and **Phi-4** allow 16,000 tokens, while **LlaMa3** extends up to 128,000 tokens. This limits how much data can be processed in a single call to LLM. There is a restriction on how much data can be passed to small context window LLMs. Conversely, larger context models, which can accommodate large amounts of data, are constrained by inference time computational resources, meaning they can not process large data.

LLMs are extremely large in size. Due to proprietary software constraints, engineers cannot send product-related information to closed-source LLM providers, which have larger sizes and computational resources that may use input data for training. As a result, LLMs must be downloaded and run locally. However, hardware limitations make it difficult to use larger models. For instance, a 14-billion-parameter quantized model can take up to 9–10 GB of space on a machine. Running these models locally also requires high computational power, such as GPUs, which restricts the size of LLMs that can be used efficiently.

1.3 Enhancing Code Generation with Retrieval Augmented Generation (RAG)

All the LLMs, whether trained for general purposes or specialized in code generation, lack an understanding of domain-specific data. There are various methods to improve the accuracy of LLMs for domain-specific tasks, which include techniques like full parameter fine-tuning (Dodge et al., 2020), parameter-efficient fine tuning (PEFT) (Weyssow et al., 2025), hyperparameter optimization (Arora et al., 2024), in-context learning (ICL) and few-shot learning (FSL) (Brown et al., 2020). However, full parameter fine-tuning goes through the same process as pre-training, which requires good-quality data as well as a large amount of computing power (Ding et al., 2022). As new data becomes available, fine-tuning needs to be performed repeatedly. This makes fine-tuning impractical for tasks like test code generation, where features, libraries, and functions are frequently updated. Moreover, LLMs are hard to fine-tune for long-length code due to the context window problem as well as computational limitations. Due to the generalization and other issues mentioned, fine-tuning is not an efficient solution. In the case of few-shot learning, the user needs to provide a similar example to LLMs, which is hard to find manually when the dataset is large. There are algorithms like

Beam Search (BM25) for searching similar documents, but it is computationally expensive and biased towards the shorter sequence (Turnbull, 2015).

Retrieval Augmented Generation (RAG) is an approach that improves LLM response by providing additional external information that is not used for training. Instead of relying entirely on LLMs' internal knowledge, it provides LLMs with the necessary external knowledge from the database, which helps LLMs generate consistent outputs. In RAG, necessary data is first chunks into small pieces and converted into vectors. Then a query is encoded into a vector. A similarity search, like cosine similarity between the input query vector and vectors present in the database, is performed in a vector database (e.g., FAISS (Johnson et al., 2019)) to find the most relevant chunks (these chunks are stored in the form of document) based on embeddings. The retrieved documents are appended to the input query to provide more context. The LLM generates a response based on the query and the retrieved information. Retrieved factual data reduces LLM hallucination. Unlike fine-tuning, RAG does not increase the LLM model's actual size, which reduces memory footprint (Lewis et al., 2020). Even retrieved sources can be cited to make the LLM's response trustworthy. In the case of test code generation, with an updated vector database, new information of updated functions, methods, features, and other code changes can be easily incorporated into LLM's knowledge to generate query test case code (Gao et al., 2023). Thus, RAG helps LLMs generate large code more efficiently by using a custom dataset without retraining, making it a useful alternative to fine-tuning. The figure 1.1 shows the Retrieval Augmented Code Generation flow.

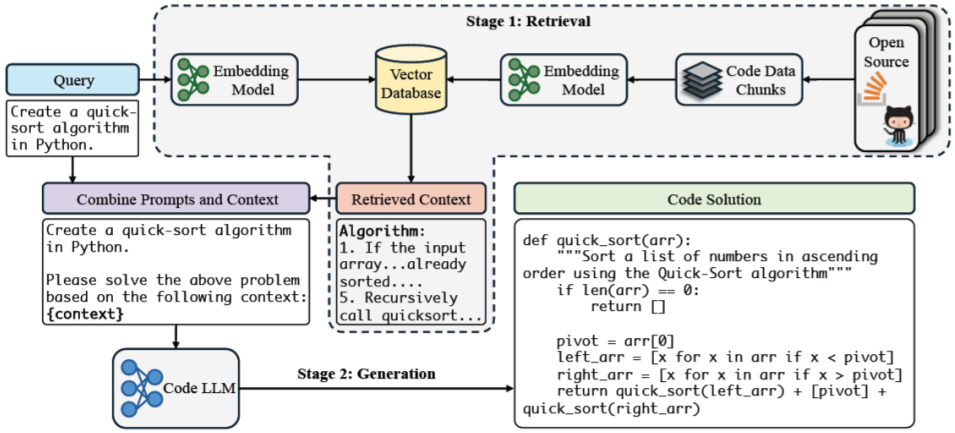


Figure 1.1: A workflow illustration of the Retrieval-Augmented Code Generation (RACG) Reproduced from the paper (Jiang et al., 2024).

1.4 Methodologies for Assessing Code Generation Quality

To track the improvement of the response generated by LLM, it is necessary to monitor its performance. Several automated metrics exist to assess text generation quality. For example, BLEU (Papineni et al., 2002), ROUGE (Lin, 2004), METEOR (Banerjee and Lavie, 2005) are commonly used matrices. These metrics focus on token overlap and n-gram matching rather than functional and logical correctness. This makes them insufficient for evaluating generated code (Evtikhiev et al., 2023a).

Evaluating RAG systems is also challenging because it involves two parts: first retrieval and second generation. Both of these parts need to be evaluated, making it more difficult. The retrieval part must find accurate and relevant information from large, constantly changing data sources. Retrieved data may be outdated or poorly quality, further complicating evaluation. On the generation side, LLMs must produce contextually appropriate responses, but measuring their correctness remains a complex task, especially for code generation tasks. Traditional evaluation methods also fail to fully capture how retrieval enhances generation. (Yu et al., 2024).

For the evaluation of LLM-generated code, metrics like CodeBLUE exist. However, there is a problem with all these metrics; they can give different scores to semantically similar generations. For example, two or more different implementations of the same code may exist, but these metrics give them very different scores. Therefore, human evaluation remains the gold standard for assessing LLM-generated output. However, human evaluation is time-consuming, especially for large codebases. Also, due to varying levels of experience, human evaluation can introduce biases (Zhao et al., 2023). Using LLMs as judges has emerged as a new approach in recent years. In this approach, LLM is used to assess the quality of output generated by other LLM with carefully designed prompts. It is possible to incorporate human values in LLM Judge through system prompt which is something traditional metrics fail to capture (Zheng et al., 2023). To ensure a thorough assessment, we employ both human evaluations, despite being time-consuming, to validate the effectiveness of LLMs as judges, alongside LLM-based evaluation itself.

1.5 Direct Preference Optimization (DPO)

To align the model output to humans, the Direct Preference Optimization technique was introduced, where the model was first fine-tuned on a dataset. Then, multiple responses generated by the model were evaluated by human reviewers who ranked them based on preference. The loss was calculated based on the discrepancy between the model's predicted probabilities for the selected and rejected responses, aligning the model's outputs with human preferences. The model was further updated based on this feedback, eliminating the need for a separate reward model and simplifying the optimization process. (Rafailov et al., 2024)

1.6 Thesis Goals

The goal of this thesis is to achieve the following points in a resource-constrained environment with limited computational power and for a very large code-length database.

- To study the effect of different hyper-parameters on the code generation and memorization effect in LLM through Few Shot Learning (FSL).
- To find the best set of hyper-parameters and best open-source, small-sized LLM for code generation task.
- To reduce hallucinations by retrieving similar context test cases using Retrieval Augmented Generation (RAG) and study the quality of retrieved-context thereby increasing the reliability and quality of code generation
- To study the different LLMs for reviewing code and implementing LLM Judge for assessing the quality of code.
- To achieve the LLM-generated code's accuracy of 80% or higher through iterative refinement of code generation and code review based on human evaluation, thereby reducing the overall time of writing the code for test cases.

1.7 Thesis Contributions

This thesis contributes the following things:

- Our research provides the best set of hyper-parameters, and sheds light on the choice of LLM for large code generation task
- Our research provides a detailed study of different open-source, small-sized LLMs for code generation and for LLM as Judge, their pros and cons for large code generation.
- Since direct human evaluation is conducted, the results are practical and directly applicable to real-world software development.
- We tested our approach for different types of test cases that are related to Windows and Linux, and our approach can be easily generalizable to other test code generation tasks.
- This approach accelerates the software testing process by significantly reducing the time required to write custom test code—from one to two days down to a few hours, consequently reducing the time required for the overall development process.

1.8 Scope and Limitations

This research focuses on generating test code for custom test cases using open-source LLMs. Our approach explores hyper-parameter tuning, prompt engineering, few-shot learning (FSL), retrieval-augmented generation (RAG), and LLM as a Judge. We developed and used LLM-as-a-judge for automatic code review and expert evaluation to evaluate the generated code. An iterative process combining code generation and LLM-based assessment was implemented to improve accuracy. In the end, we explored Direct Preference Optimization (DPO) on a small dataset, the alternative approach to Reinforcement Learning.

However, this research did not use standard metrics for code evaluation, nor did it rely on open-source datasets to benchmark the approach. During the whole thesis, we worked with small-sized LLMs due to computational requirements and did not study the very large-sized LLMs like **LlaMa3.1:70B**. Additionally, the dataset used in this study cannot be publicly released due to proprietary restrictions.

Chapter 2

Preliminary

2.1 System Configuration

The experiment in this project was performed on Linux-based virtual machines with the following configurations.

Component	**Details**
Operating System	Ubuntu 20.04.6 LTS (Focal)
Kernel Version	5.4.0-205-generic
CPU Model	AMD EPYC 7F72 24-Core Processor
CPU Architecture	x86_64 (64-bit)
Total CPU Cores	16
Threads per Core	1
Total RAM	31 GiB
Swap Memory	8 GiB (4.7 GiB used, 3.3 GiB free)
GPU Model	NVIDIA GRID A100D-40C
GPU Memory	40 GiB
NVIDIA Driver	525.105.17
CUDA Version	12.0

Table 2.1: System configuration

Python version and version of specific libraries used during this project are listed in the table below.

Package	Version
Python	3.9.5
requests	2.31.0
pandas	2.0.3
scikit-learn	1.3.0
numpy	1.25.2
langchain_community	0.3.15
langchain_core	0.3.31
FAISS (langchain_community)	0.6.3
Document (langchain_core)	0.1.14
HuggingFaceBgeEmbeddings	0.6.3
umap-learn	0.5.5
matplotlib	3.7.2

Table 2.2: Python packages used in the experiment

2.2 LLM Setup Using Ollama

To run large language models (LLMs) locally, the **Ollama** platform was used. Ollama simplifies the process of downloading, customizing, and deploying models on a local machine.

2.2.1 Installation

To install Ollama on an Ubuntu-based Linux machine, the following command was executed:

```
curl -fsSL https://ollama.com/install.sh | sh
```

2.2.2 Model Preparation

After installation, models were downloaded using the following command:

```
ollama pull <name_of_model>
```

For specific model sizes and quantization settings, detailed instructions are available at ollama.com.

2.2.3 Model Customization

To customize models by modifying hyperparameters or the system prompt, the following command was used:

```
ollama create example -f <Path to the Modelfile>
```

The `Modelfile` contains information about the desired hyperparameters and system prompt settings. Below is an example of a sample `Modelfile`:

```
FROM llama3.1:8b-instruct-q5_K_M

PARAMETER seed 42
PARAMETER temperature 0.2
PARAMETER top_k 10
PARAMETER top_p 0.2
PARAMETER repeat_last_n 0
PARAMETER num_ctx 64000

SYSTEM """You are a helpful assistant specializing in generating code."""
```

2.2.4 Model Execution

To run the model locally:

```
ollama run <name_of_model>
```

2.2.5 Model Information

To view model details such as hyperparameters, configuration, and system prompt:

```
ollama show <name_of_model>
```

Example output of the above command:

```
  Model
    architecture        llama
    parameters          8.0B
    context length      131072
    embedding length    4096
    quantization        Q5_K_M
  Parameters
    top_k            10
    top_p            0.2
    num_ctx          64000
    repeat_last_n    0
    seed             42
    stop             "<|start_header_id|>"
    stop             "<|end_header_id|>"
    stop             "<|eot_id|>"
    temperature      0.2
  System
    You are a helpful assistant specializing in generating code.
  License
    LLAMA 3.1 COMMUNITY LICENSE AGREEMENT
    Llama 3.1 Version Release Date: July 23, 2024
```

This approach ensured efficient deployment, customization, and experimentation with Large Language Models using Ollama.

2.3 Hyper-parameters

A list of hyper-parameters and their meaning is listed in the table below:

Parameter	Meaning	Value Type
seed	Sets the random number seed to use for generation. Setting this to a specific number will make the model generate the same text for the same prompt.	int
temperature	The temperature of the model. Increasing the temperature makes the model respond more creatively. In code generation, it may import new libraries, create its own method and function names, restructure the code, and sometimes stop midway.	float
top_k	Reduces the probability of generating nonsensical outputs. Top-k sampling limits the selection to the k most probable next tokens. A higher value (e.g., 100) results in more diverse outputs, while a lower value (e.g., 10) is more conservative. In code generation, it affects naming styles, library imports, and method definitions.	int
top_p	Works alongside top-k. A higher value (e.g., 0.95) encourages diversity, while a lower value (e.g., 0.5) keeps the text more controlled. In FSL, it helps reuse previously shown methods and maintains a consistent naming style and structure.	float
num_ctx	Defines the size of the context window used to generate the next token.	int
repeat_last_n	Determines how far back the model looks to prevent repetition. If enabled, the generated code avoids reusing previously used variables and methods. Disabling it causes repeated use of earlier functions, methods, and variables.	int

Table 2.3: Description of key hyper-parameters for text and code generation

Source: https://github.com/ollama/ollama/blob/main/docs/modelfile.md

2.4 Quantization in Large Language Models

Quantization is a technique used to reduce the memory footprint and computational cost of deep learning models by representing weights and activations with lower precision (e.g., 8-bit or 4-bit instead of 32-bit floating point). This optimization is important for deploying large models efficiently, especially in resource-constrained environments.

In the context of Large Language Models (LLMs), quantization enables faster inference and reduces the hardware requirements without significantly compromising performance. Various quantization methods exist. In our research, we mainly used the following set of quantization.

Quantization Type	**Memory Requirement**	**Description**
F32	100% (Baseline)	Full precision 32-bit floating point (Lossless, high memory)
FP16	∼50% of F32	16-bit floating point (Lower memory, negligible loss)
Q8	∼25% of F32	8-bit quantization (High accuracy, moderate compression)
Q5	∼15-17% of F32	5-bit quantization (Good trade-off between size and accuracy)
Q5_K_M	∼15-17% of F32	5-bit grouped quantization (Per-channel optimization, lower loss)
Q4	∼12.5% of F32	4-bit quantization (Aggressive compression, higher loss)
Q4_0	∼12.5% of F32	4-bit quantization with optimized weights (Lower loss than standard Q4)

Table 2.4: Comparison of quantization types, memory requirements, and description

Source: https://github.com/ggml-org/llama.cpp/discussions/2094?utm_source=chatgpt.com

Chapter 3

Dataset Preparation

This section provides details of the comprehensive data preparation pipeline employed in our research. A large-scale dataset was the center of this research, where most of the experimentation was done. A smaller dataset was used for targeted experimentation to check the methods applied to the smaller-scale datasets.

3.1 Larger Code Length Dataset

This research uses a large code dataset from test cases designed for an encryption and decryption system. Due to the system's proprietary nature, the dataset's specific contents cannot be directly disclosed. However, a detailed description of the data generation and preparation process is provided to ensure transparency and reproducibility.

3.1.1 Data Preparation and Cleaning:

Steps for each test case were taken from TestLink, a web-based test management system. The platform supports maintaining test cases. Data from TestLink was extracted in JSON format, with each JSON object containing information for a single test case: the test case name, a descriptive summary, a sequence of steps to be executed, and the expected output resulting from those steps. These steps are simulated using Python code. Each step corresponds to a

specific Linux command, performing operations such as file creation, copying, moving, and deletion of files and applying various encryption and decryption policies on files and folders.

The extracted JSON test case information was not easily human-readable, as many entries contained only Linux commands. A Large Language Model (LLM) was employed to translate these steps into a more natural and understandable language to improve the readability of information. This process ensured clarity of the test case information. After converting to a human-readable format, unnecessary empty lines and markers, such as / and *, were removed to create a clean and consistent data representation. This process was repeated multiple times. The resulting data was then organized into a text file. The text file contains the test case name, description of the test case, steps, and expected output. Information on each test case is separated by two empty lines in a text file. This specific separator was deliberately chosen to enable the efficient creation of a vector database, as described later in this section. Text file content formatting is shown below:

Text file content format

Name: ABCD_1234
Description: Description of the test case
Steps:
Step 1:
Expected Output:
Step 2:
Expected Output:
Step 3:
Expected Output:

Name: XYZW_1234
Description: Description of the test case
Steps:
Step 1:
Expected Output:
Step 2:
Expected Output:
and so on

In addition to the test case descriptions and steps, the dataset also had the actual Python code used to execute these test cases. The code for 430 test cases had already been written by human experts and stored in a structured directory system. Each file in the directory was named using a consistent format that matched the test case name. To include the existing code in the dataset, we conducted a thorough search to find the code for each test case in the TestLink data. For each match, a new Python file was created, named identically to the test case (e.g., if the test case were named "ABCD-1234," the corresponding Python file would be "ABCD-1234.py"), and the existing code was copied into this file. All such code files are stored in a folder. This created a direct connection between each test case description and its corresponding executable code. In the end, we had a text file containing descriptions for 420 test cases and a folder with 420 Python files, each matching a specific test case.

