

Code Summarizer: A Deep Learning Transformer Based Model



Submitted by

Kainat Farooqi

AU-02-05-10229

Maqddus Butool

AU-02-05-10220

Safa Saeed

AU-02-05-10234

Qirat Qadeer

AU-02-05-10201

Mehvish Zahid

AU-02-05-10207

Supervised by

Dr. Nazia Bibi

**Submitted in Partial Fulfillment for the Degree of Bachelor of Studies in
Computer Science**

**FAZAIA BILQUIS COLLEGE OF EDUCATION FOR WOMEN, PAF NUR
KHAN, RAWALPINDI**

(June, 2024)

This work, entitled
**“Code Summarizer: A Deep- Learning Transformer-Based
Model”**

has been approved for the award of

Bachelors of Studies in Computer Science

Date: 27th June, 2024

Supervisor: Dr. Nazia Bibi

External Examiner: Dr. Saleem Iqbal

Head of Department: Ms. Tayyaba Naseer

Department of Computer Science

FAZAIA BILQUIS COLLEGE OF EDUCATION FOR WOMEN,
PAF NUR KHAN, RAWALPINDI

INTELLECTUAL PROPERTY RIGHT DECLARATION

This is to declare that the work under the supervision of **Dr. Nazia Bibi** having title: **“Code Summarizer: A Deep-Learning Transformer-Based Model”** carried out in partial fulfillment of the requirements of Bachelor of Studies in Computer Science, is the sole property of the Fazaia Bilquis College Of Education For Women, PAF Nur Khan, Rawalpindi and is protected under the intellectual property right laws and conventions. It can only be used for purposes like extension for further enhancement, product development, adoption for commercial/organizational usage, etc., with the permission of the University.

Date: 27th June, 2024

Kainat Farooqi Signature: _____

Maqddus Butool Signature: _____

Safa Saeed Signature: _____

Qirat Qadeer Signature: _____

Mehvish Zahid Signature: _____

ANTI-PLAGIARISM DECLARATION

This is to declare that the above publication produced under the supervision of Dr. Nazia Bibi having title: **Code Summarizer: A Deep-Learning Transformer-Based Model** is the sole contribution of the authors and no part hereof has been reproduced on as it is basis (cut and paste) which can be considered as Plagiarism. All referenced parts have been used to argue the idea and have been cited properly. We will be responsible and liable for any consequence if a violation of this declaration is determined.

Date: 27th June, 2024

Authors:

Kainat Farooqi Signature: _____

Maqddus Butool Signature: _____

Safa Saeed Signature: _____

Qirat Qadeer Signature: _____

Mehvish Zahid Signature: _____

ACKNOWLEDGMENTS

In the name of Allah, the Most Gracious and the Most Merciful Alhamdulillah, all praises to the Creator of the universe for the strengths and His blessing in completing this project. This study is nothing, but an effort to understand and articulate the principles of one of the several hundred thousand phenomena, with a tool, the brain, a precious gift from the Almighty.

We would like to express our sincere gratitude to our advisor Dr. Nazia Bibi for the continuous support of our BS Project, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped us in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for this project.

Kainat Farooqi

AU-02-05-10229

Maqddus Butool

AU-02-05-10220

Safa Saeed

AU-02-05-10234

Qirat Qadeer

AU-02-05-10201

Mehvish Zahid

AU-02-05-10207

DEDICATION

We dedicate this project to the sake of Allah, our creator, and our great teacher and messenger, Muhammad (May Allah Bless and Grant him), who taught us the purpose of life. Our beloved Supervisor, Dr. **Nazia Bibi** leads us through the valley of darkness with light of hope and support. Our great parents never stopped giving of themselves in countless ways. Our friends have encouraged and supported us. All the people in our life who helped us in completing this project, we dedicate this project to them.

TABLE OF CONTENTS

Chapter 1	INTRODUCTION	2
1.1	Background.....	3
1.2	Purpose of the Project.....	3
1.3	Problem Statement	4
1.4	Existing Systems	4
1.5	Research Gap.....	5
1.6	Goals and Objectives.....	5
1.7	Proposed Solution	6
1.8	Scope of this Project.....	7
1.9	Tools and Techniques.....	7
Chapter 2	LITERATURE REVIEW	8
2.1	Literature Review	8
CHAPTER 3	Data Analysis.....	15
3.1	Data Analysis	15
3.2	System Requirements.....	16
3.2.3	Clients, Customers and Users.....	16
3.3	Functional and Data Requirements.....	17
3.4	Non-Functional Requirements.....	17
CHAPTER 4	DESIGN CONSIDERATIONS	19
4.1	Design Constraints	19
4.2	Architectural Strategies	19
4.3	Methodology	22
4.4	Deep Learning Model	25
4.1	Project Management Strategies	33
CHAPTER 5	SYSTEM DESIGN.....	37
5.1	System Architecture and Program Flow.....	37
5.2	Program Flow	38

5.3 Detailed System Design.....	39
5.4 Use Case Diagram	39
5.4.1 Use Cases Description	40
5.5 Sequence Diagram.....	44
5.6 Class Diagram.....	47
CHAPTER 6 IMPLEMENTATION.....	48
6.1 Introduction	48
6.2 Algorithms	51
6.3 Comparison	56
6.4 Libraries and Packages	57
6.5 User Interface (UI).....	58
6.6 Case Study	62
Chapter 7 TESTING AND EVALUATION.....	68
7.1 Testing	68
7.2 Unit Testing	68
7.3 Test Cases.....	68
Chapter 8 CONCLUSION AND FUTURE WORK.....	79
8.1 Conclusion.....	79
8.2 Future Work.....	79
REFERENCES	80
Appendix A.....	82
Appendix B.....	85

LIST OF FIGURES

Figure 1.1: Flow Diagram of Code Summarizer Website	2
Figure 4.1: Architecture Diagram	19
Figure 4.2: Methodology.....	21
Figure 4.3: Llama 3 Architecture	24
Figure 5.1: Data Preprocessing	30
Figure 5.2: DFD Level 0	31
Figure 5.3: DFD Level 1	32
Figure 5.4: Use Case Diagram	33
Figure 5.5: System Sequence Diagram	37
Figure 5.6: Class Diagram	38
Figure 6.1: Architectural Diagram of Proposed System	40
Figure 6.2: Model Training Code	43
Figure 6.3: Llama 3 Train Loss Graph.....	43
Figure 6.4: Evaluation Graph for Llama 3	44
Figure 6.5: CodeBERT Train Loss Graph	45
Figure 6.6: Ealuation for CodeBERT	46
Figure 6.7: Comparative Analysis of Llama 3 and CodeBERT	47
Figure 6.8: Home Page.....	49
Figure 6.9: Sign Up or Rgistration Page	50
Figure 6.10: Login Page	50
Figure 6.11: Code Summarizer Page	51
Figure 6.12: Contact Us Page	52
Figure 6.13: Code Summarizer Output	52
Figure 6.14: ChatGPT Output.....	53
Figure 6.15: Gemini Output	53

Figure 6.16: CodeGPT Output	54
Figure 7.1: Input Code Test Case.....	59
Figure 7.2: Sign Up Page Validation Testing	60
Figure 7.3: Login Page Testing.....	61
Figure 7.4: Code Summarizer Testing	62
Figure 7.5: Error Detection Testing	63
Figure 7.6: Comment Provider Testing.....	64
Figure 7.7: Usability and User Experience Testing	65
Figure 7.8: Contact Us Form Testing.....	66
Figure 7.9: Social Media Handle Testing	67
Figure 7.10: Responsiveness Testing.....	68

LIST OF TABLES

Table 1.1: Research Gaps and Objectives Mapping	6
Table 1.2: Tools and Technologies	7
Table 2.1: Literature Review	12
Table 3.1: Dataset Used in Code Summarizer	14
Table 4.1: Project Management Strategies	27
Table 4.2: Time Management	28
Table 5.1: Enter Code Use Case	33
Table 5.2: Register Use Case	34
Table 5.3: Log In Use Case.....	34
Table 5.4: Summary Function Use Case.....	35
Table 5.5: Error Detection Function Use Case	35
Table 5.6: Comment Provider Function Use Case.....	35
Table 5.7: Customization Function Use Case.....	36
Table 5.8: Generate Result Use Case.....	36
Table 5.9: Display Result Use Case.....	37
Table 6.1: Evaluation Table for Llama 3	44
Table 6.2: Evaluation Table for CodeBERT.....	46
Table 6.3: Evaluation Comparison of Llama 3 and CodeBERT.....	48
Table 6.4: Comparison of Various Deep Learning Models	56
Table 6.5: Comparison of Various Deep Learning Models using BLEU, ROGUE-L and METEOR	57
Table 7.1: Test Case 1 (Input Code Testing)	58
Table 7.2: Test Case 2 (Sign Up Page Testing)	59
Table 7.3: Test Case 3 (Login Page Testing).....	61
Table 7.4: Test Case 4 (Code Summarizer Testing)	61

Table 7.5: Test Case 5 (Error Detection Testing)	62
Table 7.6: Test Case 6 (Comment Provider Testing).....	63
Table 7.7: Test Case 7 (Performance and Scalability Testing).....	64
Table 7.8: Test Case 8 (Usability and User Experience Testing)	64
Table 7.9: Test Case 9 (Contact Us Form Testing).....	65
Table 7.10: Test Case 10 (Social Media Handle Testing)	66
Table 7.11: Test Case 11 (Responsiveness Testing).....	67

ABSTRACT

With the rapid expansion of software development projects, understanding and summarizing large volumes of source code has become increasingly challenging. Human-written comments often fall short of providing comprehensive insights due to their labor-intensive nature and susceptibility to become outdated. The goal of source code summarization is to produce natural-language, concise descriptions of code snippets, making program comprehension and software maintenance easier. A rapidly expanding area of research is automatic code summarization, particularly given that the community has benefited greatly from advancements in AI and neural networks. Techniques for summarizing source code typically take the source code as input and produce a description written in natural language. Foundation models function admirably for some programming tasks. After being trained with billions of code tokens, these models are fine-tuned with hundreds of thousands of labeled examples, which are typically taken from numerous projects. Software phenomena, on the other hand, may be very project-specific. To address this issue, we tend to explore the working of two deep learning models; CodeBERT and Llama 3. Our objective is to explore which of these models can work effectively for generating summaries of the provided source code in Java and Python. The best working model is then integrated into a user-friendly interface to demonstrate real time generation of code summaries to create a code summarizer tool. The project is evaluated based on its effectiveness in generating summaries. Through rigorous testing and validation, the project demonstrates its potential to significantly improve the effectiveness of the code summarization process. Additionally, we explore the effectiveness of the error detection, customization, and comment provider features. This project contributes to the field of automated code analysis by demonstrating the effectiveness of a Large Language Model for code summarization and by providing a valuable tool for developers. In conclusion, this project enhances the capability of two deep learning models by fine-tuning them on the subset of a benchmark dataset, offering potential to improve the summary generation task.

Keywords: Artificial intelligence, CodeBERT, CodeSearchNet, Code Summarization, Deep learning, Llama3, Large Language Model, Multi-headed attention mechanism, Transformers

Chapter 1 INTRODUCTION

Code summarization has also become a crucial task in software development. Also called code comment generation, it is a text description for the function and purpose of special identifiers in computer programs (C. Zhang et al., 2022). With concepts like Object-Oriented Programming, code reusability has become a common practice. Popular platforms like Github, StackOverflow and Hugging Face enable the developers to use the code present in their repositories. But the problem of whether a piece of code is required to a developer or not is still encountered. Program comprehension is a necessary part of software development and maintenance because programs can't be modified properly unless they are well understood. Documentations or comments in the code provide a concise explanation of the code and aid in whether a particular piece of code is useful to the developer or not. But many a times, the comments are not present in the code or are too vague for the developers to understand or are obsolete. This hinders in their ability to decide whether to use a particular piece of code or not. Almost all automatic source code summarization technology involves the following stages: source code modeling, code summarization generation, and quality evaluation (C. Zhang et al., 2022)

Our project aims to improve code summarization further by fine-tuning two transformer-based models (Llama 3 and CodeBERT) and analyzing their performances. System Flow Design is being demonstrated in Figure 1.1 emphasizing the flow of our project.

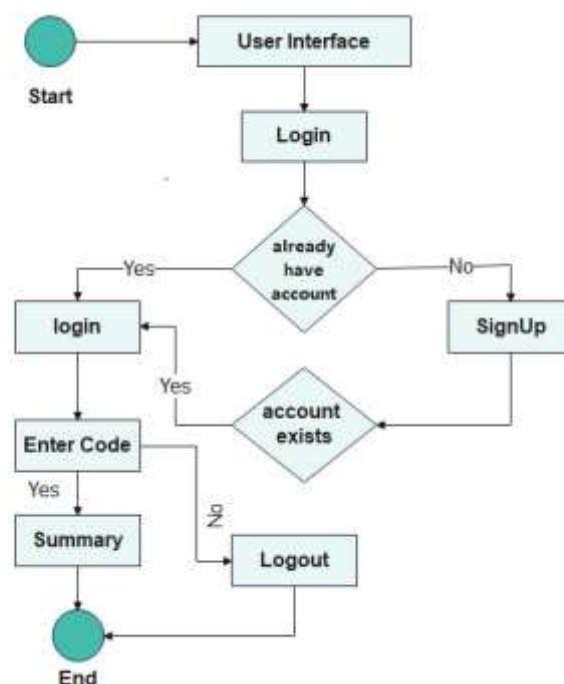


Figure 1.1: Flow Diagram of Code Summarizer Website

In summary, this research project aims to explore two deep learning models and use the best one to create a code summarization tool that mainly focuses on generating code summaries and additionally, explore the functionalities like error detection, comment provider and customization. By harnessing the power of deep learning models, the proposed system generates accurate summaries, aiding in the quest of whether to use a particular code or not.

1.1 Background

In the present programming advancement landscape, the volume and intricacy of codebases keep on increasing dramatically, presenting critical moves for software developers to comprehend, maintain, and document code effectively. A recent empirical study shows that 80% of practitioners consider that code summarization tools can help them improve development efficiency and productivity (Xing Hu et al., 2022). Conventional techniques for code documentation like human-recorded feedback, frequently end up being lacking, work escalated, and inclined to become obsolete as code advances over a long time. Thus, there is a need for automated arrangements that proficiently summarize code and give software developers brief insight into its usefulness and purpose. Advancements in Deep Learning have paved way for extraordinary ways to deal with code summarization. Transformer-based models like BERT, CodeT5, and Llama 2 pretrained on huge archives of code and text have shown promising outcomes in different code-related tasks, including code understanding, completion and summarization. Expanding upon these progressions, this project aims to leverage the power of CodeBERT and Llama 3 to find out an effective model for summary generation additionally for error detection, code comment provider and customization), customized explicitly for Java and Python codebases. By incorporating the best model into an easy-to-use web-based platform, this task tries to democratize admittance to cutting-edge code summarization strategies, engaging developers of all expertise levels to explore and comprehend complex codebases more easily. In short, the project offers a solution to the difficulty related to code understanding and documentation in modern software development environments.

1.2 Purpose of the Project

The purpose of this project is to revolutionize the way developers comprehend, interact with and collaborate on codebases in Python and Java programming languages. This

purpose is driven by the recognition of escalating complexities in modern software systems, emphasizing the need for innovative tools to enhance code readability, streamline collaboration, and ultimately elevate the efficiency of the development process.

This project aims to develop a robust code summarization website using deep learning. The goal is to automate the creation of concise and coherent code summaries; making it easier for developers to understand and manage complex software systems. By leveraging sophisticated deep learning techniques, the project enhances code comprehension, provide the facility of error detection, comment provider and customization to streamline developer workflow, and contribute to more efficient and collaborative software development practices.

1.3 Problem Statement

Machine learning and deep learning has automated various software development steps. However, comprehending and summarizing vast amount of source code poses a significant threat to the developers and software engineers. This complexity is compounded by the intricate nature of modern programming languages, diverse coding styles and the continuous evolution of software projects. Human-written comments, intended to offer insights into code functionality, often prove insufficient in providing comprehensive and up to date information. The creation and maintenance of detailed comments require substantial time and effort, making it impractical to keep pace with the rapid development and frequent updates inherent in modern software projects. Human-written comments are susceptible to becoming outdated or incomplete over time, failing to reflect the latest changes and leading to a disconnection between the actual code implementation and its corresponding documentation. This not only depletes code comprehension but also poses challenges in maintaining software integrity.

1.4 Existing Systems

Pre-trained models in Natural Language Processing (NLP) have witnessed significant advancements with models like BERT and GPT demonstrating exceptional performance in tasks such as language understanding, translation and sentiment analysis. Techniques such as natural language processing (NLP) methods, recurrent neural networks (RNNs), and attention mechanisms are used for capturing the intricacies of code and generating coherent and contextually relevant summaries (Zhou et al., 2022). Both sequential and

structural information of the code is to be considered if the model aims to generate accurate summaries. This involves an exploration of models that effectively capture the sequential nature of code execution and dependencies, highlighting the role of Recurrent Neural Networks (RNN), long short-term memory (LSTM), and other sequential modelling architectures (Choi et al., 2022). The structural information includes abstract syntax trees (ASTs), graph-based representations, and other structural components that enhance the understanding of code hierarchies and relationships, contributing to more accurate and context-aware summaries (Choi et al., 2022). Many transformer-based models have been proposed for generating code summaries like SG-Trans, Hybrid DeepCom etc. which serve as a starting point for our project. But these models lack the sophistication that generating accurate code summaries need.

1.5 Research Gap

There are some areas in the field of code summarization that need more attention from researchers. Firstly, pre-training models such as BERT and GPT have shown impressive performance in natural language processing but their application to code summarization is still underexplored. Given the unique challenges posed by syntax and semantic of code, there is room to understand how to properly use and apply them for code summarization tasks (Choi et al., n.d). Furthermore, current models such as SG-Trans and Hybrid-DeepCom have demonstrated the effectiveness of code summarization by introducing code structure features and combining dictionaries with structure information. However, there is room in exploring new architectures that better capture the characteristics of benchmarks and generate accurate and contextual summaries (Gao et al., 2023).

1.6 Goals and Objectives

Our project aims to leverage advanced technologies to address the challenges associated with code comprehension and summarization in software development. Some common goals and objectives of our project are as follows:

- i) To fine-tune and analyze the working of CodeBERT and Llama 3 to find out which one can be effectively used for code summarization.
- ii) To design and implement a user-friendly web-based platform capable of automatically summarizing code snippets using more effective of the two models that is accessible to developers of all skill levels and backgrounds.
- iii) To ensure that the platform generates accurate and informative summaries.

- iv) To streamline summary generation by automatically generating up to date summaries that serve as valuable references for developers and stakeholders, thereby improving the overall quality of code summarization.
- v) To create a code summarization tool that provides additional functionality in addition to code summarization like error detection, code comment provider, and customization.

Table 1.1: Research Gaps and Objectives Mapping

Sr.	Research Gaps	Objectives
i	Pre-training models like BERT and GPT excel in NLP but are underexplored in code summarization. (Wang et al., 2021)	Fine-tune and evaluate CodeBERT and Llama 3 to identify the most effective model for code summarization.
ii	Current models like SG-Trans and Hybrid-DeepCom introduce code structure features but need exploration of new architectures for better capturing characteristics of benchmarks and generating accurate summaries. (Wang et al., 2021).	To design and implement a user-friendly web-based platform capable of automatically summarizing code snippets using the more effective model .
iii	The unique challenges posed by syntax and semantics of code require better understanding of how to apply NLP models for code summarization tasks. (Wang et al., 2021)	To ensure the platform generates accurate and informative summaries.
iv	Existing models lack the sophistication needed for generating accurate code summaries. (Wang et al., 2021)	To streamline summary generation by automatically generating up-to-date summaries that serve as valuable references for developers and stakeholders, thereby improving the overall quality of code summarization .

1.7 Proposed Solution

To address the challenges outlined in the problem statement, the proposed project offers a comprehensive solution that explores the working of two deep learning models. The project aims to enhance code summarization by leveraging the suitable of the two models (Llama 3 and CodeBERT). These models are equipped with the architecture that understands the syntax as well as semantics of the code snippet given as the input. They are fine-tuned on the python and Java dataset. They are analyzed based on which model

generates effective summaries. The effective model is then integrated into a web-based platform so that software developers can easily use this model to generate summaries of code snippets as well as perform error detection, code comment provider, and customization.

1.8 Scope of this Project

Our approach utilizes the power of two deep learning models i.e. Llama 3 and CodeBERT. It involves fine-tuning and integration of the effective of the two into a user-friendly interface to facilitate the interaction with the summarization models. The tool aims to generate concise and informative summaries for Java and Python code snippets. However, the scope excludes the direct modification of the model, focusing instead on its accuracy enhancement and its integration in a user-friendly web-based platform. The primary goal is to demonstrate the effectiveness and potential of these models in improving code summarization accuracy and comprehension.

1.9 Tools and Techniques

Different tools and techniques are used in the project, and are mentioned below in Table 1.1.

Table 1.2: Tools and Technologies

Tools And Technologies	Tools	Version
	VS code	1.89
	Google Colab	-
	Kaggle Notebook	3.9
	Technology	Version
	Python	3.9
	FastAPI	0.27

Chapter 2 LITERATURE REVIEW

Program understanding is invaluable in software development and maintenance because programs must be sufficiently understood before they can be properly modified. It is reported that, on average, software developers spend around 58% of time on program comprehension related activities (Xia et al., 2018). Numerous studies show that descriptive summaries, which explain the functionality of code snippets by brief natural language sentences, are conducive to software comprehension (Stapleton et al., 2020). There are many approaches used to generate natural language summaries of a code snippets.

In this chapter, we will explore the techniques that can be employed for code summarization as well as examine their associated drawbacks. Additionally, we will delve into the realm of Deep Learning, exploring its techniques and highlighting the reasons for its preference over traditional machine learning methods. Furthermore, we will discuss the existing research conducted in this domain. Finally, we will provide detailed information regarding the dataset utilized in our project, outlining its key characteristics and relevance to our study.

2.1 Literature Review

Keeping in view the importance of automatic code summarization, various machine learning approaches have been proposed for it. Machine learning techniques that can be used for code summarization are LSA (Latent Semantic Analysis), Clustering algorithms, Topic models like LDA (Latent Dirichlet Allocation), parts-of-speech tagging and Code cloning-based methods. Latent Semantic Analysis (LSA) is a method that generates a set of concepts related to a set of documents and the terms they contain and analyzes the relationships between them. It can be used to extract semantic information from source code (Liu et al., 2017). Clustering Algorithms can group similar pieces of code together based on certain features or characteristics. These algorithms can help in organizing and summarizing code segments effectively (Liu et al., 2017). Topic models like Latent Dirichlet Allocation (LDA) can be used to identify the topics within a collection of documents. By applying the topic models to source code, researchers can predict and generate method summaries (Eddy et al., 2013). Parts- of- Speech Tagging is a technique used to assign grammatical categories (such as noun, pronoun, adjective, etc.) to words

in a sentence. By identifying keywords that best represent the features of a source code through parts-of-speech tagging, code summarization can be improved (Wang et al., 2015). Code Cloning-Based methods involve searching for similar code segments within a dataset and extracting comments associated with those segments to generate summaries for the provided source code. This method heavily relies on the quality and quantity of code snippets and corresponding comments in the dataset (Ou et al., 2016). These methods were a starting point in code summarization. These techniques do not provide the accuracy as compared to deep learning models but they are not obsolete. They can be used in conjunction with the deep learning models for code summarization task.

Deep learning is a branch of machine learning that utilizes Artificial Neural Networks inspired by the human brain. It enables computers to learn and make decisions from vast amount of data without explicit programming. Deep learning models consist of multiple layers of interconnected artificial neurons, allowing them to learn complex patterns and relationships within data. It has achieved remarkable success in various domains, including computer vision and Natural Language Processing, and relies on large datasets and computational power. Convolutional Neural Networks (CNNs) are type of Artificial Neural Network and are used for image processing and recognition specifically designed to analyze pixel input. They are Artificial Intelligence (AI) systems for image processing that typically use machine vision which includes image and video recognition, as well as recommender systems and Natural Language Processing. They use Deep Learning to do both generative and descriptive tasks (Divya, P and Aiswarya, 2021). Recurrent Neural Networks (RNNs) were initially created to help with sequence prediction; the Long Short-Term Memory (LSTM) algorithm, for instance, is renowned for its adaptability. These networks solely use data sequences with different input lengths as their foundation. The RNN employs the knowledge gained from its prior state as an input value for the current prediction. Therefore, it can help a network to achieve short-term memory. RNN designs for problem analysis fall into two categories: Memory-based models called LSTMs are used to forecast data in temporal sequences. The three gates are input, output and forget. Gated RNNs are useful for temporal sequence prediction using memory-based data. The two gates are Update and Reset (Divya, P and Aiswarya, 2021). Transformer Model is widely used in Natural Language Processing for its parallel computation capabilities and effectiveness in capturing long-term dependencies in sequence. It overcomes the limitations of RNN parallelization and CNN feature extraction, making it a powerful tool for code summarization tasks (C. Zhang et al.,

2022). Graph Neural Network (GNN) are utilized to capture structural information and relationships in the source code, enabling more effective summarization by considering the code's graph representation (C. Zhang et al., 2022). There are several reasons why deep learning has gained popularity and is preferred in certain scenarios. Deep learning excels at capturing intricate patterns and relationships in large and high-dimensional datasets, making it suitable for complex data analysis. Deep learning models have shown exceptional performances while dealing with massive datasets, enabling effective handling of big data problems. These models can learn directly from raw input data to produce desired outputs, eliminating the need for manual feature engineering or preprocessing steps and can leverage knowledge gained from pre-training on large scale datasets, enabling efficient training on specific tasks with limited labeled data. Deep Learning can handle various data types including images, audio, video, text and sequential data, allowing it to be applied to a wide range of real-world problems.

The Distilled GPT model, which is simply the GPT-3.5 model trained using knowledge distillation (smaller model is capable to mimic the behavior of a large model) from the larger GPT-3.5 model. This smaller model is capable of generating code summaries of Java methods. The model can replicate the performance of the larger GPT-3.5 model for code summarization tasks and can be run locally. But it has some drawbacks. While small models can mimic the large model for code summarization, it may not perform as well across a wide range of tasks as the larger model does. The model's architecture and training process may be more complex and require careful tuning, potentially leading to higher computational costs and longer training times. There may be trade-offs in terms of accuracy, completeness and conciseness of the generated code summaries compared to human-written summaries of those produced by larger models (Su & McMillan, 2024).

The Llama 2 model is also a large language model developed for natural language processing tasks. It is pre-trained on large text data to learn language patterns and semantics, enabling them to understand and generate responses. They undergo fine-tuning process to adapt them to specific tasks, enhancing their performance and relevance in generating responses tailored to different contexts. This model may still exhibit biases present in training dataset, leading to biased and inappropriate responses in certain contexts (Touvron et al., 2023).