A vector database was built to enable semantic search and retrieval of test case information. It helps find meaningful connections between the input query (usually a set of test case steps formatted like the stored text file) and the relevant test cases. The *BAAI/bge-base-en* (Xiao et al., 2023) model was chosen for embedding because of its strong performance in encoding natural language. FAISS (Facebook AI Similarity Search) was used as the vector database for its ability to efficiently handle large datasets and perform fast similarity searches. To index the test steps from the text file, a two-empty-line separator strategy was used, ensuring that each test case was treated as a separate document in the database. This approach, combined with *BAAI/bge-base-en* embeddings and FAISS, enables effective retrieval of relevant test cases based on semantic similarity.

3.1.2 Comparison with Existing Benchmarks:

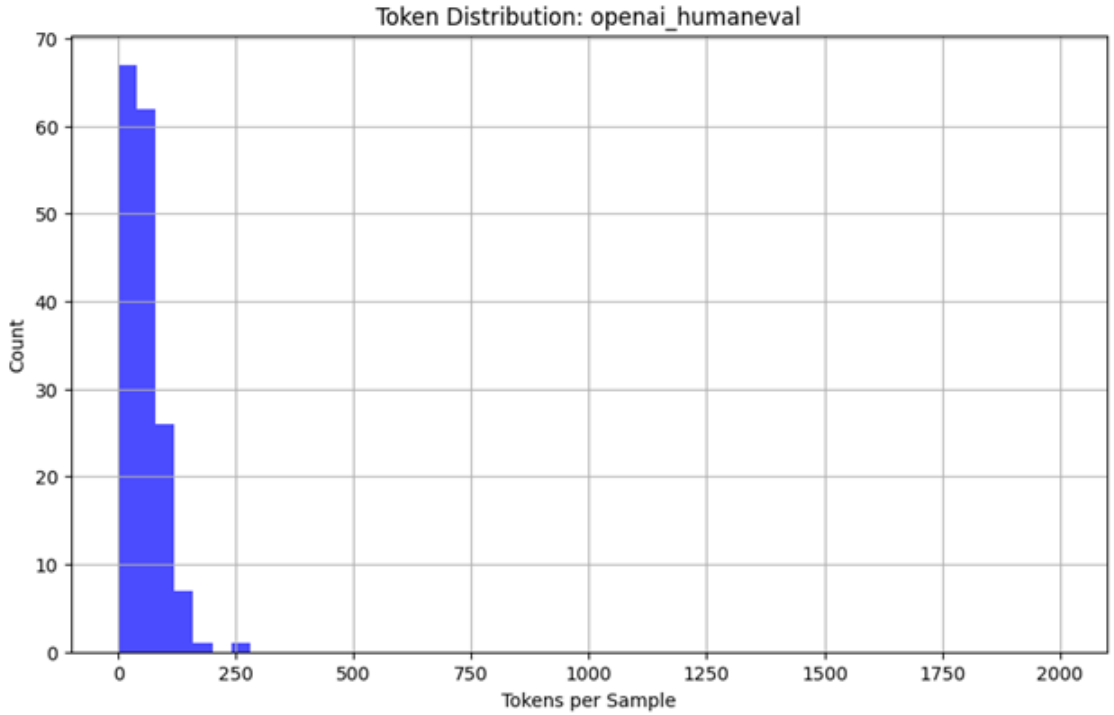
A differentiating factor of our research lies within our dataset. This section provides a comparison between existing Python benchmarks, which are used for testing LLMs, with respect to a summary of a number of tokens and a number of lines with our dataset. Table 3.1 represents the five-number summary of a number of tokens within each of the standard benchmarks. Code within each sample from benchmarks is tokenized using Tiktoken, which is an open-source tokenizer.

Dataset Name	Samples	Minimum	Maximum	Mean	Median	P25	P75
openai_humaneval	164	5	240	53.8	45	25	73
Code_X_Glue_CT_Code_To_Text (Python)	1000	27	3347	219.7	147	92	279
mbpp (Full)	374	9	255	56.7	43	30	66
code_search_net (Python)	1000	27	6063	317.8	168	91	313
iamtarun/python_code_instructions_18k_alpaca	18612	1	4012	121.1	71	42	119
Training Dataset	**447**	**406**	**8213**	**2214.6**	**1802**	**1280**	**2851**
Validation Dataset	**10**	**1188**	**4870**	**2535.0**	**1476**	**1354**	**4111**

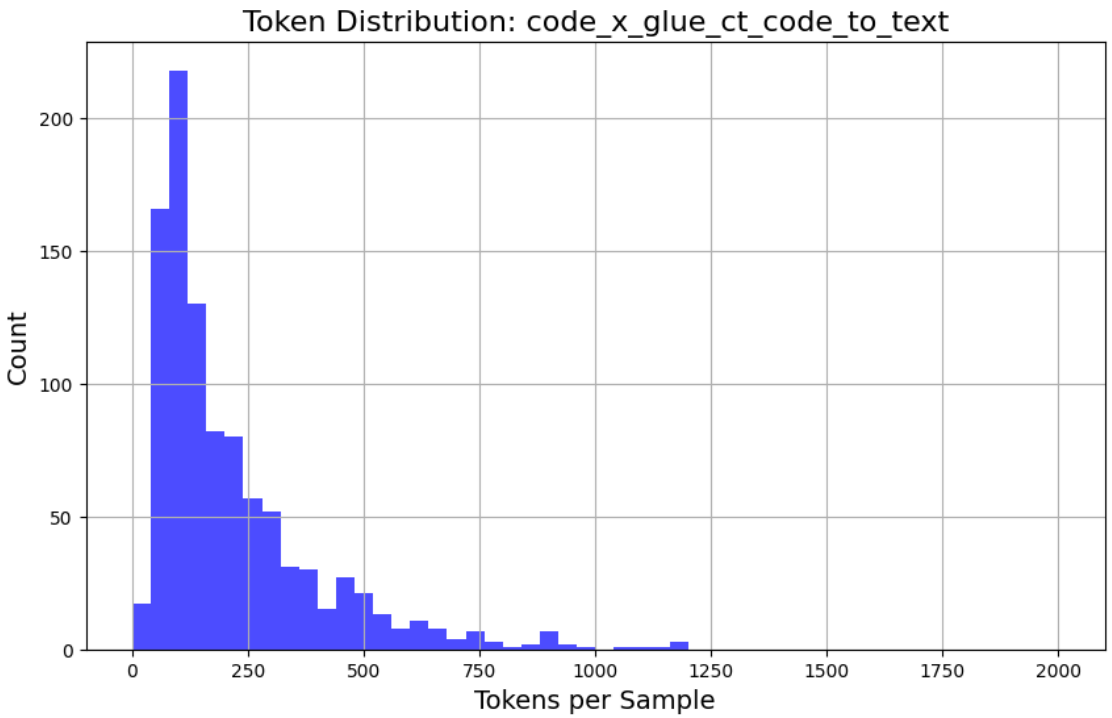
Table 3.1: Comparison of our dataset with existing Python benchmarks in terms of token statistics.

From the above table, it is evident that compared to benchmarking datasets, samples from our dataset contain a far more number of tokens, with mean and median numbers of tokens 2214.6 and 1802, respectively. In the same way, the token distribution of the validation dataset is of equivalent size to the training dataset.

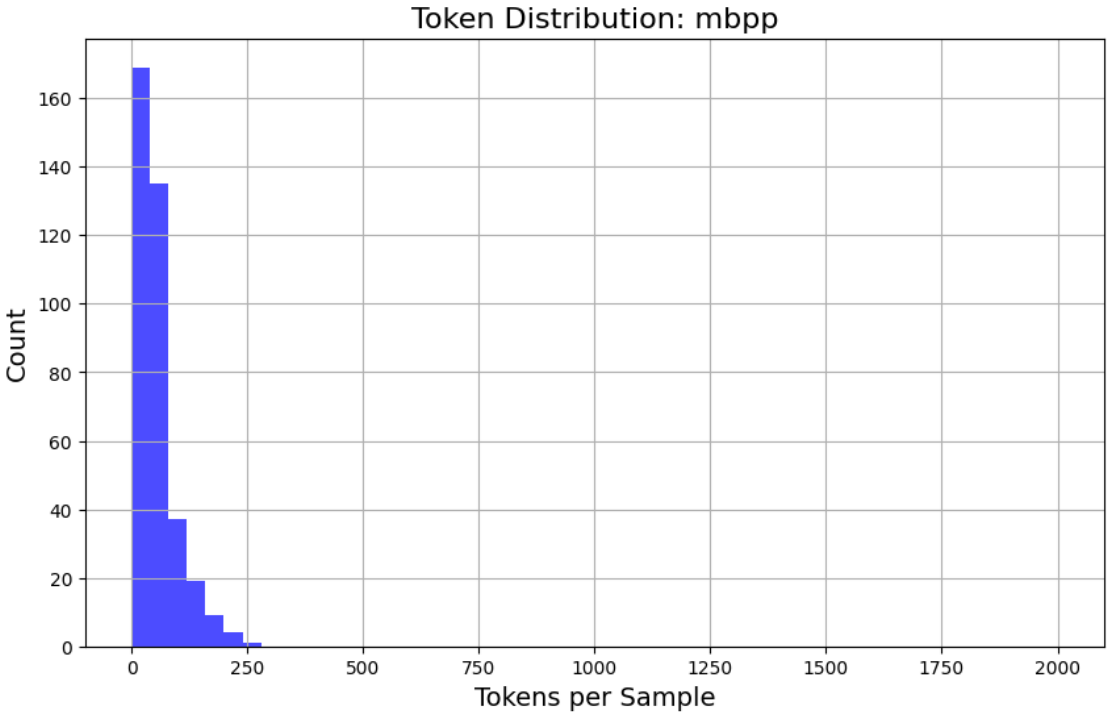
Below is the token distribution of samples from benchmarking datasets and our dataset. It is clear that the distribution of the number of tokens in each sample from these datasets is heavily right-skewed, which implies most of the samples have smaller code lengths, which is not comparable to the dataset this experiment used 3.1f.



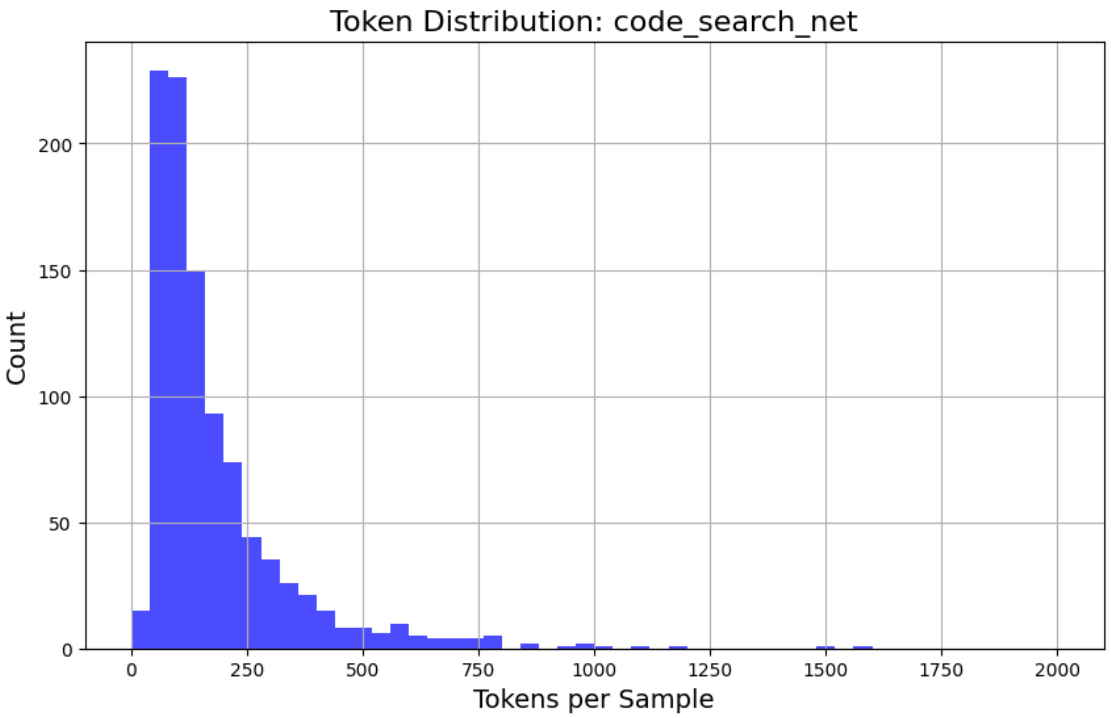
(a) OpenAI HumanEval



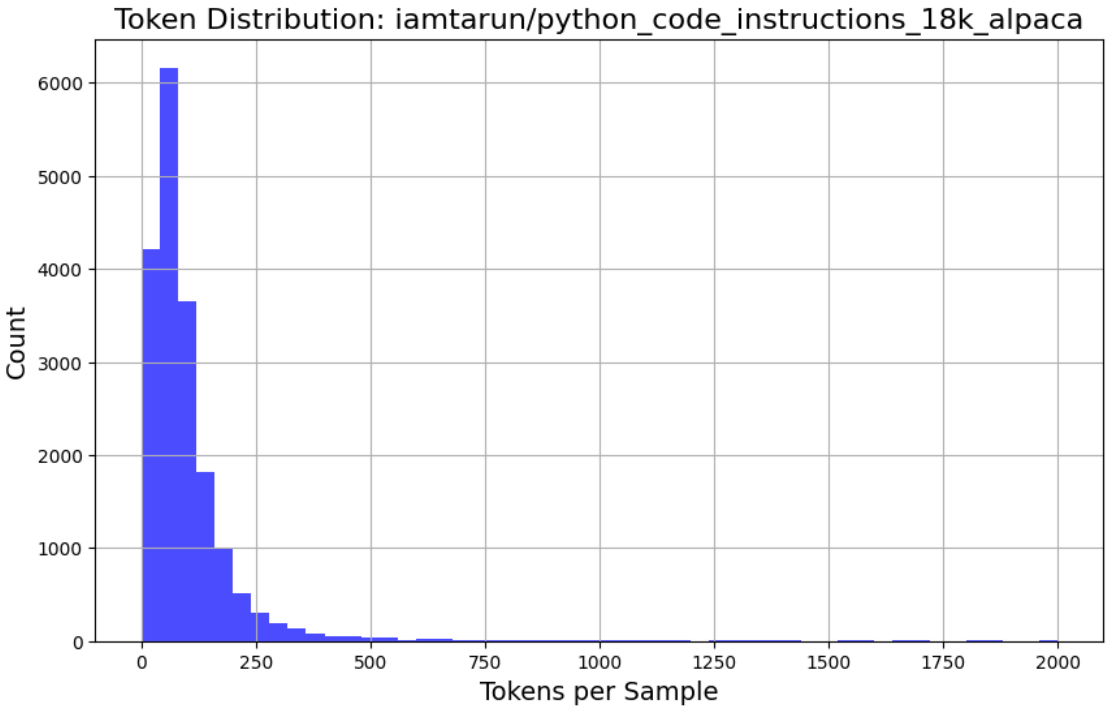
(b) CodeXGLUE Code-to-Text



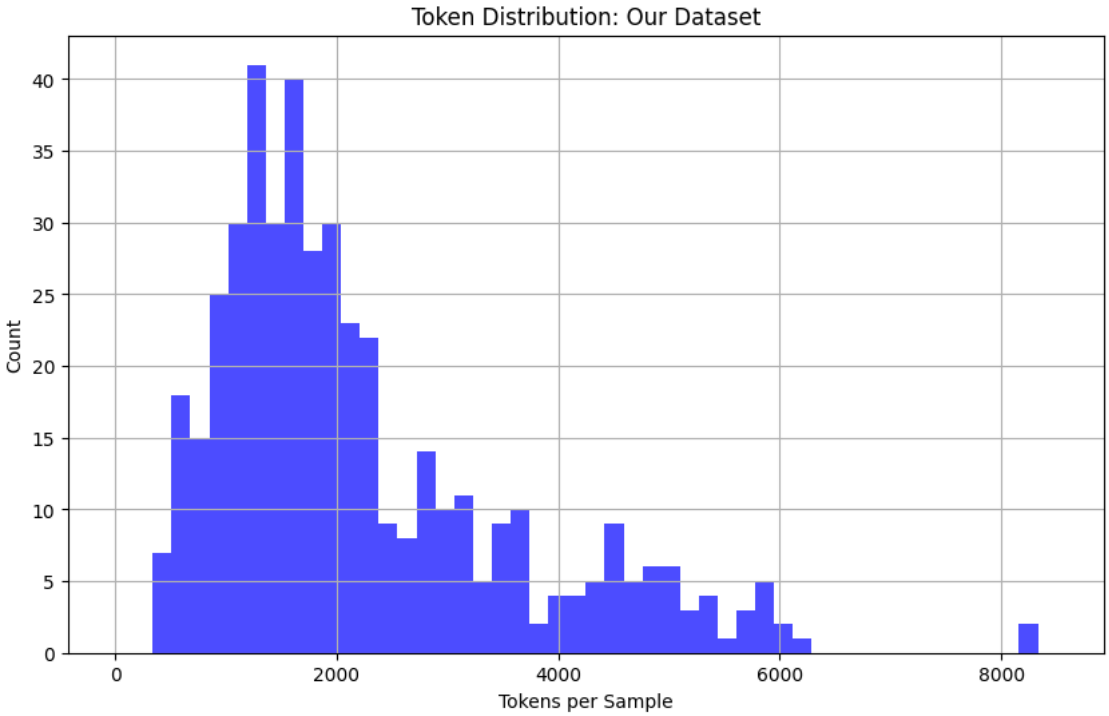
(c) MBPP (Full)



(d) Code Search Net (Python)



(e) Python Code Instructions



(f) Our Dataset

Figure 3.1: Token distribution of existing Python benchmarks in comparison with our dataset

Table 3.2 represents the five-number summary of a number of Python code lines within each of the standard benchmarking datasets and our dataset.

Dataset Name	Samples	Minimum	Maximum	Mean	Median	P25	P75
openai_humaneval	164	2	32	7.8	7	3	11
Code_X_Glue_CT_Code_To_Text (Python)	1000	3	204	26.4	19	12	34
mbpp (Full)	374	2	29	6.6	5	3	8
code_search_net (Python)	1000	3	171	22.1	16	10	26
iamtarun/python_code_instructions_18k_alpaca	18612	1	545	16.9	11	6	18
Training Dataset	**447**	**31**	**713**	**220.7**	**182**	**129**	**286**
Validation Dataset	**10**	**124**	**445**	**233.6**	**156**	**144**	**322**

Table 3.2: Comparison of our dataset with existing Python benchmarks in terms of number of Python code lines statistics.

From this table, we can see our dataset samples have an average of 220.7 Python code lines, about 14 times more than the average of the five benchmarking datasets. The median is 182 lines, approximately 15.5 times higher than the benchmark average. Similarly, our validation dataset has a minimum of 124 lines of Python code to a maximum of 445 with an average of 233.6 which is of comparable size to the training dataset

In summary, our dataset has significantly larger lines of code and higher token counts per sample than other benchmarking datasets. This difference highlights the comprehensive nature and representative resource for real-world code generation tasks compared to the benchmarking datasets. This dataset is also constantly updated with new functions, methods, and APIs, highlighting the challenges of generating product-specific code in an evolving environment, making it hard for LLM to generate custom code and represent the real-world software testing scenario.

3.2 Smaller Code Length Dataset

The smaller code datasets contain approximately 1,100 test cases for various APIs testing. Each test case includes 10–12 lines of Python code. Unlike the larger code datasets, these test cases do not have test steps available. Consequently, a vector database is created, where each test case's code is stored in a separate document. The embedding model and vector

database used are the same as those for the larger datasets. Queries are structured as JSON objects containing information such as the API endpoint, description, inputs, outputs, and status code. An example of code from the smaller dataset is given below:

Python Code Example from Smaller Code Length Dataset

```
def test_deploy_list_filter_with_scope_my(run_api, admin_exec_api,
↪  library_add_new_vm):
    """
    filter deploy list using scope = my
    """
    params, r = library_add_new_vm
    lib_id = r["uuid"]
    r = admin_exec_api.deploy_image(lib_id)
    deploy_id = r.json()["uuid"]
    count = check_count_deploylist(run_api, deploy_id, params={'scope':
    ↪  'my', 'uuid': deploy_id})

    if run_api.user_type == USER_TYPE['non_admin'] or run_api.user_type ==
    ↪  USER_TYPE['manager']:
        assert count == 0
    elif run_api.user_type == USER_TYPE['admin']:
        assert count == 1

    r = admin_exec_api.deploy_image_delete(deploy_id)
```

Chapter 4

Few Shot Learning for Code Generation

Chapters 4 to 8 will provide the details of the experiments performed and the purpose of each experiment. Each section goes into the different methods and procedures followed, limitations, and assumptions made. Sections also include information about LLMs, along with prompts and hyper-parameters used for generating results.

This chapter will provide the details of the various Few Shot Learning experiments performed and the purpose of the experiment. Detailed prompt, hyper-parameters, and information about the LLMs is provided.

This experiment was performed to find the built-in capabilities of Large Language Models (LLMs) to generalize from limited examples, understand the syntax and semantics of input code, and generate logical output. This experimentation provided the foundation for subsequent experiments and more complex methodologies, such as Retrieval Augmented Generation (RAG) and the application of iterative code refinement. The experiment was also performed to identify the optimal prompt structure, hyper-parameter configurations, LLM choice, and the relative effectiveness of two-shot versus three-shot learning. The outcomes obtained – the best-performing prompt structure, the set of optimal hyper-parameters, and the selection of the most effective LLM – directly impacted the design and implementation of subsequent experiments.

4.1 Two and Three Shot Learning

In this experiment, LLMs were provided with two and three similar examples to the query. There was a set of 10 testing queries on which this experiment was performed. As the name suggests, in Two Shot Learning approach, we provided two similar test cases to LLM and in Three Shot Learning, we provided three similar test cases. Out of 10 queries, only 6 queries had three similar examples, and the other 4 had only two similar examples. Set of queries and similar examples already created by an expert who was working on the dataset. Similar examples were found manually during this initial phase, and no other similarity search algorithm was used.

To generate the code using Two and Three Shot Learning, local LLMs were run using Ollama. Below is the format in which input is given to LLM to generate the code for query test cases. The actual prompt is given in the subsequent box.

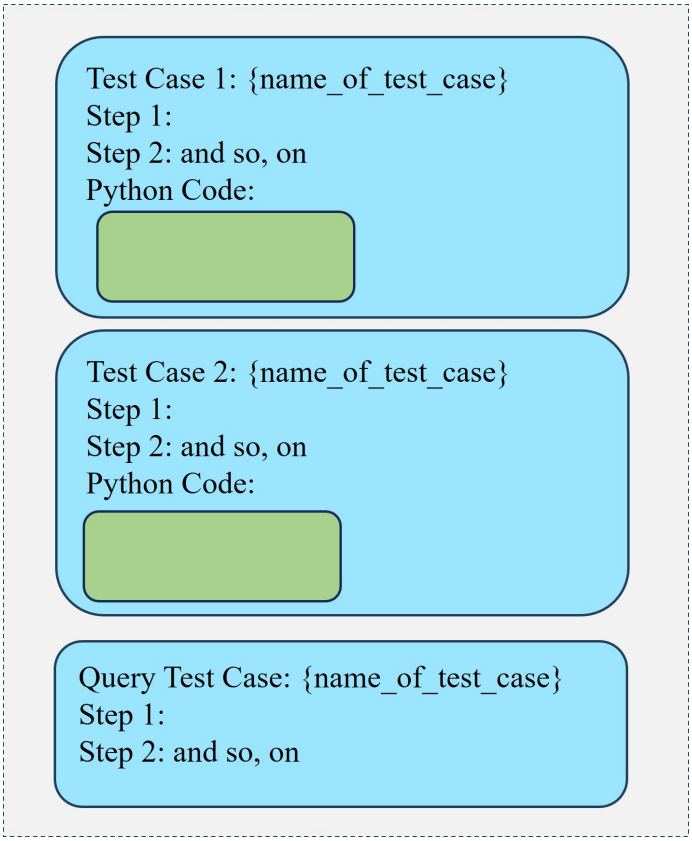


Figure 4.1: Input format to LLM for Two shot Learning

Prompt for Few Shot Learning

[INST] *Generate Python code to automate the following test case based on the pattern in the given examples.*
— Adheres to PEP 8 coding guidelines and uses snake_case for variable, function, and method names. —

Test Case:
Step 1:
Step 2:
Step 3: and so on

Python Code:
 # – Start of Python Code –
<python code>
 # – End of Python Code –

Test Case:
Step 1:
Step 2:
Step 3: and so on

Python Code:
 # – Start of Python Code –
<python code>
 # – End of Python Code –

Test Case to be Automated:
Step 1:
Step 2:
Step 3: and so on

Write a code to test the case to be automated. Do not import new libraries. Use the libraries imported in example test cases. Understand the pattern in the example test cases and try to generate similar code for the given test case.
[/INST]

4.2 Few-Shot Learning Prompts

The version of the prompt shown in the above box is just an example. Exploration of different prompts, from very basic to more complex, based on the output generated by LLMs has been done throughout various experiments. Prompts were sequentially updated based on the output generated after the use of the previous prompt. Though the basic format of similar test cases and their code is the same, the initial prompt messages were changed a lot of times until they met all the requirements in the generated output. Following are the initial prompt messages used for code generation using Few Shot Learning. These prompts were structured to ensure the generated code adhered to PEP 8 guidelines and utilized pre-existing classes and methods without unnecessary imports. We wanted to keep the prompts as simple as possible.

Once the best prompt and best set of hyper-parameters were found, a comparison between Two and Three Shot Learning was made on 6 query test cases. Expert verification of the generated code for Two and Three Shot Learning was done, and a score out of 10 was given. The scoring guide and results are included in a later chapter of this thesis.

Prompt 1

Generate Python code to automate the following test case based on the pattern in the given examples. Do not generate using `unittest`*. Import* `BaseInitializeClass` *and inherit it.*
— Adheres to PEP 8 coding guidelines and uses snake_case for variable, function, and method names. —

Prompt 2

You are a Python developer and help to generate test cases for a particular product. You have already written some code that will help generate new test cases.
Generate Python code to automate the following test case based on the pattern in the given examples. Do not generate using `unittest`*. Import* `BaseInitializeClass` *and inherit it.*
— Adheres to PEP 8 coding guidelines and uses snake_case for variable, function, and method names. —

Prompt 3

You are a Python developer and help to generate test cases for a particular product. There are a few cases given to you. Use the classes and methods from the given code.
Generate Python code to automate the following test case based on the pattern in the given examples. Do not import any other library other than those shown in the example.
— Adheres to PEP 8 coding guidelines and uses snake_case for variable, function, and method names. —

Prompt 4

Generate a complete Python code to automate the following test case to be automated based on the pattern in the given Test Case examples.
***Write each and every function of your code completely with exactly as it is done in Test Case examples and do not pass any function. Adheres to PEP 8 coding guidelines and uses* $snake_case for variable, function, and method names. **$

Prompt 4 was used for comparison between Two and Three Shot Learning using **Codellama:7b-instruct-Q5_K_M** LLM. The configuration used for these LLMs is given in the table 4.1. Detailed information about the configuration can be found in the Setup chapter.

4.3 Experimentation with different LLMs

In the beginning, experiments related to prompts and hyper-parameters were done with only one LLM. **Codellama:7b-instruct-Q5_K_M** was used for finding the best prompt and set of hyper-parameters. My colleague generated results on 10 query test cases using different LLMs while I explored the different hyper-parameters. Comparison was made between **Codellama:7b-instruct-Q5_K_M** and **LlaMa3.1:8b-instruct-Q5_K_M** to finalize the model for further experiment. During this comparison, Prompt 4 given above was used with the same set of configurations except for the context length for both models.

It was observed that for the same configuration, **llama3.1:8b** performed best among all the models studied. Also, the maximum context length of **llama3.1:8b** is 128k tokens, which means we could provide and process more context with this model.

Configuration	LlaMa3.1:8B	Codellama:7B
architecture	llama	llama
parameters	8.0B	7.0B
context length	131072	16384
embedding length	4096	4096
quantization	Q5_K_M	Q5_K_M
seed	42	42
temperature	0.2	0.2
top-p	0.2	0.2
top-k	10	10
num_ctx	32768	16384
repeat_last_n	0	0

Table 4.1: Configuration settings for llama3.1:8b and codellama:7b

4.4 Hyper-parameter Tuning

In the case of Large Language Models (LLMs) like LLaMA, and others, hyper-parameters control various aspects of code generation, model behavior, and inference. Precise tuning of inference-time hyper-parameters (like temperature, top-k, top-p, etc.) can reduce or even eliminate the need for full fine-tuning in many cases. During our experiment, we studied the behavior of the model for hyper-parameters such as temperature, top-p, top-k, repeat_last_n, and num_ctx. Meaning and how each of these hyper-parameters impacts the generated output is given in the chapter Preliminaries.

Results were generated for different temperatures, top-p, and top-k. The parameter seed was kept fixed, which produces the same output for the same input and makes the experiment reproducible. Other parameters, such as repeat_last_n, which sets how far back the model to look back to prevent repetition, were set to zero (disable), implying that the model can generate tokens repetitively and does not prevent LLM from generating the same token again and again. This is necessary in code generation because there are names of variables, functions, and methods that are called repetitively. Hyper-parameter num_ctx was set based on the maximum context window of the LLM model as well as the number of tokens that can be processed given the hardware availability.

4.5 How Models Remember Different Parts of Code

Theoretically, the self-attention mechanism in Transformers can attend to all previous tokens, meaning it has unlimited context length. However, computational constraints impose practical limits. This limitation directly leads to observations like LLMs focusing on a particular part of the input and forgetting about another part. During the study of FSL, it was observed that many times, LLM directly copied and pasted from the similar test cases provided. This can be termed as the memorization effect in LLM.

An experiment was performed to understand which part of the input, whether it is the initial part, middle part, or end part, LLM remembers better. For a given query for which code has to be generated, we had three similar test cases and their corresponding code. In this experiment, similar test cases were provided in different permutations. In each similar test case, a particular code block was marked, and we were looking for the presence of the marked code block in the code generated for the query test case. The figure given below explains the experiment described.

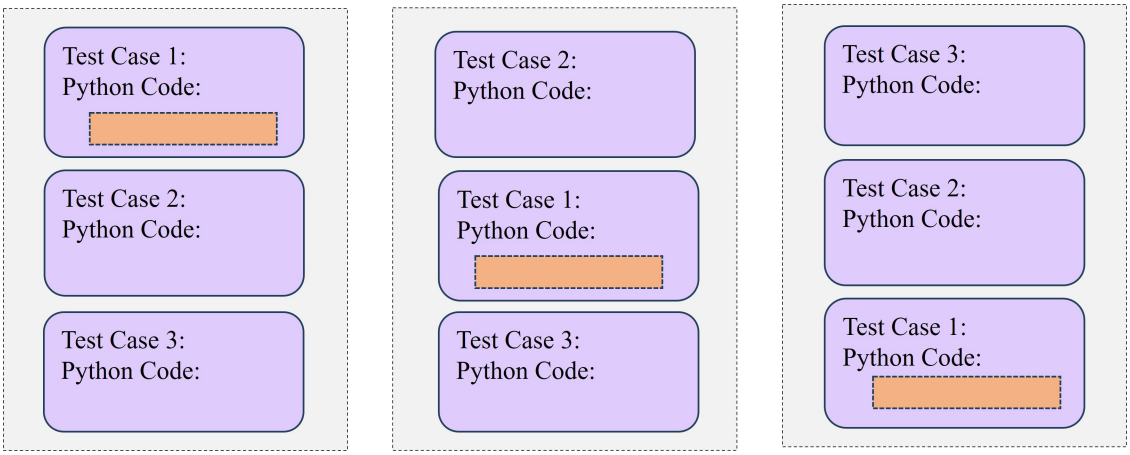


Figure 4.2: Permutation of similar test cases with a marker (orange box) to study memorization effect in LLM

This experiment was performed by providing two and three similar test cases with different combinations. For example, in the case of three similar test cases, say Test Case 1, Test Case 2, and Test Case 3, all possible permutations were provided to LLM to generate code for the query test case. Possible permutations for Three Shot Learning are as follows:

Test Cases	**Permutation**
Test Case 1, Test Case 2, Test Case 3	1
Test Case 1, Test Case 3, Test Case 2	2
Test Case 2, Test Case 1, Test Case 3	3
Test Case 2, Test Case 3, Test Case 1	4
Test Case 3, Test Case 1, Test Case 2	5
Test Case 3, Test Case 2, Test Case 1	6

Table 4.2: Permutations of test cases

This experiment was performed on one simple and one complex test case. Simple here implies each similar test case to the query has an approximate code length of 100 lines, and for complex, it's around 200-300 lines of code.

Null and Alternative Hypotheses for this experiment were:

Null Hypothesis (H_0): The output generated using similar test cases is the same despite the order in which test cases are presented to the LLM.

Alternative Hypothesis (H_1): The output generated using similar test cases is different if the order in which test cases are presented to the LLM is changed.

Chapter 5

Retrieval Augmented Generation (RAG)

LLMs have shown great capability in learning from the data, but the downside of LLMs is that they can't revise or update their memory easily. Due to a lack of domain-specific knowledge, LLMs produce hallucinations. As discussed in the introduction chapter, fine-tuning these models is not computationally efficient for constantly updating datasets. To incorporate the external knowledge base, Retrieval Augmented Generation (RAG) has emerged as a promising solution. RAG comprises two steps: the first step is retrieval, and the second step is generation. Relevant information to answer the query is fetched using the information available on the internet, PDF, excel files, and, in our case, using the code repositories. Retrieved information is then passed to LLM to generate more accurate and less hallucination-free output to the given query. This chapter covers the use of RAG to incorporate the custom code base and various steps from embedding queries and existing information to create a vector database, similarity search and code generation.

5.1 RAG Resources

Before the start of the experiment, resources required to build a vector database were created. These resources were named RAG Resources. RAG Resources contained two types of resources:

1. Text file containing information of test cases

2. A folder containing Python code for corresponding test cases

The process of data cleaning and dataset preparation is described in section 3.1.1. This section provides information about how a text file containing information on the test cases and a folder containing Python code for the corresponding test cases is created.