The SG-Trans model is an extension of the Transformer architecture. It introduces structure-guided self-attention and hierarchical structure-variant attention. It captures the

structural information into a self-attention mechanism. It uses token-guided, statement-guided, and data flow-guided self-attention heads to capture hierarchical code structures. The model allows to focus on local structures at shallow layers and global structures at deeper layers. It enables it to capture both local and global information of the code (Gao et al., 2023).

Graph Structure and Semantic Sequence for Code Summarization (GSCS) leverages both graph structure and semantic sequence information to generate concise descriptions for Java methods. The model includes a semantic encoder that utilizes a Bidirectional Gated Recurrent Unit (BiGRU) to build a language model on tokenized code sequence. This component focuses on capturing the semantic meaning of the code snippets. In addition to this, this model features a structural encoder that processes the tokenized Abstract Syntax Trees (ASTs) using multiple Graph Attention layers connected with BiGRU. This component encodes the structural information of the code snippets. This model aims to learn the conditional probability of generating a summary sequence based on these inputs (Zhou et al., 2022).

The CodeT5 model is a unified encoder-decoder model designed for code-related understanding and generation tasks. It is a transformer-based model and incorporates a novel identifier-aware pre-training objective to enhance code understanding. The model's performance has been validated through extensive experiments, demonstrating its capabilities in capturing code semantics and improving code-related tasks. It has some limitations. It is domain specific and performs better on the programming languages it is trained on rather than other ones. It hugely depends upon the training dataset (Wang et al., 2021).

The CoCoSUM model involves leveraging a Multi-Relational Graph Neural Network (MRGNN) to enhance code summarization by incorporating both intra-class and inter-class context. MRGNN encodes class relational embeddings by analyzing the inter-class relationships derived from UML class diagrams. By learning dependencies and connections between classes, it enhances the model's ability to generate accurate and contextually rich summaries. It incorporates class names such as intra-class context to generate class semantic embeddings. By considering class names within the same method, CoCoSUM captures the local context information that aids in generating more informative summaries. It analyzes inter-class relationships to extract additional context beyond the method's local scope. This broader context helps in understanding the

relationships between classes and enriches the summarization process. The evaluation and comparison were primarily done on Java set. Its evaluation focusses on specific design choices, such as incorporating global context, while other aspects of the model architecture such as token embeddings were kept constant. (Yanlin et al., 2021)

One approach combines a sequence of source code tokens with Convolutional Graph Neural Network (ConvGNN) to encode the ASTs of a Java method and generate natural language summaries. The model architecture is designed to leverage the structural information present in the AST, using the ConvGNN to encode the AST nodes and edges. The use of ConvGNN helps the model to determine when to directly copy tokens from the source code and create better representations of tokens in the AST. Leading to improved performance in code summarization task (LeClair et al., 2020).

The retrieval-based model combines an attention encoder-decoder neural network with retrieval components. The model retrieves the most similar code from the training set based on the syntactic and semantic level. These are incorporated into the attentional encoder decoder model in different scenarios to improve the generation of code summaries (J. Zhang et al., 2020).

In summary, the proposed approach presents a promising avenue for improving the accuracy of code summary generation through the integration of deep learning models.

The key findings and limitations of selected research papers are summarized below in Table 2.1. These papers collectively offer insights into various approaches to code summarization using cutting-edge technologies while acknowledging limitations such as dataset size, method comparison.

Table 2.1: Literature Review

Sr.	References	Methodology	Research Findings	Limitations
i.	(Su & McMillan, 2024)	Compare the performance of the distilled GPT model to human-written code summaries for Java methods.	70% of the functions aligned with the results, indicating consistency in model performance. GPT-3.5 produced summaries of higher quality than reference	Version and prompt given to the model as it is subject to change without notice, affecting results. Choice of java methods affects performance. May not perform as well across a wide range of tasks

			summaries.	
ii.	(Touvron et al., 2023)	Training the model. Check for safety violations across approximately 2000 adversarial prompts.	Outperforms existing open-source models. Handled safety violations.	Subjectivity in human evaluation. Biases in human evaluation.
iii.	(Gao et al., 2023)	Uses token-guided, statement-guided, and data flow-guided self-attention heads to capture hierarchical code structures.	Captures the structural information into a self-attention mechanism.	Evaluation relies on human annotations which can be influenced by the participant's programming experience and understanding of the evaluation metrics.
iv.	(Zhou et al., 2022)	Consists of a semantic encoder (for semantic information), a structural encoder (for structural information), and a decoder for generating accurate summaries.	Capture the structural as well as semantic information to generate accurate code summaries.	Redundancy can increase in the AST which can lead to noise, (when using multi-layered network).
v.	(Wang et al., 2021)	Pre-trained and then fine-tune paradigm for deriving generic language representations.	Outperforms prior methods in tasks like code defect detection, clone detection, code summarization etc.	Domain specificity, limited applications to a broader range of programming language. Training Data biases.
vi.	(Yanlin et al., 2021)	Leverage Multi-Relational GNN (MRGNN) by incorporating both intra-class and inter-class contexts.	By incorporating inter-class and intra-class context, quality and relevance of summaries is increased.	Only conducted on Java programming language. Training dataset can be biased which affects performance.

vii.	(LeClair et al., 2020)	Sequence of source code tokens is combined with the (ConvGNN)	ASTs are encoded to generate natural language summaries.	Lack of extensive hyper-parameter optimization which impacts the performance.
viii.	(J. Zhang et al., 2020)	Combines an attention encoder-decoder neural network with retrieval components.	Enhances the summarization process.	Relies on the training dataset (biased or noisy training dataset can hinder). May fail to capture the whole information of highly complex codebases.

In this comprehensive literature review in Table 2.1, a diverse range of methodologies and research findings in the critical domains of code summarization have been explored. Various innovative approaches, including the integration of deep learning techniques like neural networks, transformer-based models etc. have shown immense promise in enhancing code summary generation accuracy. However, several limitations, such as small datasets, limited method discussions, and potential biases, have been acknowledged across these studies. These findings collectively underscore the need for further research, larger datasets, and rigorous evaluations to advance these technologies.

CHAPTER 3 Data Analysis

3.1 Data Analysis

Our project focuses on addressing the challenges of accurate generation of code summaries. To achieve this, we propose a novel approach that enhances the capabilities of two deep learning models; Llama 3 and codeBERT. Data analysis involves examining and processing different types of data to extract meaningful insights and improve the performance in summarizing the code. It encompasses code corpus analysis, training data preparation, evaluation data analysis, performance monitoring and analysis, and model performance analysis. Code corpus analysis is conducted to understand code distribution, common patterns and syntax in the code corpus. Training data is prepared by preprocessing and cleaning the code corpus, extracting features and creating training examples. Evaluation data analysis examines user behavior and preferences through collected data such as search queries and feedback.

3.1.1 Dataset Used

CodeSearchNet is a dataset utilized in our study, renowned for its extensive collection of code snippets sourced from various open-source repositories (https://huggingface.co/datasets/code-search-net/code_search_net/tree/main/data). It includes Python, Java, and a number of other programming languages. Each code snippet in the dataset is supplemented with valuable metadata like its origin repository, associated functions and contextual information, enabling streamlined code summarization task. However, the data's sheer size, standing at a substantial 6 GB, posed a challenge in terms of manageability. So, we opted to narrow down our focus solely on Python and Java subsets within the dataset.

Table 3.1: Dataset Used in Code Summarizer

Dataset	Programming Languages	Size	Training Data	Testing Data	Validation Data
CodeSearchNet	Python	50000	35000	10000	5000
	Java	50000	35000	10000	5000

Table 3.1 presents the dataset split for source code summarization. It outlines the number

of samples available for Java and Python and their distribution among training, validation and testing sets.

3.2 System Requirements

The system requirements for the website can vary depending on the specific details and functionalities of the website. Some of them are as follows:

3.2.1 Hardware Requirements

A modern multi-core processor (e.g., Intel Core i5 or higher) for efficient computation. Sufficient RAM to handle data processing and model training tasks. A minimum of 8 GB is recommended, but more may be beneficial. Adequate storage space to store the application code, dataset, and any additional resources. SSD (Solid State Drive) is recommended for faster read/write operations. While not essential, having a dedicated GPU can significantly accelerate the training of deep learning models. NVIDIA GPUs, such as the GeForce series, are commonly used for this purpose.

3.2.2 Software Requirements

Software requirements outline the functionalities, features and constraints that a system or application must exhibit to meet user's needs and organizational objectives. The Code summarizer can be developed and deployed on various operating systems including Windows, macOS and Linux distributions. The implementation of the code summarizer involves Python language. Suitable code editors or IDEs such as Visual Studio Code are necessary for code development, debugging and version control. An internet connection may be required for accessing external APIs and cloud-based resources.

3.2.3 Clients, Customers and Users

The system benefits software development companies, developers, open-source communities, project managers, software documentation writers, researchers and academics. Companies can integrate the summarizer into development workflows and provide it as a tool to their development teams. Individual developers can gain insights into unfamiliar code and leverage existing code more efficiently. Open-source communities heavily rely on code reuse so, they can improve their understanding of existing code and identify relevant code snippets for their projects using the summarizer. Project managers can ensure the availability of an advanced tool for code understanding, leading to improved project outcomes and reduced development time. Software

documentation writers can gain insights into the code functionality which would help them to produce high quality documentation that accurately represent their codebases. Researchers and academics can explore the techniques and methodologies used in the summarizer, further advance the field and contribute to future enhancements and research directions.

3.3 Functional and Data Requirements

Summarize large amount of code snippets within a paragraph and return accurate and contextually relevant results. Capture the intent and purpose of code snippets, recognizing the relationships between different code element and their functionalities. Understand control flow, variable dependencies and the overall code structure.

A diverse dataset of code snippets covering various programming languages, domains, and complexity levels in JSON format is required for training and evaluation purposes. The dataset should include a sufficient number of code examples to ensure the model's generalizability and effectiveness across different contexts. The dataset used for training the models should be annotated with relevant metadata, such as code summaries.

3.4 Non-Functional Requirements

The system's effectiveness and viability hinge on several critical aspects that collectively determine its success. Performance is a foundational pillar, as the system must operate swiftly and efficiently to meet the demands of the stakeholders. Equally paramount is privacy and security, ensuring that data remains safeguarded against breaches. Scalability is another key concern, allowing the system to accommodate growing volumes of data and users. Accuracy is also a primary concern as the result should align with the user's intel and requirements.

3.4.1 Look and Feel Requirements

Within this system, some integral components play pivotal roles. The user interface (UI) governs how users interact with the system, ensuring a user-friendly and intuitive experience. The flow guides users through various processes, optimizing their engagement. Finally, deep learning model is the core of the system, driving accurate results. These elements collectively shape a comprehensive summarizer.

3.4.2 Usability Requirements

The summarizer excels in several critical aspects. The user-friendly interface should be

intuitive and self-explanatory, requiring minimal training or prior knowledge to navigate and perform tasks. It should be efficient, delivering accurate summaries with minimum manual efforts. It should be effective in helping developers understand code functionality with accurate results. It should be designed to prevent errors through clear guidance and input constraints. It should be able to recover from errors with clear messages and easy correction options. It should be consistent in design and behavior across the entire system. It should be open to user feedback for ongoing improvement.

3.4.3 Security Requirements

The summarizer should ensure secure data transmission with encryption to prevent eavesdropping. It should maintain user data privacy by adhering to regulations and securing user information. It should securely store sensitive data through encryption and access controls. It should ensure robust logging and monitoring to track activity, detect breaches, and respond to incidents.

CHAPTER 4 DESIGN CONSIDERATIONS

4.1 Design Constraints

The development of the code summarizer needs to consider several design constraints. Resource limitations like hardware, software, and developer availability can impact features and performance. Additionally, the quality and amount of training data can affect the accuracy of the summarization. Compatibility with existing tools and user interface design for various devices and accessibility needs are also crucial aspects during development.

To ensure a smooth user experience, the system should prioritize fast response time and provide clear documentation and support resources. By carefully considering these constraints, the code summarizer can be designed and deployed effectively, aligned with project goals and user needs.

4.2 Architectural Strategies

We proposed a robust approach for source code summarization, utilizing deep learning model. The code summarizer leverages several key architectural strategies to deliver robust and user-friendly experience. Firstly, by adopting a scalable architecture, the system can effectively handle increasing amounts of code and user traffic without compromising performance. This ensures smooth operation even as the user base or code corpus grows. Secondly, the system is designed with modularity in mind. This means different functionalities like code processing, summarization algorithms, and user interface components are built as modules. This allows for easier maintenance, future upgrades, and potential integration of new features. Furthermore, the architecture prioritizes flexibility. This allows for customization based on specific user needs or project requirements. The system might be adaptable to different programming languages or summarization techniques without needing a complete overhaul. This integration fosters a more comprehensive and efficient development environment. Figure 4.1 represents the architecture of the code summarizer.