5.2 Embedding Model

Information about each test case was converted into a vector using an embedding model. ***'BAAI/bge-base-en'*** (Xiao et al., 2023) was used to embed the test cases information. It is a bi-encoder sentence embedding model developed by the Beijing Academy of Artificial Intelligence (BAAI). This embedding model maps any text to a 768-dimensional dense vector, which can be used for tasks like retrieval, classification, clustering, or semantic search. This model is fine-tuned on multi-domain datasets and due to its generalization capacity, it was a good choice for retrieving similar test cases to query. The HuggingFaceBgeEmbeddings class from langchain_community.embeddings is used to integrate *'BAAI/bge-base-en'* embeddings into a LangChain pipeline.

5.3 Creating Vector Database

FAISS (Facebook AI Similarity Search) vector database was used to create a vector database. FAISS is developed by Meta for efficient similarity search in dense vectors. It is used for fast retrieval in high-dimensional space. **FAISS** can handle billions of vectors efficiently with GPU support. It is useful for exact and approximate nearest neighbor searches and can be easily integrated with Hugging Face embeddings and LangChain, making it a good choice for RAG.

The text file was chunked using two empty line separators, and information about each test case was converted into the normalized vector using *'BAAI/bge-base-en'* embedding model. Since the size of the dataset is small, comprising around 450 test cases, the default

indexing 'IndexFlatL2' was used. 'IndexFlatL2' is the Euclidean distance index, which uses the exact nearest neighbor search. It calculates the Euclidean distance between vectors to find similarities between them. Due to the normalized vectors, the Euclidean distance is the same as the cosine distance. The *'BAAI/bge-base-en'* model generates 768-dimensional embeddings, represented as:

$$x = (x_1, x_2, \ldots, x_{768}), \quad y = (y_1, y_2, \ldots, y_{768})$$

L2 (Euclidean) Distance

The L2 distance between two embeddings x and y is computed as:

$$\text{L2}(x, y) = \sum_{i=1}^{768} (x_i - y_i)^2$$

Cosine Distance

When embeddings are normalized, cosine similarity is used, defined as:

$$\cos(x, y) = \frac{x \cdot y}{\|x\| \|y\|}$$

Since normalized embeddings satisfy $\|x\| = 1$ and $\|y\| = 1$, this simplifies to:

$$\cos(x, y) = \sum_{i=1}^{768} x_i y_i$$

The cosine distance is then given by:

$$\text{Cosine Distance}(x, y) = 1 - \cos(x, y)$$

5.4 Similarity Search

Query test cases, usually in a similar text format as stored in a text file, are first converted to 768-dimensional normalized vectors using the same embedding model. Distance between the query vector and vectors stored in the vector database was computed, and test cases were retrieved in ascending order of distance. Finally, top-k similar test cases to query, with minimum distance from the query vector, were retrieved. This was done by the 'similarity_search_with_score' method of the **FAISS** library, which takes the number of similar test cases the user wants to retrieve as arguments.

Various vector databases were created to retrieve the best matching test cases with the query test cases. The experiment of storing the entire code for a test case in a vector database was performed, and similarity analysis was done. There were test cases from two different test suites. Distance score analysis across different suites was carried out to get an idea of the similarity between query test cases and test cases from different suites. Manual verification was conducted to verify the similarity between the query test case and retrieved content.

5.5 Code Generation using RAG

RAG served as the core component for custom large-scale code generation by incorporating external knowledge—in our case, similar test cases along with their corresponding code. The flow of code generation using RAG is shown in the figure 5.1. A vector database was created, as explained in the above section. Steps for query test case were taken as input, which has information such as name, summary, and test steps for the test case. The query was embedded using the same embedding model. A similarity search was executed to retrieve similar test cases. Based on the name of the test case, the corresponding Python code was fetched from the folder created and appended after the query test case. Three such templates (Test case steps + Python code) combined with the initial prompt message were fed to code LLM. The code LLM used was **LlaMa3.1:8B**. Started with the best set of hyper-parameters, the best LLM model, and prompt found in FSL were employed. However, other prompts were explored, keeping the remaining things constant based on the output generated by the model.

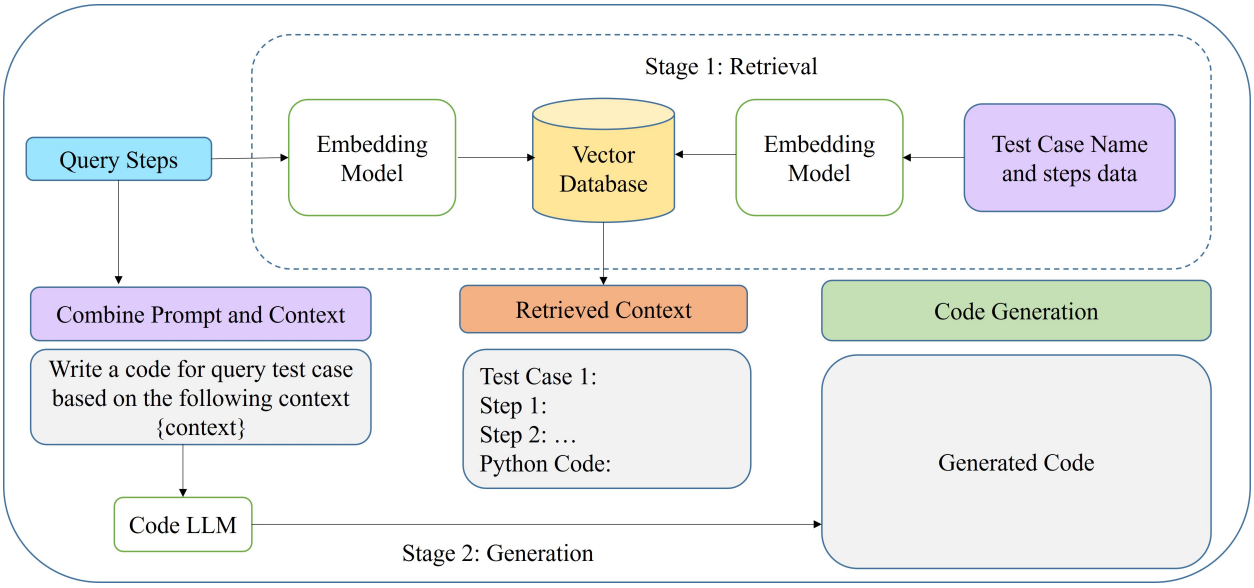


Figure 5.1: RAG flow for code generation

An example template is shown in the figure below. Three such templates were passed to the LLM. LLM used along with the configuration and set of hyper-parameters for RAG-based code generation is given in the table below.

Model Name	**LLaMA**
Architecture	LLaMA
Parameters	8.0B
Context Length	131072
Embedding Length	4096
Quantization	Q5_K_M
Repeat Last N	0
Seed	42
Temperature	0.2
Top-K	10
Top-P	0.2
Num_Ctx	64000

Table 5.1: Model details for RAG-based code generation

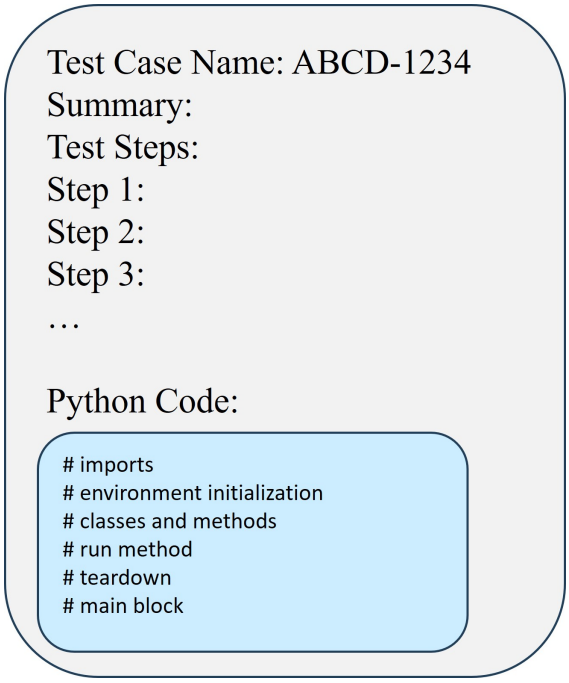


Figure 5.2: Example Template

The initial prompt message followed by three similar templates is shown in the box below:

Prompt

<|begin_of_text|><|start_header_id|>system<|end_header_id|>
You are a helpful assistant specializing in generating code. <|eot_id|>

<|start_header_id|>user<|end_header_id|> Generate a complete Python code to automate the following test case to be automated based on the pattern in the given Test Case examples.

Write each and every function of your code completely with exactly as it is done in Test Case examples and do not pass any function.
Do not directly use the functions and methods provided in the context. Instead, understand how they are used in the test cases and write all the required functions and methods explicitly to complete the steps in the test case.
Use Chain of Thought while generating code for the test case to be automated.
Adheres to PEP 8 coding guidelines and uses snake_case for variable, function, and method names.

Test Case 1: Generate a Python script based on the provided testing steps to verify the test mentioned in the summary.
Template 1

Similarly for Template 2 and Template 3

Test Case To be Automated: Generate a Python script based on the provided testing steps to verify the test mentioned in the summary.
Name:
Summary:
Test Steps:

<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Evaluation of generated code was done by human experts during the initial phase of the project. Results of evaluation and observations of distance score are included in the chapter results.

5.6 Code Generation for Smaller Code Dataset

Due to the absence of test steps and the small size of the code, direct code embeddings were stored in a vector database. Input is given in the format of the name of the API endpoint, description of the test case, expected input, and output provided to the LLM. Based on this input, six similar test codes were fetched and passed to the LLM. LLM used here was the same, **LlaMa3.1:8B**, which was used for the above case. The generated code is evaluated using the **ChatGPT** by providing context and by a human expert. The prompt used for this task is given in the box below:

Prompt for Smaller Code Generation Dataset

You are an automation engineer, good at writing test code.
Generate comprehensive test code for the user query, drawing upon the knowledge and functionality provided within a given context. Understand the context and generate code for the API specifications mentioned in the query. The provided context is the test code for an already existing API specification or it may be the content of some library file. The query is the specific REST API for which a test function is to be written.
Follow the below given guidelines:

1. Do not return results that are not in the context.
2. The library files are already written; call the library functions where needed instead of rewriting the functions.
3. Make use of docstrings that specify the function's functionality.
4. Use appropriate function names.
5. Assert the correct status code and message as given in the query.
6. Use decorators where necessary.

Think step by step and use the same coding style as that of the context.
Give me an executable test code for the following query.

Further, to align the model output to humans, the Direct Preference Optimization technique was applied where the model was first fine-tuned on a smaller dataset. To reduce the memory footprint, we generated only one response from a fine-tuned model and created a dataset with query test case information, original code, and generated code. This dataset was then passed to the DPO trainer, a hugging face class, which calculated the probability of token generation of original code and code generated by the fine-tuned model. Then, this probability gets converted into loss, and the model is fine-tuned based on human preferences.

Chapter 6

Evaluation of LLM Generated Code

We conducted two types of evaluation of LLM generated code. Started with Human Evaluation and later moved to LLM as a judge. This chapter covers the specification of Human Evaluation and implementations of LLM as a judge for review and scoring of generated code.

6.1 Expert Verification of LLM Generated Code

Existing matrices are not sufficient to capture the structural and functional correctness of generated code output. As discussed in the chapter Introduction, matrices such as BLEU, ROUGE, and METEOR rely on n-gram accuracy, recall, and precision of uni-gram and lack the syntactical and semantical accuracy of the generated code. Code-specific matrices such as CodeBLEU exist, but previous works have shown that CodeBLEU score is not reliable for functional correctness and human preferences (Evtikhiev et al., 2023b).

Due to limitations on the existing matrices, Human Evaluation remains a gold standard. However, it is time-consuming, especially in the case of long code datasets, and there exist personal biases. We tried to minimize these factors by employing-

1. Experts with similar level of experience
2. Experts who work on the same suite of test cases
3. Experts who had written code for existing test cases

Further, detailed guidelines were provided to evaluate the generated code as given in the table 6.1. The score was given on a scale of 10. Score 1 being very bad results and 9 being very good results. A score of 10 was given only when the generated code ran successfully on the system without making any changes. Based on the scoring guidelines, experts gave scores and suggested issues in the generated code. The score was not only given on the basis of a number of issues present in the generated code but also on the basis of the severity of the issues. Emphasis was given on logical and functional correctness rather than exact code match. The amount of human effort required to make the generated code fully functional was considered while giving the score.

Scoring Guidelines

Rating	
1 - 3	1. Very bad results 2. Need a lot of modification in order to run the test case 3. Model is hallucinating
4 - 5	1. Unusable code 2. Results are in the format of input test case but missing the objective 3. It will take more time to debug than writing from scratch
6 - 7	1. Ok results 2. Results are either missing one or two functions 3. Function call missing
8	1. Good results 2. Some logging errors or change of variable name 3. Objective is achieved
9 - 10	1. Very good results 2. Can be run directly on machine

Table 6.1: Code quality rating criteria

6.2 LLM as a Judge for Code Review

Though human review is the gold standard, it is a time-consuming and tedious task. To review each code, the experts needed a time of 15-20 minutes. Repetitively reviewing the same test case code for different configurations where code contains more than 150 lines of code was tiring. Due to the absence of a trusted evaluation matrix and the tiring nature of human evaluation, we decided to use LLM as a Judge to review and score LLM-generated code. The following sections provide details on how we experimented with different LLMs, what the prompt format was, what additional context was provided to the LLM as a judge, and different approaches and expert verification of issues suggested by the LLM judge.

6.2.1 Experimentation with different LLMs

This experiment leveraged different open source and small LLMs for review of generated code. **LlaMa3.8b-instruct-Q5_K_M**, from which the code was generated, was also used for the review. To keep the memory footprint less only small LLMs were tried. Different sets of prompts were tried to obtain the best possible results. We set all other hyper-parameters of LLMs to low values based on the experiment performed during FSL. A small value of hyper-parameters implies less deviation from the provided context. Context window length was either set to maximum or such that GPU usage is less than the maximum GPU compute available. LLM Judge provided a set of issues in LLM-generated code and scored out of 10 based on the scoring guide provided in the prompt. Different LLMs and configurations of hyper-parameters explored are given in the table 6.2. Output generated by LLM was carefully reviewed, and **Phi3:medium14b_Q4_0** with context length 16384 and temperature 0.2 were selected for further experimentation. One out of three issues pointed out by experts were also spotted by **Phi3:medium14b_Q4_0**. Until this point in time, the **Phi4:14b-Q4_K_M** model was not released. Later, the same experiment with **Phi4:14b-Q4_K_M** showed that two out of three issues were suggested by expert capture by this LLM. Detailed observation of experimentation with different LLMs for judge is provided in the result section.

Config. \ LLM	phi3:medium 14b_Q4_0	phi3:medium 14b_Q4_0	codellama:7b-instruct-q5_K_M	gemma2:9b Q4_0	llama3.1:8b-instruct-q5_K_M	llama3.2:3b-instruct-fp16	phi4:14b-q4_K_M
architecture	phi3	phi3	llama	gemma2	llama	llama	phi3
parameters	14.0B	14.0B	6.7B	9.2B	8.0B	3.2B	14.7B
context length	131072	131072	16384	8192	131072	131072	16384
embedding length	5120	5120	4096	3584	4096	3072	5120
quantization	Q4_0	Q4_0	Q5_K_M	Q4_0	Q5_K_M	FP16	Q4_K_M
seed	42	42	42	42	42	42	42
temperature	0.5	0.2	0.2	0.2	0.2	0.2	0.2
top_k	10	10	default	10	10	10	10
top_p	0.2	0.2	default	0.2	0.2	0.2	0.2
num_ctx	16384	8192	16384	8192	64000	64000	16000
repeat_last_n	0	0	default	0	0	0	0

Table 6.2: Different small and open source LLMs and configurations experimented for LLM as a Judge

Effect of Temperature on Score

To study how temperature affects the score given by the LLM Judge, we set all other parameters constant and generated the results for 10 test cases for 0.1, 0.2, and 0.5 temperatures. LLM used for this experiment was **Phi3:medium14b_Q4_0** with context length 16384. The same prompt was used for all temperature variations to generate the results and average across 10 test cases calculated.

6.2.2 Expert verification of issues by LLM Judge

A comparison between issues suggested by the expert and given by the LLM Judge was made to check the reliability of the score and issues given by the LLM Judge. First, generated codes were shown to the expert, and issues were noted down. Afterward, issues suggested by the LLM Judge were compared against these. To reduce human biases, issues were not directly shown to the human expert. Based on the strategy used by the human expert and scoring guide, the prompt for the LLM Judge was designed to mimic the thinking of the human expert while evaluating the generated code. Scores given by human experts and obtained by LLM Judges were compared across different test cases.

6.2.3 Scoring Guide and Prompt for LLM Judge

A similar scoring guide, as used by the human reviewers, was provided in the prompt of the LLM Judge. It was asked to provide a score out of 10 to LLM and also provide an explanation justifying the assigned score. Emphasis on functional correctness should be given rather than variable names and comments between the lines. Different prompts were tried based on the context provided to LLMs during the generated code evaluation. What context is provided was mentioned at the beginning of the prompt. The patterns, functions, and names of methods should be understood while producing the issues, and comments on generated code were told to LLM in its prompt. The use of 'Chain of thoughts' and 'think steps by step' were mentioned in the prompt to slow thinking of LLM. The prompt with the best results is provided in the next section.

6.2.4 Different Approaches used for LLM Judge

We could not provide the generated code to LLM without providing any context. Different contexts were provided as a reference to point out the issues in LLM-generated code.

During initial experimentation with the LLM Judge, the original code of the test case with generated code was passed to the LLM. LLM Judge were asked to compare the original and generated code and find the issues and improvements. This study was undertaken to understand whether LLM was useful for reviewing the code. However, while generating code for the completely new test case, there would be no original code exists. Therefore, first, LLM was provided with test steps and a description of the test case and then generated code. In the second approach, one similar test case and its code were passed to LLM. Similar test cases were found by the same similarity search approach as described in the chapter RAG. Similarly, two similar test cases feed to LLM judge to draw insight from the similar test cases and understand the patterns in naming style, function, and methods. The following box provides the prompt for the best results obtained by the LLM Judge. LLM used was **Phi4:14b-Q4_K_M**.

Though the LLM judge is not the gold standard, it was able to capture 2/3 of the issues suggested by humans. However, sometimes, it provided redundant issues regarding assertion messages and comments, which would not impact the actual flow of the code. On the

other side, LLM-Judge was able to provide issues other than human experts, which provided valuable suggestions on improving the security and efficiency of the generated code and could not be captured by the human expert.

Prompt for LLM Judge

`You are a helpful LLM as a judge for programming code evaluation assistant. Your task is to:`

1. Identify the patterns in the human-written test case steps and their corresponding code.
2. Evaluate the LLM-generated code by comparing it to the provided test steps and the patterns observed in the human-written examples.

`<|user|>`

The text contains test cases, test steps, and corresponding Python codes:

- The beginning section provides one similar test case, along with Python code written by humans.
- At the end, it includes the test steps for a new test case to be automated and the code generated by an LLM for this case.

Evaluation Guidelines:

- **Score** the generated code on a scale of 1 to 10.
- **Highlight all issues** in the generated code, including missing elements, incorrect logic, or deviations from the provided test steps.
- **Provide a justification** for your score, focusing on alignment with the test steps and identifying any missing or incorrect parts.
- **Understand the comments between the code and log messages to map steps to the corresponding code blocks.**
- **Avoid redundant comments;** if the steps are clear, corresponding code block is present, and the logic is correct, do not add comments.
- Use **Chain of Thought reasoning** to explain your analysis clearly.

Scoring Guide:

- **1-3:** Very poor results. The code has major issues or hallucinations and requires significant modifications to run the test case.
- **4-5:** Unusable code. It is somewhat related to the input steps but misses key objectives. Debugging would take more time than rewriting.
- **6-7:** Adequate results. The code is usable but needs adjustments, such as fixing incomplete functions or incorrect function calls.
- **8:** Good results. The code achieves the objective, with only minor issues (e.g., logging errors, variable name mismatches).
- **9-10:** Very good results. The code aligns well with the steps, achieves the objective, and can be executed without issues.

Ensure your evaluation includes the score, a detailed explanation, and any recommendations for improvement.

Similar Test Case: Generate a Python script based on the provided testing steps to verify the test mentioned in the summary.

Similar Test Case Python Code:

```
###--Start of python code--
<python code>
###--End of python code--
```

Test Case to be Automated:

Code Generated by LLM:

```
###--Start of python code--
<python code>
###--End of python code--
```

Chapter 7

Pipeline for Code Generation

After getting satisfactory results on code generation using RAG and finding reliable suggestions and issues on generated code by LLM Judge, this research explored the iterative process of incorporating suggestions by LLM judge for further refining and increasing the accuracy of previously generated code. This chapter focuses on the process of selecting LLM for second iteration code generation. Various combinations of LLMs and prompts were explored to improve the accuracy of the second iteration code generation. Based on the results obtained, the pipeline for iterative code generation and refinement was established.

7.1 First Iteration Code Generation

After the release of **Phi4:14B**, state of the art small sized open source model with enhanced reasoning and logic capabilities, comparisons between the **LlaMa3.1:8b** and **Phi4:14B** were made for the first iteration code generation on the validation set of 10 test cases. Due to the limitation on context length, which is 16000 for **Phi4:14B**, an experiment with two similar test cases was first done. Later, a comparison between these two models was made by passing three similar test cases to the query while generating code. Configuration with a context length equal to 32000 was also explored for **Phi4:14B** to understand the model's behavior. The comparison was made by keeping all other hyper-parameters and prompts the same except for the number of context examples and context length. **LlaMa3.1:8B** performed better in all the comparisons.

- **Inputs:** Three similar templates and test steps of a test case to be automated.
- **Output:** Code generated by LLM.

7.2 First Iteration Code Review

Once the first iteration code was generated by **LlaMa3.1:8B**. For reviewing this code, **Phi3** and **Phi4** were compared for LLM Judge. A detailed comparison of suggestions, issues, and reliability of scores was verified by human experts. Issues suggested by both models were compared against the issues pointed out by human experts. During the generation of results from LLM Judge, one similar template was provided as a reference. Other hyper-parameters and prompts were kept the same.

- **Inputs:** One similar template, test steps of a test case to be automated, code generated by LLM in the first iteration.
- **Output:** Code review with suggestions and issues along with a score out of 10.

7.3 Second Iteration Code Generation

Extensive experimentation was done for generating the second iteration of code generation. The goal was to achieve higher accuracy by incorporating suggestions given by the LLM Judge. Various combinations of LLMs were tried. For the experiments below, the first iteration code was always generated using **LlaMa3.1** because it gave the best results. Every time, while generating a review from LLM Judge, one similar template was provided as a reference.

- **Inputs:** One similar template, test steps of the test case to be automated, code generated by LLM, code review generated by LLM Judge.
- **Output:** Refine version of code with issues fixed.

Second Iteration Code Generation using LlaMa3.1

After reviewing the code by **Phi3** and **Phi4**, the second iteration codes were generated using **LlaMa3.1:8B**, and comparisons were made with the first iteration code. Whether the issues given by **Phi3** and **Phi4** were fixed or not in the second iteration was inspected. In both

cases, **LlaMa3.1:8B** was not able to incorporate the suggestions and fixed issues given by **Phi3** and **Phi4**, and therefore, the following experiments were conducted.

Second Iteration Code Generation in Same Prompt as LLM Judge

For a review of the first iteration code, **Phi3** and **Phi4** were used. In this combination, the same LLM was asked to generate code for the second iteration and fixed the issues given by the same LLM. One single prompt was written, which combined both tasks- the first iteration of code review and, and based on the suggestion, the second iteration of code generation. The process is shown in the figure 7.1.

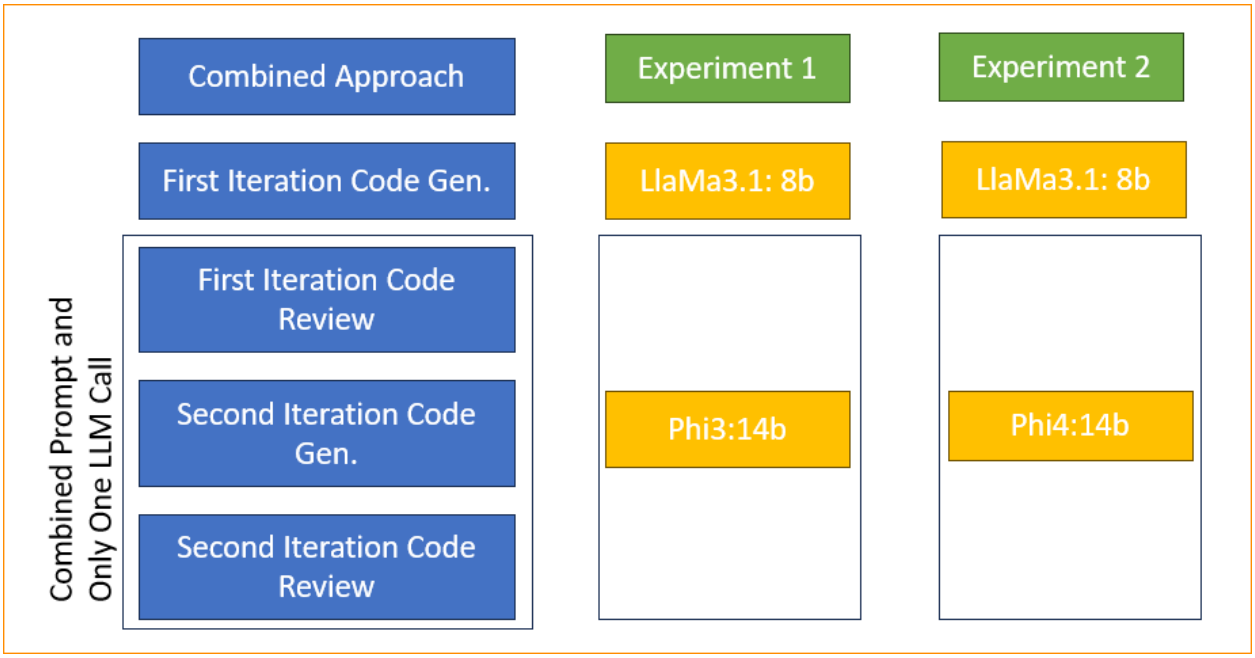


Figure 7.1: Second iteration code generation and review using same prompt and single LLM call

Second Iteration Code Generation in Different Prompt

It was observed that **Phi4:14B** was best at fixing the suggestions and issues given by the LLM Judge. Therefore, **Phi4:14B** was used for the second iteration code generation but this time using different prompts. The above process was broken into separate LLM calls using different prompts. The comparison was made between the second iteration code generation

using **LlaMa3.1:8B** and **Phi4:14B**. The process is detected in figure 7.2.

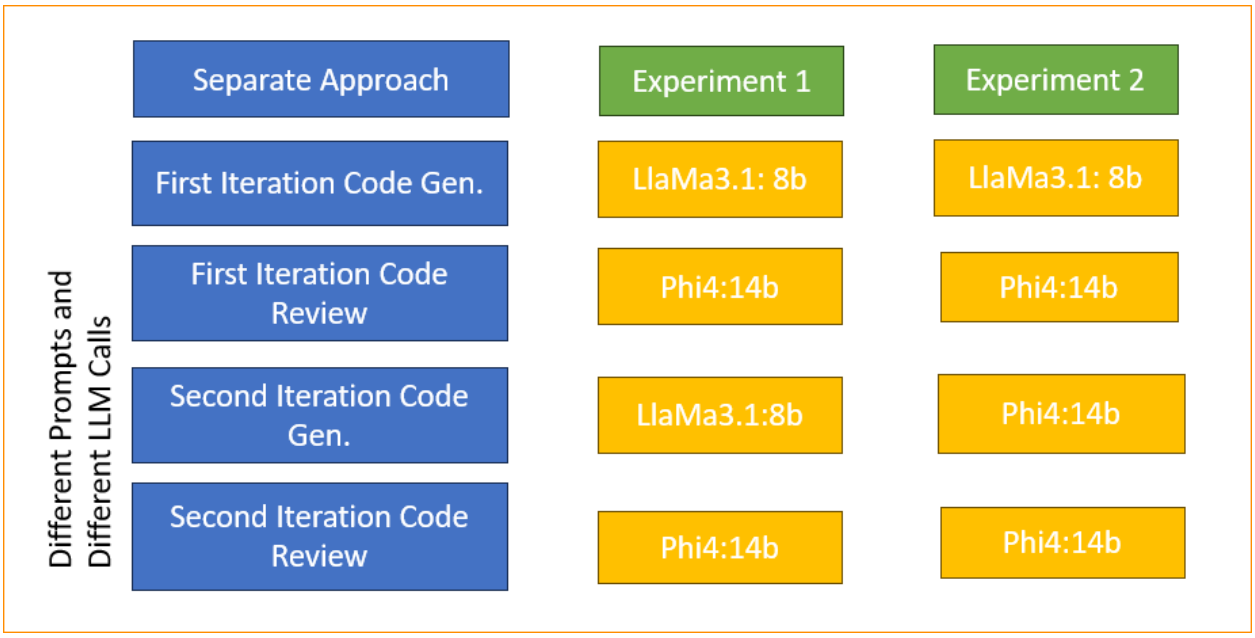


Figure 7.2: Second iteration code generation and review using different prompts and different LLM calls

7.4 Second Iteration Code Review

In continuation of the above experiment, second iteration code reviews were generated using **Phi3** as well as **Phi4** in the same prompt as the first iteration review, second iteration code generation, and second iteration code review. Also, reviews were generated using the same prompt as the first iteration code review, where one similar template and code generated in the second iteration passed.

- **Inputs:** One similar template, test steps of a test case to be automated, code generated by LLM in the second iteration.
- **Output:** Code review with suggestions and issues along with a score out of 10.

7.5 Models and Prompts

Models and Configurations

The above experiments leveraged **LlaMa3.1:8B**, **Phi3:14B**, and **Phi4:14B**. The table containing architectural information and set of hyper-parameters used for the above experiments are listed in the table 7.1. Hyper-parameter num_ctx was different for **Phi4:14B** due to its maximum content length being less, keeping all other hyper-parameters the same. The choice of hyper-parameters was made based on the experimentation done during Few Shot Learning (FSL).

Configuration	**LLaMa3.1:8B**	**Phi3:14B**	**Phi4:14B**
architecture	llama	phi3	phi3
parameters	8.0B	14.0B	14.7B
context length	131072	131072	16384
embedding length	4096	5120	5120
quantization	Q5_K_M	Q4_0	Q4_K_M
seed	42	42	42
temperature	0.2	0.2	0.2
top_p	0.2	0.2	0.2
top_k	10	10	10
num_ctx	64000	64000	16000
repeat_last_n	0	0	0

Table 7.1: Configurations of LLM used for the code generation pipeline

Prompts

The prompt used for the first iteration of code generations was the same as given in the Chapter Retrieval Augmented Generation for Code Generation. Prompts for LLM Judge used were different in combined and separate cases. For generating code in the second iteration code using **Phi4:14B**, a different prompt was used. This section provides detailed prompts for different approaches:

Combined Prompt for code review and second iteration code generation

```
System Message:
You are a helpful LLM acting as a judge for programming code evaluation and
generation assistant.  Your task is to:
```

1. Identify the patterns in the human-written test case steps and their corresponding code.
2. Evaluate the LLM-generated code by comparing it to the provided test steps and the patterns observed in the human-written examples.
3. Re-generate the entire code fixing all the identified issues.
4. Evaluate the code you generated in the same way as done in step 1 and 2.

<|user|>

The text contains test cases, test steps, and corresponding Python codes:

- The beginning section provides one similar test case, along with Python code written by humans.
- At the end, it includes the test steps for a new test case to be automated and the code generated by an LLM for this case.

Evaluation Guidelines:

- **Score** the generated code on a scale of 1 to 10.
- **Highlight all issues** in the generated code, including missing elements, incorrect logic, or deviations from the provided test steps.
- **Provide a justification** for your score, focusing on alignment with the test steps and identifying any missing or incorrect parts.
- **Understand the comments between the code and log messages to map steps to the corresponding code blocks.**
- **Avoid redundant comments;** if the steps are clear, corresponding code block is present, and the logic is correct, do not add comments.
- Use **Chain of Thought reasoning** to explain your analysis clearly.

Scoring Guide:

- **1-3:** Very poor results. The code has major issues or hallucinations and requires significant modifications to run the test case.
- **4-5:** Unusable code. It is somewhat related to the input steps but misses key objectives. Debugging would take more time than rewriting.
- **6-7:** Adequate results. The code is usable but needs adjustments, such as fixing incomplete functions or incorrect function calls.
- **8:** Good results. The code achieves the objective, with only minor issues (e.g., logging errors, variable name mismatches).
- **9-10:** Very good results. The code aligns well with the steps, achieves the objective, and can be executed without issues.

Ensure your evaluation includes the score, a detailed explanation, and any recommendations for improvement.

Task: You need to follow these three tasks one by one:

- Review the code based on the **Evaluation Guidelines** and **Scoring Guide** provided.
- Generate the code again using all suggestions from the review and considering the similar test case as references.
- Review the updated code based on the same **Evaluation Guidelines** and **Scoring Guide** provided above.

Expected Output: Review and score the provided code based on the given guidelines. Then, improve the code using the suggestions from your review. Finally, review and score the updated code again using the same guidelines.

Similar Test Case: Generate a Python script based on the provided testing steps to verify the test mentioned in the summary.
Similar Test Case Python Code:

```
###--Start of python code--
<python code>
###--End of python code--
```

Test Case to be Automated:
Code Generated by LLM:

```
###--Start of python code--
<python code>
###--End of python code--
```

In different calls, LLM for code review and second iteration code generation, the same prompt as given in Chapter Evaluation of LLM Generated Code was used. In the second iteration of code generation, the following prompt was used.

Second Iteration Code Generation Prompt

```
System Message:
You are a skilled programmer tasked with improving the code.  Use similar
examples, the code generated by LLM, and the issues found in that code as a
reference to create a better version.
<|user|>
```
Inputs:

- In the 'Similar Test Case' section, you'll find a test case with its steps and code for reference.
- Below that, there is Python code generated by a model based on the given test steps.

- This code was reviewed using LLM as a judge, and the feedback, including score, comments, and issues, is listed in the 'Comments and Issues' section.