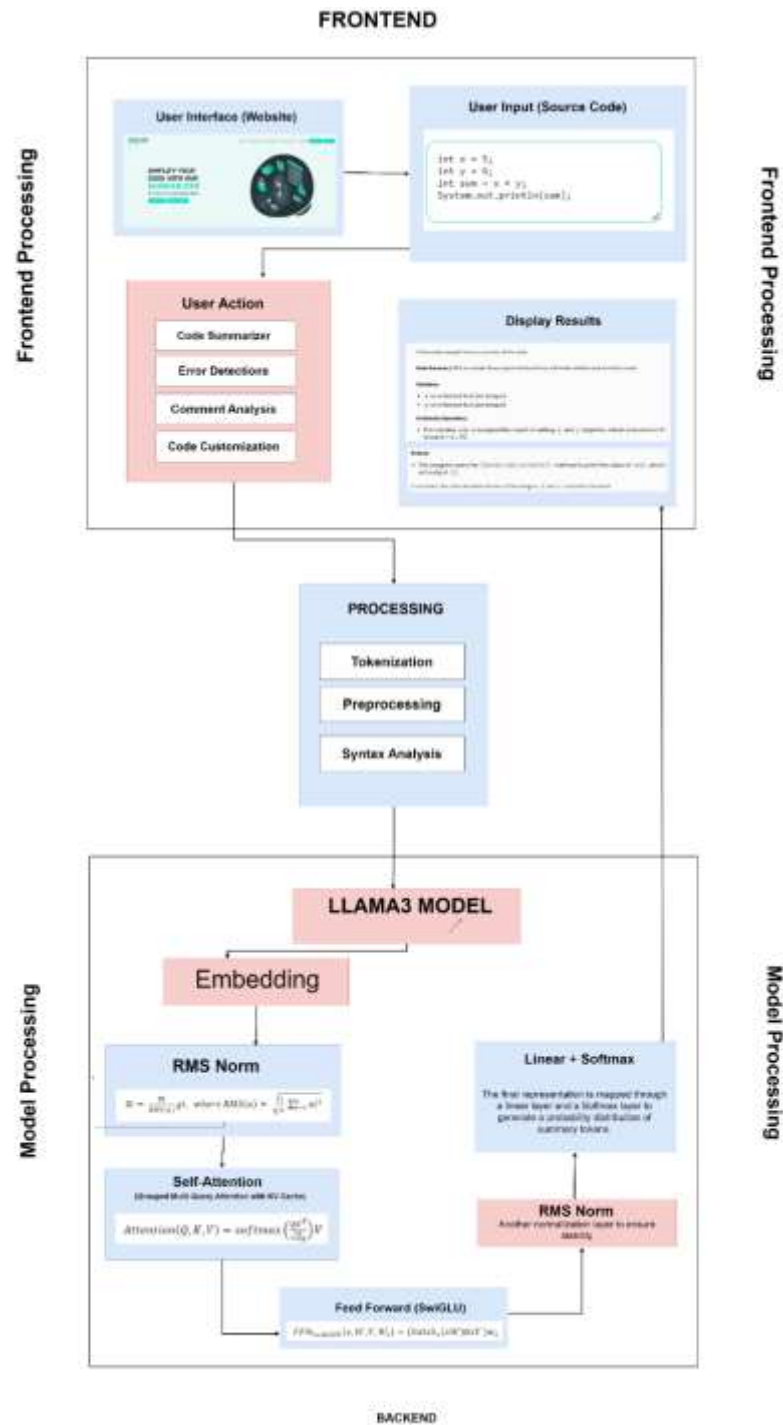


Figure 4.1: Architecture Diagram

It provides an overview of the architecture and workflow for a code summarization system using the LLaMA model. It is divided into three main sections: Frontend, Processing, and Backend.

4.2.1 Frontend

The user interface (website) provides an interface for users to interact with the system. Users can input their source code in languages like Java or Python into a text area.

Various actions can be performed through buttons or options, such as generating a summary of the input code (Code Summarizer), identifying and highlighting errors (Error Detections), analyzing and summarizing comments (Comment Analysis), and customizing the summary or other outputs (Code Customization). The results of these actions, such as the generated summary, detected errors, or analyzed comments, are displayed to the user.

4.2.2 Processing

Processing acts as a bridge between the frontend and backend, handling user inputs and preparing them for processing by the backend. The input code undergoes tokenization, converting it into smaller units (tokens). Preprocessing cleans and normalizes the tokens by removing comments and handling whitespace. Syntax analysis then analyzes the structure of the code to create an abstract syntax tree (AST).

4.2.3 Backend

The LLaMA model serves as the core deep learning model used for code summarization and other analyses. It converts tokens or AST nodes into high-dimensional vectors (embedding) that the model can process. Root Mean Square Normalization (RMS Norm) is applied to stabilize training and improve performance. The model uses self-attention (Grouped Multi-Query Attention with KV Cache) to focus on different parts of the code, understanding context and dependencies efficiently. Rotary positional encodings add positional information to embedding to capture the sequential nature of the code. The feed-forward network (SwiGLU) processes the attention outputs and introduces non-linearity to capture complex patterns. Another RMS normalization layer ensures stability. The final representation is mapped through a linear layer and a Softmax layer to generate a probability distribution over possible summary tokens.

4.2.4 Overall Workflow

The overall workflow begins with the user interface, where the user inputs their source code and selects an action, such as summarization. The input code undergoes tokenization, preprocessing, and syntax analysis in the processing stage to prepare it for the backend model. The preprocessed code is converted into embedding and passed through several layers of the LLaMA model, including self-attention, positional encodings, and feed-forward networks. Normalization layers ensure stable processing. The final processed representation is mapped through a linear layer and a Softmax layer to generate a summary or other outputs. The results, such as the code summary, are then displayed to the user on the frontend. This architecture allows for efficient and accurate

summarization and analysis of source code, leveraging advanced deep learning techniques to understand and process the input code.

4.3 Methodology

In this research, we applied two deep learning models; one is Llama 3 and the other is CodeBERT to accurately generate summaries. This process can be broken down into several key stages, each contributing to the overall functionality and effectiveness of the final system. The first stage involves meticulous requirement analysis. This entailed gathering user input, pinpointing crucial features, and establishing clear project objectives. Next, data collection and preprocessing were done. The collected data underwent the cleaning process and prepared for further processing. Next, model training was done. Deep learning models called Llama 3 and CodeBERT were trained on the preprocessed code corpus. A crucial step was model evaluation. The model's performance was measured using the specific metrics.

With a well- trained model in place, system design and development were done. This stage involved planning the system architecture and design, considering the chosen model that gave more performance accuracy after being evaluated, user interface needs, and desired functionalities. The development process itself focused on implementing all the necessary components. Finally, it was deployed to a production environment. Proper monitoring and maintenance practices were established to ensure the system's smooth operation over time.

Figure 4.2 illustrates the methodology used for the development of code summarizer. The processes involve requirement analysis, data collection and preprocessing, model selection and training, model validation and evaluation, system design and development, deployment and maintenance.

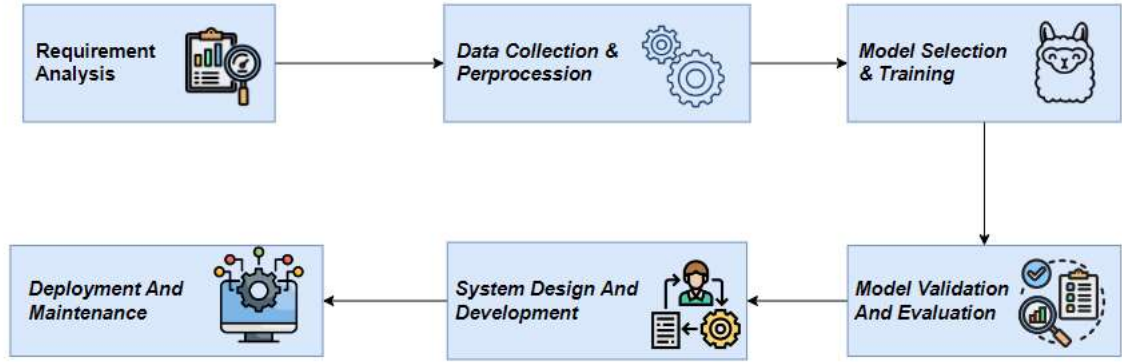


Figure 4.2: Methodology

4.3.1 Requirement Analysis

the project aims to develop a tool that summarizes Java and Python code, incorporating functionalities like error detection, code comment generation, and customization. The primary stakeholders include end-users who will interact with the summarizer, developers involved in the project, and project supervisors overseeing the progress. The core functionalities include taking code as input and providing a concise summary, identifying and highlighting errors, generating appropriate comments, and allowing users to customize the output. Additionally, the system must meet performance requirements such as fast response times and scalability, usability requirements for an intuitive interface, reliability for robust error handling, and security for data privacy and user authentication. The technical requirements involve specific hardware and software, utilizing tools like Google Colab for model fine-tuning and FastAPI for backend development.

4.3.2 Data Collection and Preprocessing

The initial dataset of CodeSearchNet (with the size of 6GB) with various programming languages presented manageability challenges for our project with limited storage and resources. So, we used 50,000 entries each from Python and Java, divided into training, testing and validation sets for both programming languages (https://huggingface.co/datasets/code-search-net/code_search_net/tree/main/data). The dataset underwent preprocessing steps like tokenization (splitting text into individual words or tokens), code filtering (filtering out noisy data that would affect the performance), code cleaning (removing whitespaces), and formatting prompts consisting of instructions for better handling within our system.

4.3.3 Model Selection & Training

Based on the task requirements, we selected Llama 3 and CodeBERT as the deep learning models for this study. The models were trained on the training dataset of the two programming languages. The training process involves giving preprocessed input to the model, optimizing model parameters, and updating the model weights. The training process aimed to minimize the models' error and improve their performance.

4.3.4 Model Evaluation and Validation

The trained models were evaluated using appropriate standard evaluation metrics (e.g. Faithfulness Metrics, Contextual Relevancy, Contextual Recall, Contextual Precision). Faithfulness Metric evaluates whether the information in the summary is accurate and true to the original code (check for any distortion or incorrect interpretations in summary). Contextual Relevancy Metric measures how well the generated summary captures the main ideas of the original code (summary is related to the most important parts of the code). Contextual Recall Metric measures the proportion of the relevant information from the original code that is included in the summary (summary covers all the necessary parts of code). Contextual Precision Metric measures the proportion of the summary that is relevant and accurately reflects the original code (summary does not include irrelevant or extraneous information).

4.3.5 System Design & Development

The architecture includes a frontend developed with HTML, CSS, and JS, and a backend using FastAPI and Python, integrating machine learning models Llama 3 and CodeBERT. The detailed design involves creating an intuitive user interface with Figma mockups, outlining UI elements and their functionalities. The backend design focuses on API endpoints, data flow, and processing logic, ensuring seamless integration with the ML models. The database design includes a schema and entity-relationship diagrams. The development plan encompasses a timeline with milestones, task allocation, and the chosen development methodology, such as Agile or Scrum. Implementation details cover the step-by-step process, code snippets, and solutions to challenges encountered.

4.3.6 Deployment & Maintenance

The strategy involves setting up development, staging, and production environments using cloud services and CI/CD pipelines, outlining the steps for deploying the application. Regular maintenance activities include backups, updates, and monitoring, with a system for handling bug reports and feature requests. Performance monitoring

and optimization are crucial for maintaining efficiency. User training and documentation are provided through manuals, and guides, , ensuring users understand how to use the tool effectively. Developer documentation aids in future maintenance and updates. Evaluation and feedback methods are established to collect user input and measure the project's success, with plans for future improvements based on this feedback.

4.4 Deep Learning Model

In this study, we applied the Llama 3 and CodeBERT as the deep learning models for source code summarization.

4.4.1 Llama 3

The Llama 3 is a Large Language Model (LLM) known for its abilities in handling complex tasks. It boasts significant improvements in handling complex tasks, response accuracy, and diversity. It also excels at reasoning, code generation, and following instructions. It is based on the Transformer architecture. Despite its large parameter size (8 billion and 70 billion configurations), the LLAMA3 model has outperformed even larger models like GPT-3 (175 billion parameters) in certain tasks. Its superior performance makes it a highly competitive option for a wide range of natural language processing applications (Wei Huang et al., 2024). Figure 4.3 illustrates the architecture of the Llama 3 model.

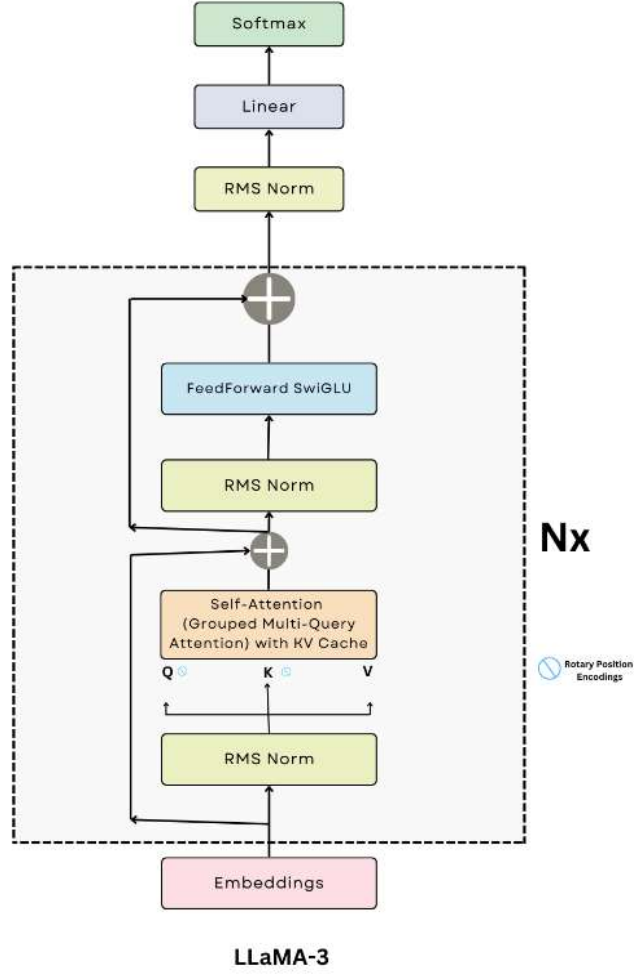


Figure 4.3: Llama 3 Architecture

Its architecture consists of the following components:

- i. **RMS Norm:** Root Mean Square Normalization (RMSNorm) is a novel normalization method. It normalizes activations using the root mean square of the activations instead of relying on mini-batch or layer statistics as shown in Equation (1). This ensures consistent scaling of activations regardless of mini-batch size or the number of features. RMSNorm also includes learnable scale parameters, similar to Batch Normalization, providing adaptability.

$$\underline{ai} = \frac{ai}{RMS(a)} gi, \text{ where } RMS(a) = \sqrt{\frac{1}{n} ai^2} \quad (1)$$

In this equation, ai represents an individual activation or feature in the neural network layer. $RMS(a)$ is the root mean square of the activations, which measures the magnitude of the activations. It's computed by summing the squares of all activations ai dividing by the total number of activations, and taking the square root.