Task:

- Your task is to update and improve the code using the similar test case, provided test steps, and the LLM as a judge's feedback.
- While making changes, keep the overall structure and correct parts of the original code the same.
- Only update sections with clear differences or those related to fixing the identified issues.

Adheres to PEP 8 coding guidelines and uses **snake_case** for variable, function, and method names. Use **chain of thought** reasoning while generating the outputs.
Similar Test Case 1: Generate a Python script based on the provided testing steps to verify the test mentioned in the summary.
Similar Test Case Python Code 1:

```
###--Start of python code--
<python code>
###--End of python code--
```

Test Case to be Automated:
Code Generated by LLM:

```
###--Start of python code--
<python code>
###--End of python code--
```

Comments and Issues:

Expected Output: The generated code should be fully accurate and optimized to achieve a higher score when reviewed again by the LLM as a judge.

Chapter 8

Results

This chapter includes the key results of different experiments performed. This includes the comparison and selection results of various LLMs. Observations on LLM response on change in hyper-parameters, analysis of distance, and retrieved context when different approaches for embedding test cases applied. Observations regarding LLM Judge selection and outcomes of different combinations of prompts and LLMs are mentioned in detail. Findings related to comparison scores given by human experts and LLM Judge presented. Impact on improvement of score and accuracy of generated code with increasing iterations discussed.

8.1 Few Shot Learning for Code Generation

The aim of performing Few Shot Learning was to select the best LLM for code generation and find a set of hyper-parameters, which results in the high accuracy of generated code.

8.1.1 LLM Selection for Code Generation

A comparison between **CodeLlaMa:7B** and **LlaMa3.1:8B** was made on a set of ten query test cases. Expert verification of code generated by these two LLMs using identical prompts and hyper-parameters was made. Table 8.1 presents the score of each test case and the average score over the ten test cases based on the complexity.

Test Cases	CodeLlaMa:7B	LlaMa3.1:8B	Complexity
Test Case 1	9	9	Simple
Test Case 2	9	9	
Test Case 3	6.5	4	Complex
Test Case 4	7	7	Medium
Test Case 5	7	7	
Test Case 6	7	9	
Test Case 7	8	8	
Test Case 8	6	9	
Test Case 9	8	8	
Test Case 10	9	8	
Total Score	76.5	78	
Average Score	7.65	7.8	

Table 8.1: Score comparison of code generated by CodeLlaMa:7B and LlaMa3.1:8B

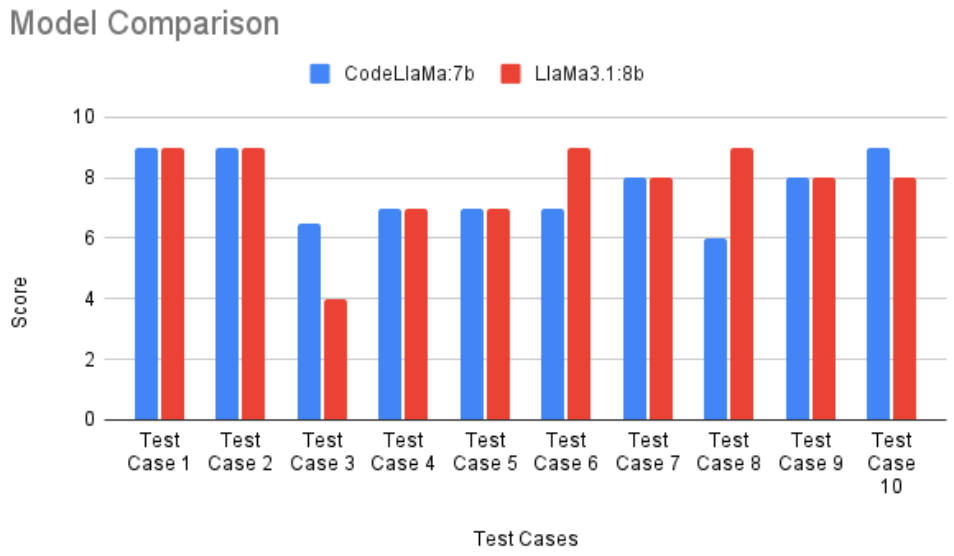


Figure 8.1: Graphical representation of scores given by CodeLlaMa:7b and LlaMa3.1:8B

One-on-one comparison implies that **LlaMa3.1:8B** performed slightly better than **CodeLlaMa:7B**. Due to the longer context length of **LlaMa3.1:8B** (131072) than **CodeLlaMa:7B** (16384), the former was selected for code generation pipeline.

8.1.2 Effect of Hyper-parameters on Code Generation

Generated code quality assessed with increasing temperature from 0.1 to 1. With increasing temperature, the quality of the generated code was denatured. For lower temperature, generated code consistent with the imports, classes, methods, and naming style from similar test code provided, but with higher temperature deviation observed in terms of importing irrelevant libraries, defining completely new classes and methods, and aberrant naming style detected. Table 8.4 provides detailed observations.

Experimentation with other hyper-parameters such as top-p and top-k showed that lower values of temperature, top-p, and top-k are better for high-quality code generation. Issues of generation stopped midway resolved by setting these hyper-parameters to a low value. Table 8.2 presents the set of hyper-parameters found to be working better for code generation tasks. This experiment was done with **CodeLlaMa:7B**. Later experiments demonstrated that the same hyper-parameter values perform well with **LlaMa3.1:8B** as well.

Parameter	seed	temperature	top_p	top_k	repeat_last_n	num_ctx
Value	42	0.2	0.2	10	0	16384

Table 8.2: Set of Hyper-parameters working best for code generation task

Using the above set of hyper-parameters, the generated code captures the patterns from two/three similar test cases provided in the context. Table 8.3 summarizes common observations from the analysis of output code generated for ten query test cases, evaluating different factors related to code quality.

Factors	Observations
Imports	As per example test cases
Structure	Correct
Class & methods	Correctly inherited, no new methods defined
All code blocks	correct and similar as given in context
All steps	Logical errors in few cases (only in the 'run_test' method)
Code length	Expected
New import	No new imports observed
Incomplete code	Not observed

Table 8.3: Code quality check factors and observations

Temperature	Common Observations
0.1	• Fewer imports than expected, with some new imports. • The 'teardown' method in the 'main block' is correct and often copied directly from context test cases. • Consistent naming style. • Generation stopped midway.
0.2	• Issues with imports, either partial or completely new imports observed. • Incorrect class inheritance instead of 'BaseInitializeClass'. • New methods defined. • Incorrect or missing '__init__' block. • The main block is either partially correct or directly copied from given examples. • Only the code structure is provided with method names and doc strings.
0.3	• Issues with imports. • Some code blocks are missing. • In some cases, only method names with doc strings are provided.
0.5	• New imports introduced. • Class not inherited correctly. • Incorrect structure. • New methods defined. • The main block is absent; if present, it is directly copied from examples.
0.7	• Incorrect or new imports. • Many observations are similar to temperature 0.5.
0.9	• In many cases, a partial or complete structure is provided. • Incorrect main block. • Some code blocks are directly copied and pasted (e.g., main and teardown). • Incomplete code.
1.0	• Incorrect structure. • Code generation starts but stops midway. • Mimicked the main block. • Inconsistent naming style. • Incorrect and new methods introduced. • Code length is very short.

Table 8.4: Common observations at different temperature values

8.1.3 Two-shot vs Three-shot Learning

In many cases, only two similar test cases to the query test cases were available. Comparison between Two-shot and Three-shot learning for six query test cases suggests that there was no huge difference in generated code quality if the first two similar cases are the same. Score comparison between Two-shot and Three-shot is presented in table 8.5. Only in two cases, the score is greater by the value of 1.

Test Cases	**Two Shots Score**	**Three Shots Score**
Test Case 1	9	9
Test Case 2	9	9
Test Case 3	7	7
Test Case 4	6	7
Test Case 5	7	8
Test Case 6	8	8
Total	**46**	**48**
Average	**7.67**	**8**

Table 8.5: Comparison of scores of two-shot and three-shot learning for CodeLlaMa:7B

8.1.4 Context Window and Copying Issues

To understand which part of the context provided LLM copies in its response, we tagged the different parts of the code. Following are the findings on relatively long code where retrieved three similar test cases have more than 200 lines of code. Figure 8.2 shows the tag part in Test Case 1, Test Case 2, and Test Case 3. The permutations given in yellow boxes show when test cases were presented in the sequence: Test Case 1 - Test Case 2 - Test Case 3 or Test Case 1 - Test Case 3 - Test Case 2 to LLM, generated code contained the tagged part from Test Case 1. This observation was consistent when only two test cases passed to LLM, in the sequence Test Case 1 - Test Case 2 or Test Case 1 - Test Case 3. Similarly, for the other four permutations of three test cases and four permutations of two test cases. A part of the first test case presented to LLM was observed in the generated code in almost all the cases. On smaller length codes, where the number of lines of codes was less than 150, we found that no matter the order in which similar test cases were presented, the generated code was always the same.

Test Case 1	Test Case 2	Test Case 3

from lib.hypervisors.cloud import awslib
import random
import string
 	
class AnonymizedError(Exception):
  def __init__(self, *args, **kwargs):
  super(AnonymizedError, self).__init__(*args, **kwargs)
 	
class TestScenario(AnonymizenitializeClass):
  def __init__(self, *args, **kwargs):
  super(TestScenario, self).__init__(*args, **kwargs)
 	
  def setup_environment(self):
 	
  def generate_policy(self):
 	
  def execute_scenario(self):
 	
  def cleanup(self):
  super(TestScenario, self).teardown()
	import hashlib
	from lib.hypervisors.cloud import awslib
	
	class AnonymizedError(Exception):
	  def __init__(self, *args, **kwargs):
	  super(AnonymizedError, self).__init__(*args, **kwargs)
	
	class TestScenario:
	  def __init__(self, *args, **kwargs):
	
	  def setup_environment(self):
	
	  def compute_md5_and_size(self):
	
	  def verify_changes(self, initial_state, final_state):
	
	  def generate_policy(self):
	
	  def execute_scenario(self):
	
	  def cleanup(self, *args, **kwargs):
		import subprocess
		
		class AnonymizedTest(BaseClass):
		  def __init__(self, *args, **kwargs):
		
		  def setup_environment(self):
		
		  def check_checksum(self, filepaths, checksum_before, checksum_after):
		
		  def verify_checksum_match(self, filepaths, checksum_before, checksum_after):
		
		  def generate_policy(self):
		
		  def cleanup(self, *args, **kwargs):
		
		  def execute_test(self):

1 – 2 – 3 1 – 3 – 2	2 – 1 – 3 2 – 3 – 1	3 – 1 – 2 3 – 2 – 1
1 – 2 1 – 3	2 – 1 2 – 3	3 – 1
	3 – 2	

Figure 8.2: Memorization effect in LLM for large code length

Permutations:	1 – 2 – 3	1 – 3 – 2	2 – 1 – 3	2 – 3 – 1	3 – 1 – 2	3 – 2 – 1
Result	Generated code was same in all the cases					

Figure 8.3: Memorization effect in LLM for smaller code length

8.2 RAG for Code Generation

8.2.1 Similarity Search and Distance Analysis

Manual verification of retrieved test case content was done for 50 input query test cases. Out of three retrieved test cases, **94%** times the first retrieved test case was similar to the input query test case. This number decreases to **82%** for the second and to **68%** for the third retrieved test case. Therefore, the overall accuracy of retrieved context was **81%**.

	Test Cases Reviewed	Top 1 Similar	Top 2 Similar	Top 3 Similar
Total	50	47	41	34
Accuracy	-	94%	82%	68%

Table 8.6: Retrieved test cases review results

The analysis of retrieved test case scores was conducted using input queries from both the same test suite as used to build the vector database and queries from different test suites. Observation showed that all top three retrieved test cases for input query from the same test suite using which vector database created had scores lesser than 0.15 while queries from other test suites had retrieved test case scores more than 0.16.

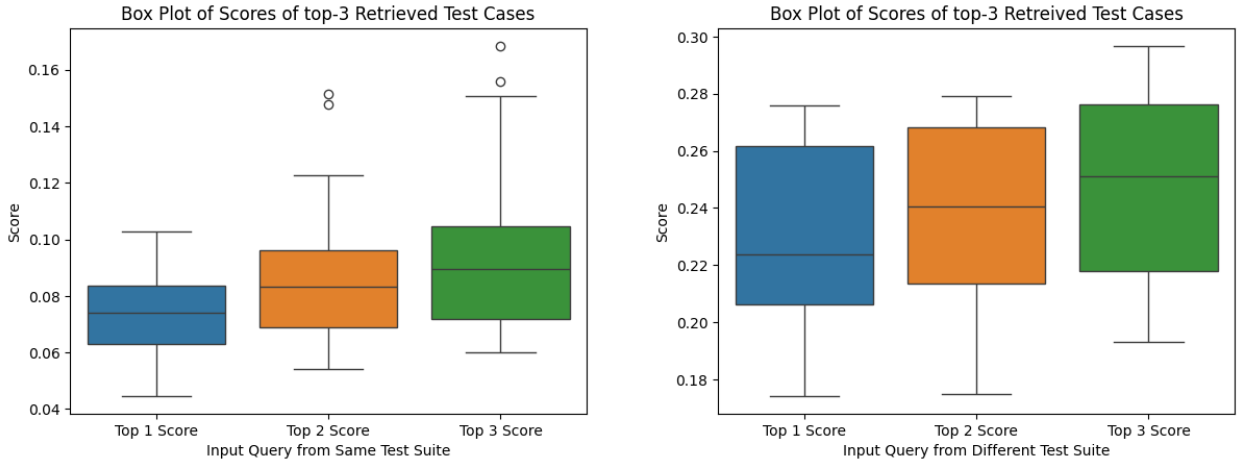


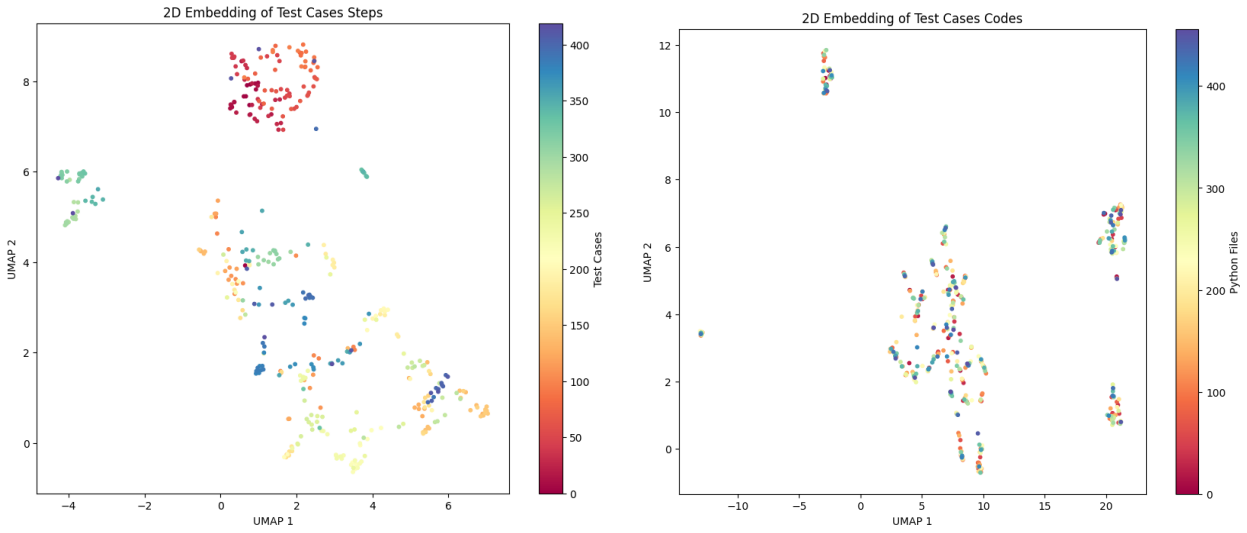
Figure 8.4: Score analysis of top-3 retrieved test cases

Instead of only embedding test case steps, the entire code for each test case was embedded, and a vector database was created. The relevance and similarity of retrieved test cases were evaluated by humans to decide whether storing test case steps was better or the entire code was better. Table 8.7 shows the statistics of similarity of retrieved test cases to the input test case. For 8 out of 20 queries, all three retrieved test cases were similar to the query when test steps were embedded. However, this never happened when the test code was embedded.

Observations	Test Steps	Test Code
All three retrieved test cases are similar to query (3/3)	8/20	0/20
Only two retrieved test cases are similar to query (2/3)	6/20	6/20
Only one retrieved test case is similar to query (1/3)	3/20	2/20
None of the retrieved test cases are similar to query (0/3)	0/20	10/20
Very few steps are similar, but the purpose of retrieved test cases is completely different	3/20	2/20

Table 8.7: Comparison of the number of retrieved test cases, out of three, that are similar to the query by embedding Test Steps and Test Code.

Embedding vectors after embedding test steps and test codes were plotted using Uniform Manifold Approximation and Projection (UMAP), a dimensionality reduction technique, from 768 dimensions to two dimensions. The observations from figure 8.5 support data presented in table 8.7 on a similarity of the retrieved context to the query. A clear cluster of test cases was observed when test steps were embedded. Each color represents a cluster of similar test cases. However, on the test code side, no clear cluster was observed. Each point in figure 8.5a represents the embedding of test steps for a test case while in figure 8.5b represents the embedding of test code for a test case.