The fraction $\frac{ai}{RMS(a)}$ normalizes each activation ai by dividing it by the RMS

value, ensuring consistent scaling of the activations. This normalization helps prevent issues like vanishing or exploding gradients during training.

g_i is a learnable scale parameter associated with each activation. After normalization, the scale parameter g_i allows the model to adjust the normalized activation, providing flexibility and adaptability to the network.

Finally, $\underline{a_i}$ is the output after normalization and scaling, which is then passed to the next layer in the neural network. This process maintains controlled activation magnitudes while still allowing the network to learn optimal scaling through the learnable parameters g_i .

ii. Grouped Multi-Query Attention: Grouped-query attention (GQA) is an interpolation of multi-query and multi-head attention. It achieves a quality similar to multi-head attention while maintaining a comparable speed to multi-query attention. It uses only a single key-value head for multiple queries, which can save memory and greatly speed up decoder inference. Llama incorporates the (GQA) to address the memory bandwidth challenges during the autoregressive decoding of Transformer models.

iii. KV Cache: At every step of the inference, we are only interested in the last token output by the model, because we already have the previous ones. However, the model needs access to all the previous tokens to decide on which token to output, since they constitute its context (or the “prompt”). It’s a way to make the model do less computation on the token it has already seen during inference. Its equation is shown in Equation (2).

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2)$$

where

Q (Query): This is a matrix derived from the current token's hidden state. It represents what the current token is "querying" or looking for in the context of the previous tokens.

K(Key): This is a matrix derived from all the previous tokens. It represents the features or characteristics of each previous token.

V(Value): This is a matrix also derived from all the previous tokens. It represents the actual information content associated with each previous token.

QK^T : This is the matrix multiplication of Q and the transpose of K . The result is a matrix that represents the similarity or alignment between the current token's query and each of the previous tokens.

$\sqrt{d_k}$: This is the scaling factor, where d_k is the dimensionality of the key vectors. Dividing by $\sqrt{d_k}$ helps to stabilize the gradients during training by preventing the dot products from becoming too large.

$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$: The softmax function is applied to the scaled dot products, converting them into a probability distribution. This indicates how much attention the current token should pay to each of the previous tokens.

V : After calculating the attention weights through softmax, these weights are used to compute a weighted sum of the value vectors V . This gives the final output of the attention mechanism, which is a context-aware representation of the current token.

In the context of the KV Cache, during inference, the model reuses the previously computed K and V matrices and only computes Q for the new token. This significantly reduces the amount of computation needed, as the model doesn't need to recompute K and V for tokens it has already seen.

iv. SwiGLU Activation Function: SwiGLU is an activation function used to calculate the output of a neuron in a neural network by taking in the weighted sum of the input and applying a non-linear function to it as shown in Equation (3). It has several benefits that make it a useful activation function in neural networks. First, it is based on the GLU concept, which has been shown to perform well in many applications. Second, it uses the Swish function, which has been shown to outperform other activation functions in some cases, particularly when combined with residual connections. Third, it allows for efficient computation due to its use of element-wise multiplication.

$$FFN_{\text{SwiGLU}}(x, W, V, W_2) = (\text{Swish}_1(xW) \odot xV)w_2 \quad (3)$$

The equation represents the SwiGLU activation function in a neural network, combining the Swish activation and Gated Linear Unit (GLU) mechanisms.

Here, x is the input vector to the feedforward neural network (FFN), which could

be either the output from the previous layer or the original input data. W is a weight matrix that, when multiplied by the input x , produces an intermediate representation xW . The Swish function is then applied to xW . The Swish function, defined as

$$\text{Swish}(z) = z \cdot \sigma(z) \text{ (where } \sigma(z) = \frac{1}{1 + e^{-z}} \text{)}$$

is the sigmoid function, introduces non-linearity to the model. This non-linear transformation allows the network to learn more complex patterns.

Next, V is another weight matrix that transforms the input x into another intermediate representation xV . The element-wise multiplication \odot (Hadamard product) between the Swish-activated vector $\text{Swish}(xW)$ and xV implements the gating mechanism. This means that the values in xV modulate the values in $\text{Swish}(xW)$ allowing certain elements to be emphasized or suppressed based on their importance.

Finally, W_2 is a weight matrix that further transforms the gated output $(\text{Swish}(xW) \odot (xV))$ through matrix multiplication, yielding the final output of the layer. This last step refines the output and ensures that the network can effectively learn from the data.

- v. **FeedForward Block:** It plays a crucial role, typically following the attention layer and normalization. The feedforward layer consists of three linear transformations. During the forward pass, the input tensor x is subjected to multi-layer of linear transformations. The SwiGLU activation function, applied after first transformation, enhances the expressive power of the model. The final transformation maps the tensor back to its original dimensions. This unique combination of SwiGLU activation and multiple FeedForward layer enhances the performance of the model.
- vi. **Encoder Block:** It comprises of 32 layers each consisting of self-attention mechanism, feed forward network, and layer normalization.

4.1.1 CodeBERT

CodeBERT is a transformer-based model that utilizes a multi-layer bidirectional Transformer architecture. It is based on the BERT model (Bidirectional Encoder Representations from Transformers). The architecture enables the model to capture complex relationships and dependencies in both programming and natural language. With 125 million parameters, CodeBERT is well-equipped to handle a wide range of natural language processing tasks. By leveraging this advanced neural architecture,

CodeBERT achieves impressive results in fine-tuning for specific tasks including code summarization (Feng et al., 2020).

Its architecture, primarily consists of the input segment, encoder layers, Masked Language Modelling (MLM) and Replaced Token Detection (RTD).

The input representation consists of the following embeddings:

- i. **Token Embeddings:** CodeBERT takes both natural language (NL) and programming language (PL) tokens as input. Each token is embedded into a continuous vector space.
- ii. **Segment Embeddings:** To distinguish between different types of tokens (e.g., NL vs. PL), segment embeddings are added. For example, in a pair of NL and PL sequences, different segment embeddings are used for each.
- iii. **Position Embeddings:** These are added to the input embeddings to encode the position of each token in the sequence, allowing the model to understand the order of tokens.

CodeBERT uses multiple layers of transformer encoders. Each layer consists of:

- i. **Multi-Head Self-Attention Mechanism** The Multi-Head Self-Attention Mechanism enables each token in a sequence to attend to every other token, capturing contextual dependencies. This mechanism is key to understanding the relationships between tokens in the input sequence.

The self-attention mechanism is computed as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where:

- $Q = XW_Q$ (Query matrix)
- $K = XW_K$ (Key matrix)
- $V = XW_V$ (Value matrix)
- X is the input representation
- W_Q, W_K, W_V are the learned weight matrices
- d_k is the dimension of the key vectors

In Multi-Head Self-Attention, this process is repeated h times (with different learned weight matrices for each head) and the results are concatenated:

$$MultiHead(Q, K, V) = Concat(head1, head2, \dots, headh)W_O$$

- ii. **Feed-Forward Neural Networks (FFN):** After the attention mechanism, the output is passed through a Feed-Forward Neural Network (FFN), which is applied independently to each position:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Where:

- W_1 and W_2 are weight matrices
- b_1 and b_2 are bias vectors
- The ReLU activation function is applied element-wise.

- iii. **Add & Norm:** Each sub-layer (Multi-Head Self-Attention and FFN) is followed by a residual connection and layer normalization. This helps in stabilizing the training process and preserving the input information:

$$\text{Output of Add \& Norm} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

Where:

- x is the input to the sub-layer
- $\text{Sublayer}(x)$ represents the output of either the self-attention mechanism or the FFN.
- Layer Norm represents the layer normalization function.

These components collectively allow CodeBERT to effectively model the relationships between tokens in source code and comments, enabling it to perform tasks like code summarization, generation, and comprehension.

In Masked Language Modelling (MLM), some tokens in the input sequence are randomly masked, and the model is trained to predict these masked tokens based on their context as shown in Equation (4). Given a data point of NL-PL pair ($x = \{w, c\}$) as input, as shown in Equation (4) and Equation (5), where w is a sequence of NL words and c is a sequence of PL tokens, we first select a random set of positions for both NL and PL to mask out (i.e. m_w and m_c , respectively), and then replace the selected positions with a special [MASK] token as shown in Equation (6) and (7). 15% of the tokens from x are masked out as shown in Equation (8).

$$M^w_i \sim \text{unif}\{1, |w|\} \text{ for } i = 1 \text{ to } |w| \quad (4)$$

The above equation defines M_w which is a random variable that is uniformly distributed over the range $\{1, 2, \dots, |w|\}$. Here, $|w|$ represents the length of the sequence w (the total number of elements or tokens in w). For each index i from 1 to $|w|$ M_{wi} is assigned a random value within this range.

$$m^c_i \sim \text{unif}\{1, |c|\} \text{ for } i = 1 \text{ to } |c| \quad (5)$$

This equation defines a new sequence w^{masked} which is created by replacing elements in the sequence w at positions indicated by M_w with a special token [MASK]. The $\text{REPLACE}(w, M_w, [\text{MASK}])$ function operates as follows: for each index i specified by M_w , the corresponding element in w is replaced with [MASK]. This is typically used in tasks like masked language modeling, where certain tokens are masked and the model is trained to predict them.

$$w^{\text{masked}} = \text{REPLACE}(w, m^w, [\text{MASK}]) \quad (6)$$

$$c^{\text{masked}} = \text{REPLACE}(c, m^c, [\text{MASK}]) \quad (7)$$

This equation is analogous to Equation (6), but for the sequence c . The new sequence c^{masked} is created by replacing elements in c at positions specified by M_c with [MASK]. The $\text{REPLACE}(c, m^c, [\text{MASK}])$ function performs the same operation as in the previous equation, masking specific elements in c .

$$x = w + c \quad (8)$$

This equation defines a new sequence x as the combination or concatenation of the sequences w and c . Depending on the context, the "+" symbol might represent concatenation (if w and c are sequences like strings or lists) or element-wise addition (if w and c are vectors of numerical values). In the context of NLP, it's common to concatenate sequences of tokens or characters, meaning x would simply be the sequence formed by joining w and c together.

The equations describe a process for masking specific parts of sequences w (likely a word or token sequence) and c (another sequence, possibly characters), and then combining these sequences into a new input x . Here's an explanation of each equation:

The objective of MLM is to predict the original tokens which are masked out, formulated as follows, where p^{D1} is the discriminator which predicts a token from a large vocabulary as shown in Equation (9).

$$L_{\text{MLM}}(\theta) = \sum_{i \in m_w \cup m_c} -\log p^{D1}(x_i | w^{\text{masked}}, c^{\text{masked}}) \quad (9)$$

In the above equation, w^{masked} is the result after applying the REPLACE function. It holds the new version of the string w , where all occurrences of m_w have been replaced by [MASK]. This is the final output after the replacement operation.

Replaced Token Detection (RTD) involves replacing some tokens with plausible alternatives and training the model to distinguish between original and replaced tokens as shown in Equation (12) and (13). There are two data generators here, an NL generator

p^{Gw} and a PL generator p^{Gc} , both for generating plausible alternatives for the set of randomly masked positions as shown in Equation (10), (11), and (14).

$$w_i \sim p^{Gw}(w_i | w^{\text{masked}}) \text{ for } i \in m^w \quad (10)$$

The above equation describes the prediction of masked elements in the sequence w . Here w_i represents the predicted value for the i th element of w , which was originally masked. The prediction is sampled from a probability distribution $p^{Gw}(w_i | w^{\text{masked}})$, which is the distribution over possible values of w_i given the masked version of the sequence w^{masked} . The index i belongs to the set m^w , which contains the positions of the masked elements in w .

$$c_i \sim p^{Gc}(c_i | c^{\text{masked}}) \text{ for } i \in m^c \quad (11)$$

This equation is analogous to Equation (10), but for the sequence c . Here, c_i represents the predicted value for the i th element of c , which was originally masked. The prediction is sampled from a probability distribution $p^{Gc}(c_i | c^{\text{masked}})$ which is the distribution over possible values of c_i given the masked version of the sequence c^{masked} . The index i belongs to the set m^c , which contains the positions of the masked elements in c .

$$w^{\text{corrupt}} = \text{REPLACE}(w, m^w, \hat{w}) \quad (12)$$

The above equation creates a "corrupted" version of the sequence w by replacing the masked elements at positions m^w with the predicted values \hat{w} .

$$c^{\text{corrupt}} = \text{REPLACE}(c, m^c, \hat{c}) \quad (13)$$

The above equation creates a "corrupted" version of the sequence c by replacing the masked elements at positions m^c with the predicted values \hat{c} .

$$x^{\text{corrupt}} = w^{\text{corrupt}} + c^{\text{corrupt}} \quad (14)$$

The above equation combines the corrupted sequences w^{corrupt} and c^{corrupt} into a single sequence x^{corrupt} typically through concatenation or element-wise addition.