(a) Embedding and Plotting Test Steps

(b) Embedding and Plotting Test Codes

Figure 8.5: Distribution of embedding vector on two dimensions

8.2.2 Code Generation Results using RAG

After retrieving similar test cases to the query by storing test steps in a vector database, code was generated using **LlaMa3.1:8B**. Initially, code was generated for only one type of test case and later for multiple. Human evaluation of the code was done as per the scoring guide provided to the expert. Human evaluation, table 8.8, shows that the accuracy of generated code using RAG is around **70-75%**, which is equivalent to Few Shot Learning.

Test Case	**Score**	**Test Case**	**Score**
Test Case 1	8.5	Test Case 10	8
Test Case 2	8	Test Case 11	6
Test Case 3	9	Test Case 12	7
Test Case 4	9	Test Case 13	7
Test Case 5	7.5	Test Case 14	4
Test Case 6	8	Test Case 15	8
Test Case 7	7	Test Case 16	9
Test Case 8	9	Test Case 17	7
Test Case 9	7	Test Case 18	6
Average	7.5		

Table 8.8: Test case scores of generated code for smaller dataset

Satisfactory results were obtained on code generated by smaller datasets as well. While evaluating using **ChatGPT**, only query test case information and generated code were passed. Due to the smaller length and easy-to-fix errors, no further refinement of the generated code was made. LLM used for code generation was **LlaMa3.1:8B**, with six similar test case codes provided as context.

Test Case	ChatGPT	Expert	Test Case	ChatGPT	Expert
Test Case 1	8.0	7.0	Test Case 11	8.0	6.0
Test Case 2	9.0	9.0	Test Case 12	9.0	8.0
Test Case 3	9.0	9.0	Test Case 13	8.0	7.0
Test Case 4	8.0	6.0	Test Case 14	8.0	7.0
Test Case 5	7.5	7.0	Test Case 15	9.0	8.0
Test Case 6	8.0	8.0	Test Case 16	9.0	8.0
Test Case 7	8.0	7.0	Test Case 17	7.5	8.0
Test Case 8	8.0	9.0	Test Case 18	7.0	7.0
Test Case 9	8.0	8.0	Test Case 19	7.0	7.5
Test Case 10	9.0	6.0	Test Case 20	8.0	8.0
Average	**8.15**	**7.52**			

Table 8.9: Comparison of scores given by ChatGPT and an expert for 20 Test Cases

8.3 LLM as a Judge for Code Review

This section provides the in-depth results of experiments performed during the selection of LLM with different combinations of prompts and hyper-parameters.

8.3.1 LLM Selection for Code Review

Various LLMs were explored with different context lengths for LLM as a Judge. Figure 8.6 provides scores given by different LLMs over the 10 test cases and comments by humans on the issues suggested by the LLM Judge. The same scoring guide was presented as given in Chapter Evaluation of LLM Generated Code.

Test Case	Model Name								
	codellama:7b num_ctx = 16384 temperature = 0.2	gemma:9b num_ctx = 8192 temperature = 0.2	phi3:medium14b num_ctx = 8192 temperature = 0.2	phi3:medium14b num_ctx = 16384 temperature = 0.5	phi3:medium14b num_ctx = 16384 temperature = 0.2	llama3.1:8b num_ctx = 64000 temperature = 0.2	llama3.2:8b num_ctx = 16384 temperature = 0.2	llama3.2:3b num_ctx = 64000 temperature = 0.2	Score Given by an Expert
Test Case 1	8	No score given	No score given	8	9	8	7	7	6
Test Case 2	8	No score given	Hallucinated	8	9	8	7	8	9
Test Case 3	No score given	No score given	Hallucinated	8	7	7	7	7	6
Test Case 4	7	6	Hallucinated	8	7	8	7	8	7
Test Case 5	6	4	Hallucinated	8	7	6	7	7	7
Test Case 6	7	4	Hallucinated	8	7	6	6	7	5
Test Case 7	7	6	Hallucinated	9	7	8	8	8	7
Test Case 8	7	7	Hallucinated	9	8	8	8	8	7
Test Case 9	6	6	Hallucinated	8	7	8	7	8	6
Test Case 10	7	7	Hallucinated	8	7	8	7	8	6
Average Score	**7**	**5.7**	-	**8.2**	**7.5**	**7.5**	**7.1**	**7.6**	**6.6**
Comments	Comments are very superficial. Discussed naming style, log messages, imports and missing functions. Does not give detialed explanation. Comments are mostly positive. Score and comments mismatch.	Due to large context, LLM failed to generate understadable comments in first three cases. Comments are organize into Strengths, weaknesses, and suggestions for improvement. Points given in weaknesses can be useful for better code generation. Useful coments for test case 4 onwards.	Hallucinated in all cases.	Weekeness are given in depth. In two cases, expert gave similar comments. Suggestions are looking fair and related with test steps. Can be usefult for generation of code in next iteration.	LLM gave detailed list of issues. Many of the issues mention by LLM are also mentioned by an expert. Score given by LLM justifies with the comments.	In most of the cases, comments mention the missing functions and methods. Comments about the change in log message, missing or unnecessary imports, and teardown blcok which can be ignored.	In most of the cases LLM talks about- lack of comments, inconsistent naming conventions missing imports, logging statements, error handling etc. which are redundant comments. Rarely LLM suggested use of incorrect functions and commands	Talks about code duplication, variable naming, error handling, comments, log messages, indentation, cleanup methods. Redundant comments.	

Figure 8.6: Summary of comments given by different LLMs for LLM as a judge

Effect of Temperature on Score given by LLM Judge

Phi4:medium14b was used for assessing the effect of temperature on the score. This was the same model which has given comments similar to the human expert. Table 8.10 highlights the score for ten test cases for different temperatures by keeping other hyper-parameters and prompt the same with one similar context test case example.

Phi3:medium14B with different temperature			
Temperature	**0.5**	**0.2**	**0.1**
Test Case 1	8	9	8
Test Case 2	8	9	9
Test Case 3	8	8	9
Test Case 4	8	8	8
Test Case 5	8	8	8
Test Case 6	8	8	8
Test Case 7	9	8	8
Test Case 8	9	9	9
Test Case 9	8	8	8
Test Case 10	8	8	8
Average Score	**8.2**	**8.3**	**8.3**

Table 8.10: Score given by Phi3:medium14B at different Temperatures

8.3.2 Variation in Context Provided to LLM Judge

For generating reliable issues and scores from LLM, experiments with different contexts were performed. Test steps, one similar test case, and its code, two similar test cases and their codes passed to the LLM judge along with the code generated by **LlaMa3.1:8B**. LLM used for in this case was **Phi3:medium14B**.

LLM Judge Performance with Varying Context Examples			
Test Case	**No Context Example**	**One Context Example (K=1)**	**Two Context Examples (K=2)**
Test Case 1	8	7	No Response Generated
Test Case 2	6	7	No Response Generated
Test Case 3	6	6	No Response Generated
Test Case 4	6	7	6
Test Case 5	Hallucinated	6	6
Test Case 6	Hallucinated	6	No Response Generated
Test Case 7	6	8	8
Test Case 8	6	7	8
Test Case 9	6	6	6
Test Case 10	6	6	6
Average	**6.25**	**6.6**	**6.66**

Table 8.11: Performance comparison of test cases with different number of context examples

8.4 Pipeline for Code Generation

8.4.1 First Iteration of Code Generation

After the release of **Phi4:14B**, a comparison of code generated by it was made against **LlaMa3.1:8B**. Various context lengths, and a number of context examples were explored. Figure 8.7 represents the winning rate of **LlaMa3.1:8B** against three different configurations of **Phi4:14B** after human evaluation of the generated code. The maximum context length of **Phi4:14B** is 16k, which limits the number of similar test cases that can be passed to it. Configuration of **LlaMa3.1:8B** used is same as given in table 5.1

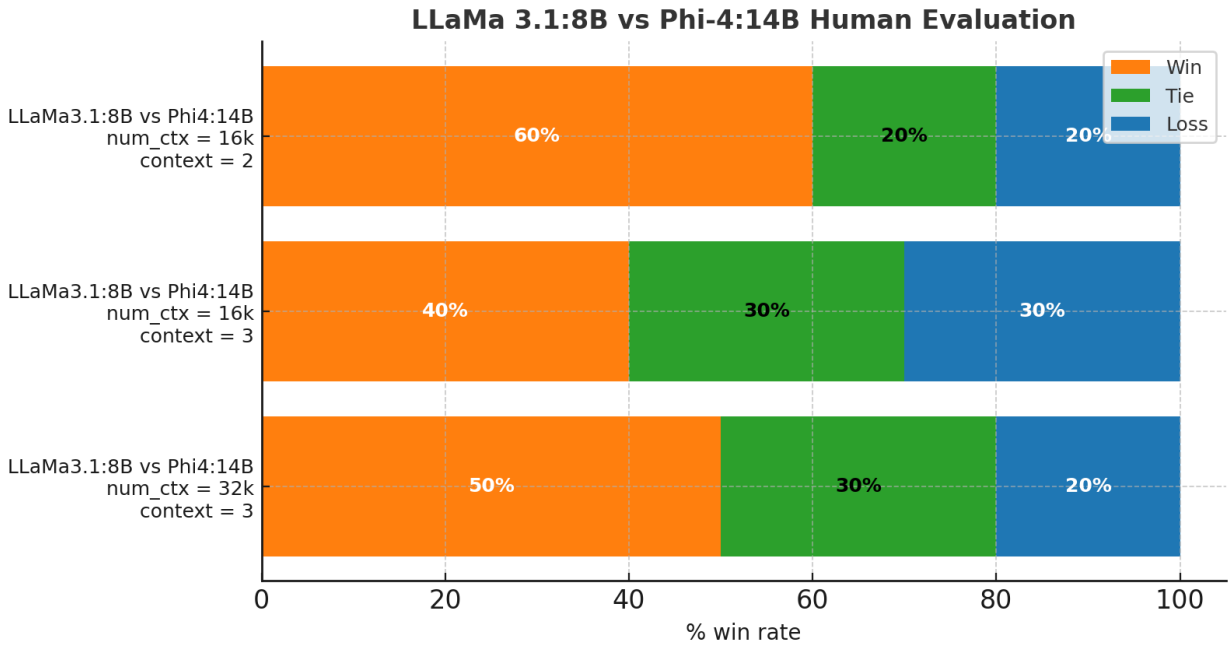


Figure 8.7: Winning rate of LlaMa3.1:8B against different configurations of Phi4:14B

8.4.2 First Iteration Code Review

The first iteration code generated by **LlaMa3.1:8B** was reviewed by **Phi3:14B** and **Phi4:14B** using one similar test case code provided as a reference. To check the reliability of the score and issues mentioned after the first iteration code review, the code was generated again using **LlaMa3.1:8B**. The generated code was checked by human experts for improvements as per suggestions given by the LLM Judge. It was observed that **LlaMa3.1:8B** was not able to fix the issues given by the LLM Judge. In many cases, the code was exactly the same after the second iteration. This generated code was then reviewed using **Phi3:14B** and **Phi4:14B**, and the generated score was checked. On average, the score given by **Phi4:14B** was found to be more reliable, as shown in the table 8.12.

Model Name	Phi3:14B		Phi4:14B		Human Eval.
	First Iter.	Second Iter.	First Iter.	Second Iter.	(First Iter.)
Test Case 1	6	6	8	8	7
Test Case 2	7	8	7	8	8
Test Case 3	7	7	7	7	6
Test Case 4	7	8	7	7	7
Test Case 5	6	7	7	8	7
Test Case 6	7	8	8	7	4
Test Case 7	8	8	7	8	8
Test Case 8	8	8	9	7	9
Test Case 9	7	6	7	8	7
Test Case 10	7	8	7	6	6
Average Score	**7**	**7.4**	**7.4**	**7.4**	**6.9**

Table 8.12: Comparison of Phi3:14B and Phi4:14B models scores after first and second iteration code generation. First and second iteration code generated using LlaMa3.1:8B

8.4.3 Second Iteration Code Generation

Second Iteration Code Generation in Same Prompt as LLM Judge

As described in figure 7.1 first iteration code review, second iteration code generation, and code review of generated code in the second iteration were done in the same prompt using a single LLM call using **Phi3:14B** and **Phi4:14B**. In both of the cases, it was observed that first iteration code review score was very less than the human evaluation. Detailed observation of the code generated after the second iteration showed that only 40-50% issues given by **Phi3:14B** were fixed, which were given by the same model after the first iteration code review. This number was high for **Phi4:14B**, where 70-80% issues were fixed. However, a comparison of the first iteration score with human evaluation shows that there is an average difference of 1 to 1.5 between the score given by **Phi3:14B** and **Phi4:14B**. This raised a question of the reliability of the score given by the LLM Judge. It was also observed that the second iteration code review only commented on positive facts about the generated code and did not mention any further issues present in the code.

Model Name	Phi3:14B		Phi4:14B		Human Eval.
	First Iter.	Second Iter.	First Iter.	Second Iter.	(First Iter.)
Test Case 1	6	8	6	8	7
Test Case 2	7	9	6	9	8
Test Case 3	6	9	6	8	6
Test Case 4	6	9	6	9	7
Test Case 5	6	9	5	9	7
Test Case 6	3	9	6	9	4
Test Case 7	6	9	4	9	8
Test Case 8	6	9	8	9	9
Test Case 9	6	6	6	9	7
Test Case 10	2	9	6	9	6
Average Score	**5.4**	**8.6**	**5.9**	**8.8**	**6.9**

Table 8.13: Comparison of Phi3:14B and Phi4:14B model scores for combined approach. First Iteration code review, second iteration code generation, and second iteration code review were done in the same prompt and using a single LLM call.

Second Iteration Code Generation in Different Prompt

This experiment is described in the figure 7.2. Here, **Phi4:14B** was used for all the code reviews. Since **LlaMa3.1:8B** was not able to fix the issues and generated code after the second iteration was almost or exactly the same as the first iteration code, it was used as a control. **Phi4:14B** was used for the second iteration code generation, but this time using a separate prompt as given in the box 7.2. **Phi4:14B** was able to fix more than 80-85% issues given by itself after the first iteration code review. It was also able to give the same issues and comments after the second iteration code generated by **LlaMa3.1:8B** since it did not fix any of the issues. The average score given by **Phi4:14B** for **LlaMa3.1:8B** generated code for both iterations was the same. Whereas the score given for **Phi4:14B** second iteration generated code was higher than the first iteration score. This score also aligns with the human evaluation score, where two out of three issues given by humans were also suggested by **Phi4**. Table 8.14 shows a comparison between the LLM Judge score and human evaluation score after the first iteration score generated and the second iteration score generated by **LlaMa3.1:8B** and **Phi4:14B**.

Model Name	Phi4:14B		Human Evaluation	
Score	**First Iter.**	**Second Iter.**	**First Iter.**	**Second Iter.**
Test Case 1	8	8	7	8
Test Case 2	7	8	8	9
Test Case 3	7	7	6	7
Test Case 4	7	8	7	7
Test Case 5	7	6	7	8
Test Case 6	8	8	4	4
Test Case 7	7	9	8	8
Test Case 8	9	9	9	9
Test Case 9	7	8	7	8
Test Case 10	7	8	6	8
Average Score	**7.4**	**7.9**	**6.9**	**7.6**

Table 8.14: Comparison of Phi4:14B Model sores with human evaluation where review and code generation are done in different prompts. The model used for the first iteration code generation was LlaMa3.1:8B, and the second iteration code generation was Phi4:14B

Model Name	Phi4:14B		Human Evaluation	
Score	**First Iter.**	**Second Iter.**	**First Iter.**	**Second Iter.**
Test Case 1	8	8	7	7
Test Case 2	7	8	8	8
Test Case 3	7	7	6	6
Test Case 4	7	7	7	7
Test Case 5	7	8	7	7
Test Case 6	8	7	4	4
Test Case 7	7	8	8	8
Test Case 8	9	7	9	9
Test Case 9	7	8	7	7
Test Case 10	7	6	6	6
Average Score	**7.4**	**7.4**	**6.9**	**6.9**

Table 8.15: Comparison of Phi4:14B Model Scores with Human Evaluation where Review and Code Generation done in Different Prompts. The model used for second iteration code generation: LlaMa3.1:8B

8.4.4 Second Iteration Code Review

Since **Phi4:14B** was able to provide reliable scores and gave issues similar to the human evaluators, it was used for the second iteration code review. Comparison between scores already made in table 8.12, 8.14 and 8.15. For the final pipeline, the following approach was used:

1. First Iteration Code Generation: **LlaMa3.1:8B**
2. First Iteration Code Review: **Phi4:14B**
3. Second Iteration Code Generation: **Phi4:14B**
4. Second Iteration Code Review: **Phi4:14B**

Further, using the above pipeline, third iteration code generation and code review were also done. Table 8.16 shows that there was not much change in individual scores and the average score remained the same.

Model Name	**Phi4:14B**			**Human Evaluation**		
Score	**First Iter.**	**Second Iter.**	**Third Iter.**	**First Iter.**	**Second Iter.**	**Third Iter.**
Test Case 1	8	8	8	7	8	8
Test Case 2	7	8	8	8	9	9
Test Case 3	7	7	7	6	7	7
Test Case 4	7	8	8	7	7	8
Test Case 5	7	6	5	7	8	6
Test Case 6	8	8	8	4	4	5
Test Case 7	7	9	9	8	8	8
Test Case 8	9	9	9	9	9	9
Test Case 9	7	8	9	7	8	8
Test Case 10	7	8	8	6	8	8
Average Score	**7.4**	**7.9**	**7.9**	**6.9**	**7.6**	**7.6**

Table 8.16: Comparison of Phi4:14B Model Scores with Human Evaluation (Three Iterations)

Chapter 9

Discussion

This research demonstrated that with careful tuning hyper-parameters of LLMs, combining Retrieval Augmented Generation (RAG) and LLM as a Judge for code review to create an effective code generation pipeline for custom datasets. Key findings include the superior performance of **LlaMa3.1:8B** for first iteration code generation and **Phi4:14B** for finding issues in code with higher accuracy and generating code for subsequent iterations with the ability to fix the issues. The overall accuracy of generated code at the end of the iterative refinement process of generating code and reviewing it by LLM Judge reached up to **80%**, which was the goal of this thesis.

9.1 Interpretation of Few-Shot Learning Results

Experimentation with Few-Shot Learning showed the need for hyper-parameters tuning, without which the quality of output generated by LLMs was significantly degraded on tasks specific to custom datasets such as code generation. Hyper-parameter tuning with prompt engineering is a necessary first step for the generation of output with the custom dataset; it leads to nonsensical generation.

Our experiments on temperature variation revealed that lower temperature settings are generally better suited for code generation tasks. As the temperature increased, the quality and reliability of the generated code deteriorated. As mentioned in table 6.1, increasing temperature leads to new methods and class generation, incomplete code, and stopping in between. A similar pattern was observed with other hyper-parameters such as top-p and top-

k. Lower values of these parameters led to more consistent and predictable code generation. This suggests that code generation tasks, unlike more creative text generation tasks, benefit from more deterministic token selection. Because code relies heavily on recurring elements such as classes, methods, functions, and variable names, constraining the model's token selection to a smaller, more probable set through lower hyper-parameter values limits output diversity but increases reliability and correctness.

Comparison in table 8.9 showed that **LlaMa3.1:8B** slightly outperformed **CodeLlaMa:7B** on our custom code generation task, achieving a higher human evaluation score. **LlaMa3.1:8B** may have been trained on a larger or more diverse dataset that included more code examples. Also, **LlaMa3.1:8B** was released later than **CodeLlaMa:7B** and may have undergone newer data with efficient techniques.

Experiment on Two and Three Shot Learning implies that there is not much difference in the accuracy of the generated score by either of these methods. This observation supports that when there are only two similar test cases present to the query test case, we can still obtain performance equivalent to Three Shot Learning.

Observations in figures 8.2, 8.3 on memorization effect in LLM showed that when context length is lesser compared to the maximum context window of LLM, no matter in which order two or three similar test cases pass to the LLM, every time generated output is same. But when context length is comparable to the maximum context window of LLM, LLM tries to copy from the initial part of the input. This suggests that LLMs focus on the initial part of the input provided when the input content is too large.

Since we explored the experiments of Two and Three Shot Learning and the memorization effect in LLM with only one LLM, future research should explore this behavior for a wider range of tasks and with different LLMs. The limitation of the context window caps the number of similar examples we can provide to the LLM. However, recent advances in the fields of LLM showed the models up to a context length of 2 million. Further research can be focused on providing more than 3 similar examples and assessing the quality of generated code. Further research on the memorization effect and focus on only the initial part of the input provided can be done with different and more advanced LLMs.

9.2 Interpretation of Similarity Search and Distance Analysis

Table 8.7 shows that the FAISS algorithm found more relevant test cases when we embed test steps to search for similar tests rather than embedding the entire code. This is because the test steps are more unique, while the code frequently has the same setup parts (like imports and environment settings) and the same cleanup parts. The differences between test steps are more helpful for finding similar tests

The FAISS algorithm usually found three retrieved tests that were similar to the query test case we were asking it to code. We also found that, for our data, if the distance score was below **0.15**, meaning an **85%** similarity score, the retrieved test cases were truly similar to the test we were asking it to code. If the test was from a different group of tests, the score was usually higher than **0.16** as shown in the figure 8.1. This shows that the embedding model we used (*BAAI/bge-base-en*) is good at understanding different kinds of tasks. This study didn't test other AI models or look closely at the distance scores, which could be good things to explore in the future.

9.3 RAG for Code Generation

Using RAG resulted in a code generation accuracy of around **75%**, as shown in table 8.8 and 8.9. This accuracy is comparable to the code generated using Few-Shot Learning. The transition from FSL to RAG significantly reduced the time spent finding test cases similar to the query. The model and hyperparameter settings optimized in FSL were directly applicable to scaling the code generation process using RAG. The performance of **LlaMa3.1:8B** appears to be generalizable to larger datasets containing various test case types as well as on smaller code datasets. The embedding model ***BAAI/bge-base-en*** also appears to be generalizable and is capable of retrieving test cases of the same type as the query test cases. The reasons for these outcomes have been discussed in previous sections.

9.4 Interpretation of results of LLM as a Judge for Code Review

Exploration with different LLMs suggested that while LLMs can provide useful feedback on code and suggest issues and places for improvement, their performance is dependent on the choice of LLM and the amount of contextual information provided.

As demonstrated in 8.6, different LLMs have varying degrees of success in identifying code defects. This highlights the important role of LLM selection. In our case, initial experimentation and comparison among different LLMs as shown in the 8.6 indicated that **Phi3:14B** provide one third issues which are similar to pointed out by an human expert. However, the optimal hyper-parameter choice may influence the score given by the LLM judge. Table 8.10 showed that a temperature value lower than 0.5 provides the same score by setting other hyper-parameters to lower values.

The impact of providing contextual information, shown in Table 8.11 implies that steps are not sufficient context for the evaluation of generated code. Providing two similar test cases and their code as context for evaluation leads to hallucination because of the limiting context window size of **Phi3:14B**. Therefore, the optimal choice is providing one context example that is most similar to the query test case while reviewing generated code using LLM Judge. This turns out to be the best result in our experiment.

These observations suggest that effective LLM-based code review requires a holistic approach. The LLM must be carefully chosen based on human evaluation of output generated by LLM, the temperature and other hyper-parameters must be tuned to encourage deterministic assessment, and the context must be selected to provide relevant information without which LLM can not point out issues in generated code. The balance of these elements is crucial in leveraging the power of LLMs for code review. These findings are aligned with the paper (Tong and Zhang, 2024).

This study only explores the temperature variation and quality of review and score given by LLM judge by setting other hyper-parameters to the fixed value. Further study can explore a wide range of hyper-parameters. Due to the smaller context window of **Phi3:14B**, only one similar context example could be passed, but further study of score and suggestion quality given by LLM Judge, with the latest LLM with a larger context window and by

providing more context examples can be done.

9.5 Iterative Refinement of Generated Code

Accuracy of generated code increased by almost 7% after iteratively generating code and reviewing it using LLM Judge as presented in the table 8.16. This improvement from 69% to 76% based on human evaluation achieved after the second iteration and further iteration demonstrated no further improvement. It was observed that **LlaMa3.1:8B** is better at generating the first iteration score but not good at the fixing suggestion given by **Phi4:14B**. While **Phi4:14B** is good at both giving issues present in code generated by LlaMa3.1 and fixing those in later iterations of code generations.

After the release of **Phi4:14B** small language model with comparable performance with Gpt-4o, the previous experiment of code generation and review was tried with **Phi4:14B**. Winning rate of **LlaMa3.1:8B** vs. Phi4:14B as indicated in 8.7 demonstrated that **LlaMa3.1:8B** still better in producing first iteration score. Though Phi4:14B outperformed **LlaMa3.1:8B** in many other tasks, **Phi4:14B** is more like a reasoning model and may not be trained on extensive code data. Technical report of **Phi4:14B** indicates that it is trained on synthetic datasets, data from filtered public domain websites, and acquired academic books and Q&A datasets (Abdin et al., 2024). This may be the reason behind out performance of **LlaMa3.1:8B** over **Phi4:14B** for our task.

For review, a comparison was made between **Phi3:14B** and **Phi4:14B** for LLM Judge. It was observed that **LlaMa3.1:8B** was not able to fix the issues suggested by both of these models, and the second iteration code was exactly the same as the first iteration code with minor changes. But **Phi3:14B** showed an average improvement of 4% in the score of the second iteration code, while **Phi4:14B** showed 0% improvement. The score given by **Phi4:14B** is more reliable. Moreover, careful analysis of issues suggested by both of these models revealed that two-thirds of issues (66%) given by **Phi4:14B** are matched with issues given by human experts, while this number is one-third (33%) for the **Phi3:14B**. This surpasses the ability of pointing out issues and giving reliable score to generated code of **Phi3:14B**. This may be due to Direct Preference Optimization (DPO) approach of training **Phi4:14B** making it more align with human preferences and better capability of reasoning.

Further, second iteration code generation observations provide astonishing results, when **Phi3:14B** and **Phi4:14B** were employed. The first iteration code generated by **LlaMa3.1:8B** and first iteration review, second iteration code generation, and second iteration code review combine in one single prompt and single LLM call as shown in the 7.1 obtained results as shown in table 8.13. These results showed that both of these LLM rate codes generated by other LLMs (in this case, **LlaMa3.1:8B**) had very low scores, with an average of **5.6** and **5.9**. But while rating, the second iteration code, generated by themselves, gave a very high score with an average of **8.6** and **8.8**, respectively. However, after human evaluation, it was observed that the code generated in the second iteration by Phi3 and Phi4, none of the code was of such high quality. While Phi4 is able to fix more issues than Phi3. Similar observations are mentioned in the paper 'LLM Evaluators Recognize and Favor Their Own Generations' (Panickssery et al., 2024), where an LLM Judge scores its own outputs higher than others' while human experts consider them of equal quality.

However, instead of combining three processes—first iteration review, second iteration code generation, and second iteration review—when each process was divided into separate calls to the LLM using different prompts for the first and second iteration of code generation and the same prompt for evaluating the generated code, the rating given by **Phi4:14B** appeared more reliable. Almost all the issues mentioned after the first iteration code review were fixed in the second iteration of code generation using Phi4. The score given by **Phi4:14** to the first-generation output and the improved generation after incorporating suggestions was more reliable and aligned more closely with human judgment, as presented in Table 8.14. This experiment demonstrates that by simply breaking the generation and review into separate calls, the LLM does not favor its own generation, as observed in the previous case. **Phi4:14B**, unlike **LlaMa3.1:8B**, was able to fix the issues in subsequent iterations and correctly identify mistakes. This may be due to the Direct Preference Optimization (DPO) training of **Phi4:14B**, which aligns more closely with human preferences. However, due to the context length limitation (16K), **Phi4:14B** cannot learn from just one or two examples to generate code with comparable accuracy to LlaMa3.1, which has a much larger context length (128K) and can easily accommodate three examples. This experiment was limited to **Phi4:14B**, but similar behavior can be studied with multiple different LLMs.

A thorough examination of different LLMs for first iteration code generation, code review, and systematic analysis of scores and issues identified by the LLM Judge led to the final pipeline for iterative code generation. First iteration code generation using **LlaMa3.1:8B**,

followed by code review and subsequent code generation using a breakdown prompt for both generation and review with **Phi4:14B**, improved the first iteration score accuracy from 69% to 76% based on expert evaluation. Meanwhile, the LLM Judge score increased from 74% to 79%, bringing it closer to the primary goal of 80% accuracy for generated code on the larger code-length dataset used in this thesis.

9.6 Future Work

The process of iterative refinement of generated code by implementing LLM as a Judge for code review can be integrated using an Agentic way, and detailed analysis of accuracy and code quality can be done. Models with larger context windows can be provided with more test cases that can be tested for tradeoffs between a number of test cases provided in context and the accuracy of generated code. A model with a larger context window can be used to study the Cache Augmented Generation (Chan et al., 2024), a method where a model is provided with a pre-existing "cache" of relevant information, for example, details of functions and API, that it can refer to when generating responses. Further study of Direct Preference Optimization (DPO) (Rafailov et al., 2024) can be done on smaller and larger datasets to align the models with human preferences.

Bibliography

M. Abdin, J. Aneja, H. Behl, et al. 2024. Phi-4: A 14-billion parameter language model with a focus on data quality. *Phi-4 Technical Report* (2024). https://arxiv.org/abs/2412.08905

Chetan Arora, Ahnaf Ibn Sayeed, Sherlock Licorish, Fanyu Wang, and Christoph Treude. 2024. Optimizing Large Language Model Hyperparameters for Code Generation. *arXiv preprint* arXiv:2408.10577 (August 2024). https://doi.org/10.48550/arXiv.2408.10577

D. Bahdanau, K. Cho, and Y. Bengio. 2014. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*. 1–15.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Association for Computational Linguistics, 65–72. https://aclanthology.org/W05-0909

Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research* 3 (2003), 1137–1155.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Process-*

ing Systems (NeurIPS 2020). Virtual. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6d13c243cd7f1b76663e6bdb3-Abstract.html

Brian Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen Huang. 2024. Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks. https://doi.org/10.48550/arXiv.2412.15605

J. Devlin, M. W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL 2019)*. 4171–4186.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. 2022. Delta Tuning: A Comprehensive Study of Parameter Efficient Methods for Pre-trained Language Models. *Research Square* (March 2022). https://doi.org/10.21203/rs.3.rs-1553541/v1

J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. Smith. 2020. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint* arXiv:2002.06305 (2020). https://doi.org/10.48550/arXiv.2002.06305

J. L. Elman. 1990. Finding structure in time. *Cognitive Science* 14, 2 (1990), 179–211. https://doi.org/10.1207/s15516709cog1402_1

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023a. Out of the BLEU: How Should We Assess Quality of the Code Generation Models? *Journal of Systems and Software* 203 (2023), 111741. https://doi.org/10.1016/j.jss.2023.111741

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023b. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software* 203 (May 2023), 111741. https://doi.org/10.1016/j.jss.2023.111741

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Gener-

ation for Large Language Models: A Survey. *arXiv preprint arXiv:2312.10997* (2023). arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Z. Ji, N. Lee, R. Frieske, et al. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.

J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim. 2024. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology* 1, 1 (2024), 1–49.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

N. Kumar, M. Nanda, P. Soni, et al. 2023. CodeLlama: A Large Language Model for Code Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023)*. 1–13.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 9459–9474 pages.

Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*. Association for Computational Linguistics, 74–81. https://aclanthology.org/W04-1013

F. Liu, Y. Liu, L. Shi, et al. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *arXiv preprint arXiv:2404.00971* (2024). https://api.semanticscholar.org/CorpusID:268819908

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *2021 IEEE/ACM 43rd*

International Conference on Software Engineering (ICSE). 336–347. https://doi.org/10.1109/ICSE43902.2021.00041

Meta. 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. *Meta AI Blog* (2024). https://ai.meta.com/blog/metallama-3/

T. Mikolov, I. Sutskever, K. Chen, et al. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS 2013)*. 3111–3119.

OpenAI. 2023. GPT-4 technical report. *OpenAI* (2023). https://openai.com/research/gpt-4

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. 2024. LLM Evaluators Recognize and Favor Their Own Generations. arXiv:2404.13076 [cs.CL] https://arxiv.org/abs/2404.13076

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 311–318. https://doi.org/10.3115/1073083.1073135

L. R. Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* 77, 2 (1989), 257–286. https://doi.org/10.1109/5.18626

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290 [cs.LG] https://arxiv.org/abs/2305.18290

C. E. Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal* 27, 3 (1948), 623–656. https://doi.org/10.1002/j.1538-7305.1948.tb00917.x

B. Sherifi, K. Slhoub, and F. Nembhard. 2024. The Potential of LLMs in Automating Software Testing: From Generation to Reporting. *arXiv preprint arXiv:2501.00217* (2024). https://doi.org/10.48550/arXiv.2501.00217

I. Sutskever, O. Vinyals, and Q. V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS 2014)*. 3104–3112.

K. S. Thant and H. H. K. Tin. 2023. The Impact of Manual and Automatic Testing on Software Testing Efficiency and Effectiveness. *Indian Journal of Science and Research* 3, 3 (2023), 88–93.

Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating Code Generation with Large Language Models. arXiv:2410.02184 [cs.LG] https://arxiv.org/abs/2410.02184

H. Touvron, M. Mouzannar, M. Funtowicz, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023).

P. Tripathy and K. Naik. 2014. *Software Evolution and Maintenance: A Practitioner's Approach*. Wiley. https://doi.org/10.1002/9781118964637.fmatter

Doug Turnbull. 2015. BM25: The Next Generation of Lucene Relevance. OpenSource Connections (blog). https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/

A. Vaswani, N. Shazeer, N. Parmar, et al. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS 2017)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

J. Wang et al. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui. 2025. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *ACM Transactions on Software Engineering and Methodology* Just Accepted (January 2025). https://doi.org/10.1145/3714461

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. arXiv:2309.07597 [cs.CL]

Hao Yu, Aoran Gan, Kai Zhang, Shiwei Tong, Qi Liu, and Zhaofeng Liu. 2024. Evaluation of Retrieval-Augmented Generation: A Survey. *ArXiv* abs/2405.07437 (2024). https://api.semanticscholar.org/CorpusID:269758033

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A Survey of Large Language Models. *arXiv preprint* arXiv:2303.18223 (2023). https://arxiv.org/abs/2303.18223

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS 2023)* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2020, 29 pages. https://github.com/lm-sys/FastChat/tree/main/fastchat/llm_judge