The equations provided describe a process where masked elements in sequences w and c are

predicted and then used to create "corrupted" versions of these sequences, which are then combined.

4.2 Project Management Strategies

4.2.1 Work Division

The table summarizes the contributions of five individuals in a project. Kainat Farooqi worked on the Llama 3 model. She also contributed to testing and documentation chapters 1, 2, and 3 including formatting. Maqddus Butool focused on the design and development of the website as well as chapter 8. Safa Saeed focused on the CodeBERT model as well as the diagrams. Qirat Qadeer focused on the fine-tuning of the CodeBERT model as well as the development of the Login and Sign-Up page. Mehvish Zahid took part in finetuning the Llama 3 model as well as contributing to chapter 5 and 6 demonstrated in below Table 4.1.

Table 4.1: Project Management Strategies

Name	Deep learning Algorithms	Website Development	Documentation
Kainat Farooqi	Llama 3 model	Testing	Chapter 1,2,3
Maqddus Butool	–	Website design and development	Chapter 8
Safa Saeed	CodeBERT model	-	Diagrams
Qirat Qadeer	CodeBERT model	-	Chapter 4,7
Mehvish Zahid	Llama 3 model	Testing	Chapter 5,6

4.2.2 Task Management

The project is divided into several modules with corresponding timelines. Software Requirements will be completed by December 2023, followed by Software Analysis continuing until January 2024. Software Design is expected to be finished by February 2024, and Development will take place from February 2024 till April 2024. Testing and debugging are scheduled for May 2024, and the Final Result is anticipated in June 2024. The whole process management alongwith the time duration are summarized in Table 4.2.

Table 4.2: Time Management

Modules	Timeline
Software Requirement	Dec 2023

Software Analysis	Jan 2024
Software Design	Feb 2024
Development	Till April 2024
Testing and debugging	May 2024
Final Result	June 2024

4.2.3 Development Method

The project aimed to leverage the deep learning models, namely Llama 3 and CodeBERT to automate the process of code summarization or code comment generation. The objective was to address the issue of obsolete or outdated code comments to foster code reusability by software developers.

The dataset used in the project consisted of 50,000 code-comment pairs in Python and Java derived from the CodeSearchNet benchmark dataset. Each code snippet in the dataset is supplemented with valuable metadata like its origin repository, associated functions and contextual information, enabling streamlined code summarization task.

For the development of a code summarization tool, the model that gives higher accuracy was used. Some additional functionalities are also added such as error detection, comment provider as well as customization. Error detection functionality checks the pasted code for errors. Comment provideing functionality provides comments to the code snippets. Customization improves the code snippet that we provide.

4.2.4 Future Enhancement Plan

Future improvements include expanding the project's focus to summarize other programming languages present in the CodeSearchNet dataset. Additionally, other benchmark datasets like the CodeXGlue can also be explored. Continuous improvement of summarization process can be made to provide more accurate and relevant code summaries. This would involve incorporation of advanced deep learning models or Large Language Models like Claude. Advanced techniques are to be explored for better understanding of the semantics of the code snippets. This also incorporates deep learning algorithms to extract meaningful information of the code such as identifying code patterns, code relationships and code intent. Adding collaborative features to the code summarizer, such as the ability to share code snippets and collaborate with other developers. This would foster a community-driven approach, where developers can learn from each other. Continuously optimizing the performance of the code summarizer to ensure fast and efficient search results, especially while dealing with large code

repositories or complex queries. These enhancements can be done in future to further enhance our project.

CHAPTER 5 SYSTEM DESIGN

5.1 System Architecture and Program Flow

The implementation of an effective code summarizer involves several crucial components, each contributing to the system's overall success. Beginning with data acquisition and preprocessing, we used the Python and Java subsets of the CodeSearchNet consisting of code-summary pairs, ensuring the dataset's integrity and security. These pairs undergo essential preprocessing tasks, including tokenization, code filtering, code cleaning etc. to make them amenable for training deep learning models. In the next step the models are applied to this dataset. The system proceeds to evaluate query results by utilizing common evaluation metrics used for such models.

5.1.1 Data Acquisition and Preprocessing

Use the Python and Java subsets of the CodeSearchNet dataset. Perform necessary preprocessing tasks on the dataset, such as tokenization, code filtering, code cleaning etc. to prepare them for training the deep learning models as shown in Figure 5.1.



```
[ ] !pip install datasets
from datasets import load_dataset, load_from_disk, save_to_disk

training_dataset = load_from_disk("/content/drive/MyDrive/llm_dataset/
code_search_net/merged_train") #split = "train"

def select_columns(example):
    # Return only the desired columns
    return {
        'language': example['language'],
        'func_documentation_string': example['func_documentation_string'],
        'whole_func_string': example['whole_func_string']
    }

# Apply the function to the dataset
filtered_dataset = training_dataset.map(select_columns, remove_columns=[col
for col in training_dataset.column_names if col not in
['language', 'func_documentation_string',
'whole_func_string']])
```

Figure 5.1: Data Pre-processing

5.1.2 Training Deep Learning Models

Train the Llama 3 and CodeBERT model using the pre-processed dataset. Evaluate the model on the test data using appropriate evaluation metrics. Integrate the accurate model to leverage its strengths for accurate code summary generation.

5.2 Program Flow

Users interact with the website, inputting the Python or Java code to be summarized in the editor. The uploaded code undergoes cleaning and tokenization to prepare it for input into the model. The pre-processed code is then fed into trained model to generate accurate code summaries. The website interface presents the final output or the summary, aiding in code comprehension. Along with this functionality, the code summarization tool provides additional functionalities like error detection, comment provider, and customization.

5.2.1 DFD Level 0

At Level 0 of the Data Flow Diagram (DFD) depicted in Figure 5.2, the user interacts with the website by providing input, including the code to be summarized. The website then forwards this input to the summarizer module, where the model analyzes the data to generate code summary. This seamless process ensures that the system efficiently processes user input and produces accurate summaries, enhancing the application's usability and effectiveness in aiding code comprehension



Figure 5.2: DFD level 0

5.2.2 DFD Level 1

At Level 1 of the Data Flow Diagram (DFD) depicted in Figure 5.3, the user's input, including the code to be summarized, is processed by the application. The data undergoes pre-processing for analysis by the Llama 3 model in the Deep Learning Model module. Integrated summaries are generated by leveraging the strengths of the model in the Model Integration module, aiding in the generation of accurate summaries. The final output or summary is displayed on the user-friendly website interface in the Result Display module.

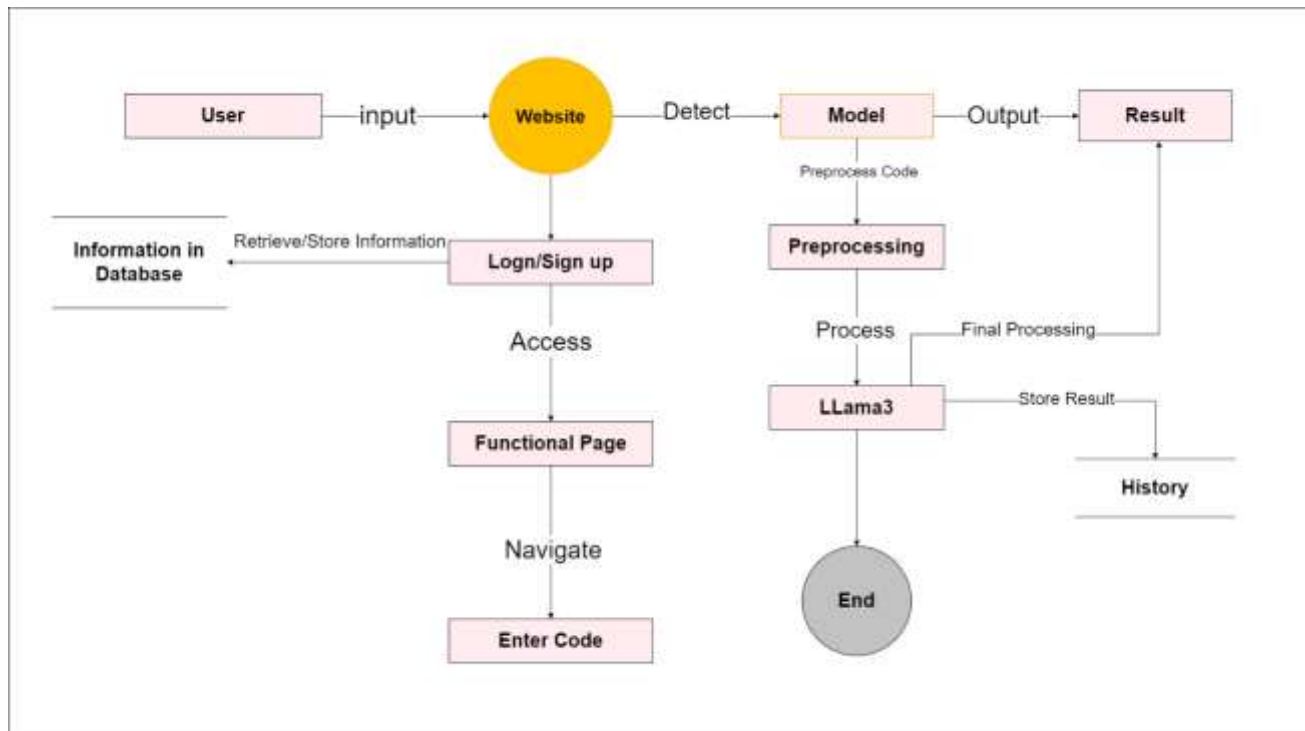


Figure 5.3: DFD level 1

5.3 Detailed System Design

The User Interface (UI) component enables user-friendly interaction with the website. The Database component securely manages the user's data. The Deep Learning Model component implements the Llama 3 model for code summary generation as well as error detection, comment provider, and customization. Model Training and Evaluation handles preprocessing and ensures accurate generation of summaries. System Integration establishes seamless communication between model and the website. Security and Privacy component addresses regulatory compliance. Performance Optimization improves system efficiency. Documentation and Maintenance ensure comprehensive records and future updates. There is also a facility of user feedback. The user's identity is maintained by the use of a database.

These components collectively contribute to the development and maintenance of the system, enabling user interaction, efficient data management, accurate predictions, and compliance with regulations.

5.4 Use Case Diagram

Below Figure 5.4. is the use case diagram of our project having two actors that are User and System. They have privileged access for user management, system configuration, and ensuring the overall system's integrity and security.

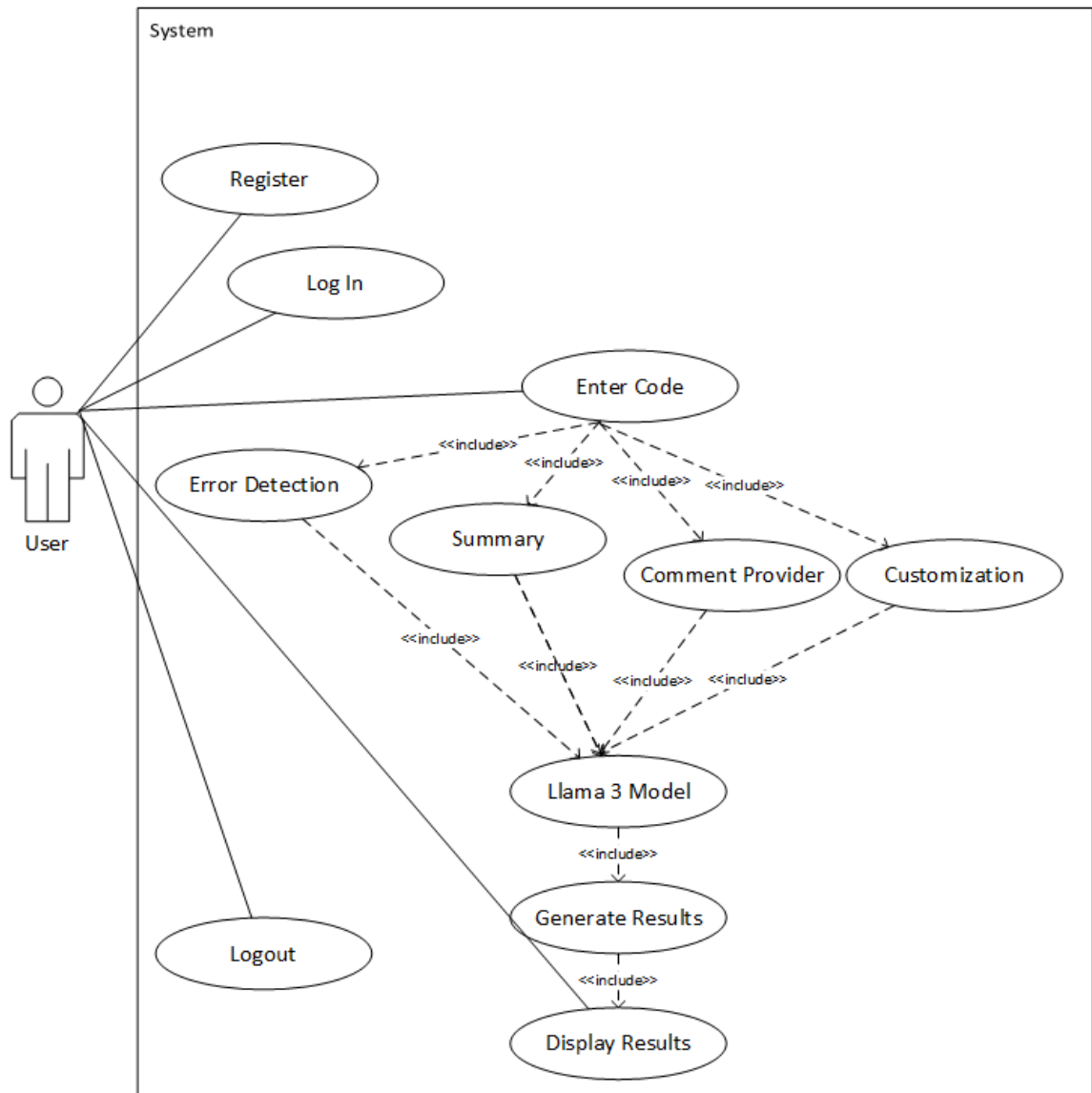


Figure 5.4: Use Case Diagram

5.4.1 Use Cases Description

Enter Code Use Case

In “Enter Code” use case (UC001) detailed in Table 5.1, users are prompted to enter a code snippet in python or java.

Table 5.1: Enter Code: UC1 1

Use Case ID	UC001
UC Name	Enter Code
Description	Code snippets in Python and Java to be summarized.
Actors	Users
Preconditions	The user is authenticated and authorized to access the website.
Postconditions	The code is successfully entered in the code editor.

Flow of Events	User copies the code to be summarized and pastes it in the summarizer page.
Exceptions	- Upload failure due to technical issues.

Register Use Case

“Register” function (UC002) detailed in Table 5.2. outlines the registration of the user on the website.

Table 5.2: Register: UC1 2

Use Case ID	UC002
Use Case Name	Register
Description	As a system, I want to register a user to the website to keep their record in a database.
Actors	User, System
Preconditions	User is not already registered.
Postconditions	User is registered successfully.
Flow of Events	<ol style="list-style-type: none"> 1. The user goes to the “Sign Up” page. 2. The user enters his credentials and press the register button. 3. The data of the user is stored in the SQL database.
Exceptions	- Failure due to technical issues.

Login Use Case

“Login” function (UC003) detailed in Table 5.3. outlines the login procedure of the user after successfully registering on the website.

Table 5.3: Log In: UC1 3

Use Case ID	UC003
Use Case Name	Log In
Description	As a system, I want to log a user in the website for maintaining history and seamless experience.
Actors	User, System
Preconditions	User is already registered.
Postconditions	User is logged in successfully.
Flow of Events	<ol style="list-style-type: none"> 4. The user goes to the “Log In” page. 5. The user enters his credentials and press the log in button. 6. The system checks whether the user is registered or not. 7. If yes, the user is logged in successfully.
Exceptions	- Failure due to technical issues.

Summary Function Use Case

In "Summary" use case (UC004) detailed in Table 5.4, describes how our Llama 3 model

analyzes preprocessed code snippets for summarization. The system fetches and processes the data, summarizes it and stores the results.

Table 5.4: Summary Function: UC1 4

Use Case ID	UC004
Use Case Name	Model Analysis
Description	Llama 3 model generates summary.
Actors	System
Preconditions	Preprocessed data is available for analysis.
Postconditions	Summary is generated.
Flow of Events	1. The system fetches the preprocessed code snippet. 2. The system passes the data through the Llama 3 model for analysis. 3. The CNN model summarizes the code.
Exceptions	- Model analysis failure due to technical issues.

Error Detection Use Case

In "Error Detection" use case (UC005) detailed in Table 5.5, outlines how error detection functionality is performed.

Table 5.5: Error Detection: UC1 5

Use Case ID	UC005
Use Case Name	Error Detection
Description	As a system, I want to detect error in the source code that is entered.
Actors	System
Preconditions	Preprocessed data is available for analysis.
Postconditions	Error is detected.
Flow of events	1. The system fetches the preprocessed data. 2. The system detects error present in the source code.
Exceptions	- Error detection failure due to technical issues.

Comment provider Function Use Case

"Comment provider" use case (UC006) detailed in Table 5.6, outlines the comment provider functionality.

Table 5.6: Comment provider Function: UC1 6

Use Case ID	UC006
Use Case name	Comment provider
Description	I want to analyze the summary.

Actors	System
Preconditions	Preprocessed data is available for analysis.
Postconditions	Analysis results are generated.
Flow of Events	1. The system fetches the preprocessed data. 2. The system performs analysis of the comments.
Exceptions	- Analysis failure due to technical issues.

Customization Function Use Case

"Customization" use case (UC007) detailed in Table 5.7, further enhances the code and optimizes it.

Table 5.7: Customization Function: UC1 7

Use case ID	UC007
Use Case Name	Customization
Description	Enhances the code, can also solve errors.
Actors	System
Preconditions	Model is integrated.
Postconditions	Analysis results are generated.
Flow of Events	1. The user checks the customization tab. 2. The user enters code snippet. 3. The system analyzes the code snippet and enhances it.
Exceptions	- Failure due to technical issues.

Llama 3 Model Use Case

"Llama 3 Model" use case (UC008) detailed in Table 5.8, further uses the model to perform the designated task.

Table 5.8: Llama 3 Model Function: UC1 8

Use case ID	UC008
Use Case Name	Llama 3 Model
Description	Performs the designated task.
Actors	System
Preconditions	Model is integrated.
Postconditions	Analysis results are generated.
Flow of Events	1. The user checks the tab (for functionality) and gives input. 2. The system prompts the Llama 3 model to process the input.

Exceptions	- Failure due to technical issues.
------------	------------------------------------

Generate Results Use Case

In "Results" use case (UC009) detailed in Table 5.9. outlines the generation of results on the interface. Users log in, adds code snippets and the model generates the results.

Table 5.9: Generate Results: UC1 9

Use Case ID	UC009
Use Case Name	Generate Results
Description	Generate the results of the functionality based on the code snippets.
Actors	System
Preconditions	Code is input and preprocessed.
Postconditions	Model generates the results
User views the analysis results.	1. User logs in to the website. 2. User prompts to perform the functionalities. 3. The model generates the results and passes it to the system.
Exceptions	- No analysis results generated.

Display Results Use Case

In "Results" use case (UC0010) detailed in Table 5.10. outlines the display of results on the interface. Users log in, and the system retrieves and displays the results.

Table 5.10: Display Results: UC1 10

Use Case ID	UC010
Use Case Name	Display Results
Description	view the results on the application interface.
Actors	User
Preconditions	Analysis results are available.
Postconditions	User views the analysis results.
User views the analysis results.	4. User logs in to the website. 5. User prompts to perform the functionalities. 6. The system displays the results on the user interface.
Exceptions	- No analysis results available for the user.

Log Out Use Case

In "Results" use case (UC0011) detailed in Table 5.11. outlines the display of results on the interface. Users log in, and the system retrieves and displays the results.

Table 5.11: Display Results: UC1 11

Use Case ID	UC011
Use Case Name	Log Out
Description	Log Out from the website.
Actors	User

Preconditions	User is logged in.
Postconditions	Log Out is successful
User views the analysis results.	1. The user clicks on the profile. A dropdown appears. 2. User clicks the log out button and is successfully logged out.
Exceptions	- No analysis results available for the user.

5.5 Sequence Diagram

The Java and Python dataset is preprocessed and used to train the models. The model which is more accurate (using evaluation metrics) is integrated into the interface. The summary is generated. The results, are shown to the user through the user-friendly website interface. System sequence is being depicted in Figure 5.5.

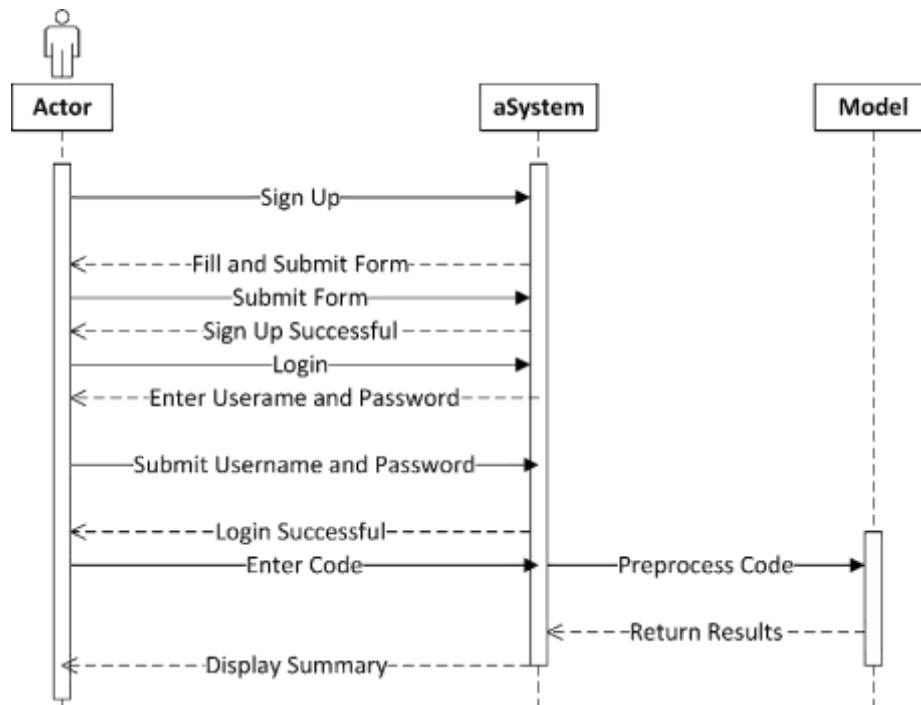


Figure 5.5: System Sequence Diagram

The actions occurring here are as follows:

- **User Setup and Login Process**

Figure 5.5 begins with the **User** signing up and setting up an account on the system. After signing up, the user proceeds to **log in** by entering their credentials (username and password) into the login form and submitting it to the system. Once the credentials are submitted, the system attempts to validate the information provided.

- **Successful Login**

If the credentials are correct, the system sends a response to the user indicating that the **login was successful**. If the login attempt fails, the system informs the user of the **login failure** with an error message.

- **Interaction with the Model**

After the user successfully logs in, the system interacts with the **Model**. The system sends a request to the model to summarize code based on the user's input or requirements. The model processes the request and generates the results.

- **Display of Results to the User**

The results are then sent back through the system and displayed to the user, completing the sequence.

In summary, Figure 5.5 demonstrates the flow from the user's sign-up and login attempts through to the successful interaction with the system and model, which results in the summarization and display of the code.

5.6 Class Diagram

The class diagram below includes all the classes required for our project. This diagram in Figure 5.7. provides a visual blueprint for our system's structure and functionality.

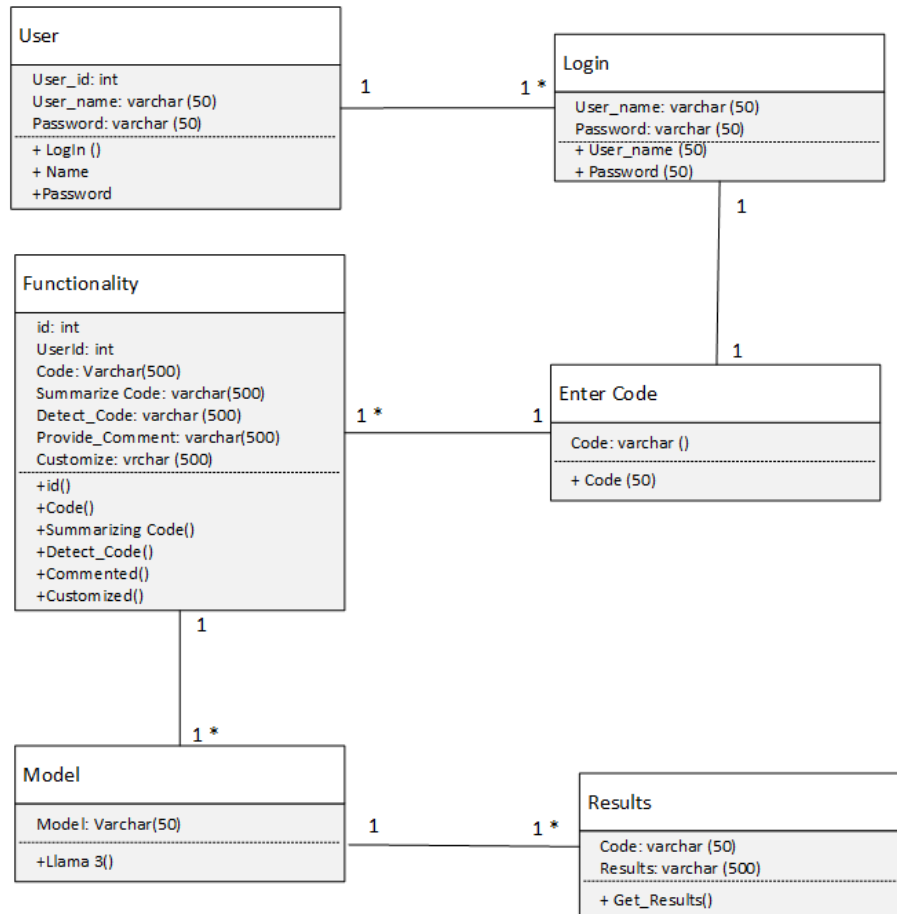


Figure 5.6: Class Diagram

CHAPTER 6 IMPLEMENTATION

6.1 Introduction

The implementation of our project represents a groundbreaking endeavor in the realm of code summarization, as it combines cutting-edge deep learning models, the Llama 3 and the CodeBERT model to generate code summaries. This introduction provides an overview of the motivation, significance, and objectives of the implementation, setting the stage for the subsequent sections.

Our implementation aims to address the pressing need for improved code summaries for enhanced code comprehension and understanding. By leveraging advanced technologies, we strive to make a meaningful impact in the software development industry. Our implementation has the potential to revolutionize this domain by utilizing deep learning models (Llama 3 and CodeBERT) to extract the syntactic and semantic information of the code to be summarized, enabling more accurate results. Our primary objectives are to leverage advanced deep learning models for accurate code summarization and to provide a code summarization tool for software developers. By accomplishing these objectives, we aim to revolutionize the software development industry.

6.2 Architecture Diagram of the Proposed System

Our tool aims to harness the power of Llama 3, a large language, deep learning model to summarize the code, detect errors, provide comments and customize it as well. The proposed architecture is broken down into front-end and backend. Figure 6.1 illustrates the architecture of our proposed system. Every module is present in the architecture for a seamless user experience.

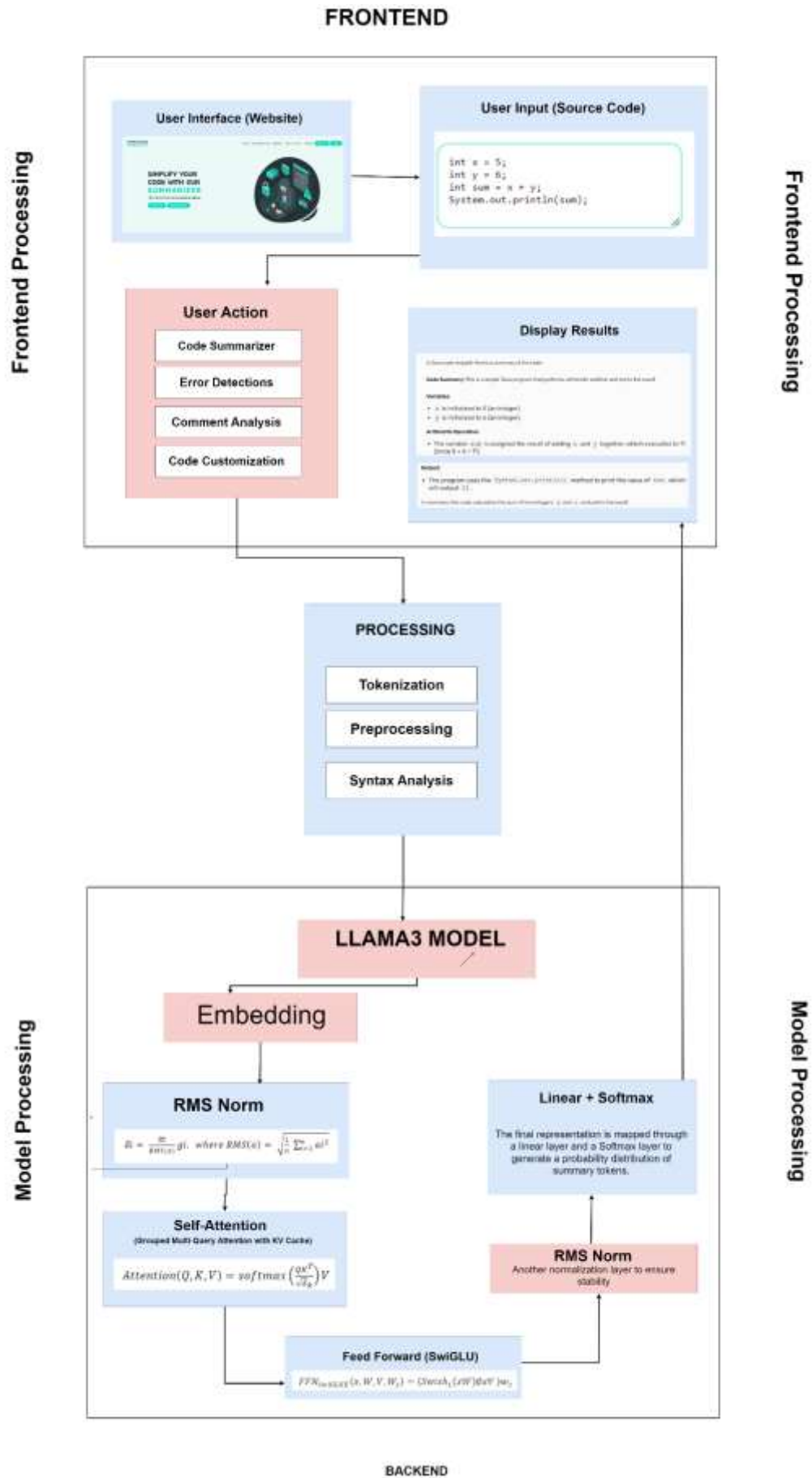


Figure 6.1. Architecture Diagram of the Proposed System

The user interface (website) provides an interface for users to interact with the system. Users can input their source code in languages like Java or Python into a text area. Various actions can be performed through buttons or options, such as generating a summary of the input code (Code Summarizer), identifying and highlighting errors (Error Detections), analyzing and summarizing comments (Comment Analysis), and customizing the summary or other outputs (Code Customization). The results of these actions, such as the generated summary, detected errors, or analyzed comments, are displayed to the user.

Processing acts as a bridge between the frontend and backend, handling user inputs and preparing them for processing by the backend. The input code undergoes tokenization, converting it into smaller units (tokens). Preprocessing cleans and normalizes the tokens by removing comments and handling whitespace. Syntax analysis then analyzes the structure of the code to create an abstract syntax tree (AST).

The LLAMA model serves as the core deep learning model used for code summarization and other analyses. It converts tokens or AST nodes into high-dimensional vectors (embedding) that the model can process. Root Mean Square Normalization (RMS Norm) is applied to stabilize training and improve performance. The model uses self-attention (Grouped Multi-Query Attention with KV Cache) to focus on different parts of the code, understanding context and dependencies efficiently. Rotary positional encodings add positional information to embedding to capture the sequential nature of the code. The feed-forward network (SwiGLU) processes the attention outputs and introduces non-linearity to capture complex patterns. Another RMS normalization layer ensures stability. The final representation is mapped through a linear layer and a Softmax layer to generate a probability distribution over possible summary tokens.

In short, the overall workflow begins with the user interface, where the user inputs their source code and selects an action, such as summarization. The input code undergoes tokenization, preprocessing, and syntax analysis in the processing stage to prepare it for the backend model. The preprocessed code is converted into embedding and passed through several layers of the LLaMA model, including self-attention, positional encodings, and feed-forward networks. Normalization layers ensure stable processing. The final processed representation is mapped through a linear layer and a Softmax layer to generate a summary or other outputs. The results, such as the code summary, are then displayed to the user on the frontend. This architecture allows for efficient and accurate

summarization and analysis of source code, leveraging advanced deep learning techniques to understand and process the input code.

6.3 Algorithms

The algorithm employed in our implementation involves the use of advanced deep learning model, namely Llama 3 and CodeBERT. These models have been selected due to their proven effectiveness in extracting meaningful syntactic and semantic information from the code. The utilization of these deep learning model enables our algorithm to leverage their unique strengths and enhance the accuracy and reliability of the summary being generated. Through an iterative process of training, validation, and fine-tuning, the algorithm optimizes its ability to generate accurate code summaries. By incorporating these advanced deep learning models into our algorithm, we aim to improve the accuracy and usefulness of the summary being generated, contributing to effective code comprehension and understanding.

6.3.1 Deep Learning Model

In this study, we applied the Llama 3 and CodeBERT as the deep learning model for code summary generation. Llama 3 achieved the Faithfulness score of 87%, contextual relevancy of 77%, contextual recall of 81%, and contextual precision of 72%.

CodeBERT achieved the Faithfulness score of 86%, contextual relevancy of 76%, contextual recall of 79%, and contextual precision of 69%.

6.3.1.1 Llama 3 model

The LLAMA3 model is built on the Transformer architecture, featuring multiple layers of self-attention mechanisms and feedforward neural networks. It consists of 8 billion and 70 billion parameters in different configurations, allowing for powerful language modeling capabilities. The model leverages extensive pre-training on over 15 trillion data tokens, enabling it to capture complex patterns and dependencies in text sequences effectively. With its large parameter size and sophisticated architecture, LLAMA3 has demonstrated state-of-the-art performance across various natural language processing tasks, making it a leading choice for researchers and practitioners in the field. The following steps are to be performed:

- i. **Import Required Libraries:** Import the required libraries for data preparation,

training and evaluation like SFTTrainer etc.

- ii. **Load the Model:** Load a pre-trained Llama 3 model using unsloth.
- iii. **Preprocess Dataset:** Load the dataset from the drive, perform some necessary filtering and use the tokenizer to convert the examples into tokenized input suitable for the model.
- iv. **Splitting Dataset:** The dataset is split into training, testing and validation dataset after preprocessing.
- v. **Finetuning Model:** The model is loaded and the tokenized training dataset is given to the model to be trained as shown in Figure 6.2.

```
from trl import SFTTrainer
from transformers import TrainingArguments
trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    dataset_num_proc = 2,
    packing = False, # Can make training 5x faster for short sequences.
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 5,
        max_steps = 60,
        num_train_epochs=30, #optional
        learning_rate = 2e-4,
        fp16 = not torch.cuda.is_bf16_supported(),
        bf16 = torch.cuda.is_bf16_supported(),
        logging_steps = 1,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
    )
)
```

Figure 6.2: Model Training Code

- vi. **Make Predictions:** The model is then tested on the testing data to find out if it is effective or not.
- vii. **Evaluate Model:** The model's results are evaluated using appropriate evaluation metrics like Contextual relevancy, Contextual Recall, Contextual Precision and Faithfulness metric.

The Llama 3 train- test results in Figure 6.3 is demonstrated:



Figure 6.3: Llama 3 Train Loss Graph

The Llama 3 train loss graph shows the model's loss decreasing over 60 global steps. Initially, the loss starts around 2.5, indicating significant errors, but it quickly drops as the model learns. After about 10 steps, the loss stabilizes around 1 with some fluctuations, reflecting the model's ongoing adjustments. Overall, the loss reduction signifies improvement in the model's performance, but the consistent fluctuations suggest there is room for further optimization. After using appropriate evaluation metrics like Faithfulness metrics, Contextual Precision, Contextual Relevancy and Contextual Recall, an evaluation bar plot was generated to better grasp the accuracy of the trained model. The evaluation graph below in Figure 6.4 summarizes our findings.

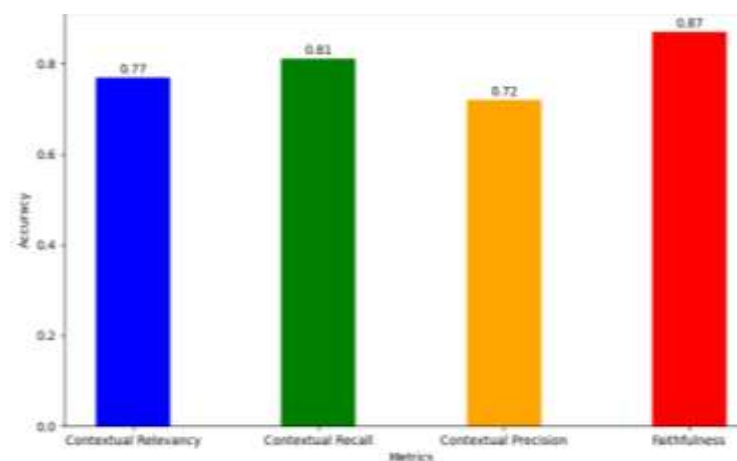


Figure 6.4: Evaluation Graph for Llama 3

Llama3 has a Contextual Relevancy of 0.77 or 77%. It means that the results generated by our model captures the main ideas of the original code by 77%. It has a Contextual Recall of 81%. It means that 81% proportion of the relevant information from the original code is included in the results generated by our fine-tuned model. It has a

Contextual Precision of 0.82 or 82%. It means that the results are relevant and accurately reflects the original code by 82%. It has a Faithfulness score of 0.87 or 87%. It means that the information in the result is accurate and true to the original code by 87%. The tabular representation of our findings are in Table 6.1.

Table 6.1: Evaluation Table for Llama 3

Mode l	Contextua l Relevancy	Contextua l Recall	Contextua l Precision	Faithfulnes s Score
Llama 3	0.77	0.81	0.82	0.87

6.3.1.2 CodeBERT model

The CodeBERT model utilizes multi-layer bidirectional Transformer architecture. This architecture enables the model to capture complex relationships and dependencies in both programming and natural languages. With 125 million parameters, CodeBERT is well-equipped to handle a wide range of NL-PL tasks effectively. By leveraging this advanced neural architecture, CodeBERT achieves impressive results in fine-tuning for specific downstream applications. While it demonstrates significant advancements, it also has some limitations to consider. It may not perform optimally in domains or programming languages that significantly differ from the ones it was trained on. Biases in the training dataset can impact model's generalization capabilities and may lead to biased outputs. Transferring knowledge from one domain to another may not always be seamless. Adapting the model to a new domain or language may require additional training and data (Feng et al., 2020). The following steps are to be performed:

- i. **Import Required Libraries:** Import the required libraries for data preparation, training and evaluation.
- ii. **Load the Model:** Load a pre-trained codeBERT model like "Microsoft/codebert-base".
- iii. **Preprocess Dataset:** Load the dataset from the drive, perform some necessary filtering and use the tokenizer to convert the examples into tokenized input suitable for the model.
- iv. **Finetuning Model:** The model is loaded and the tokenized training dataset is given to the model to be trained.

- v. **Make Predictions:** The model is then tested on the testing data to find out if it is effective or not.
- vi. **Evaluate Model:** The model's results are evaluated using appropriate evaluation metrics like Contextual relevancy, Contextual Recall, Contextual Precision and Faithfulness metric.

The train loss graph for CodeBERT is shown in Figure 6.5.

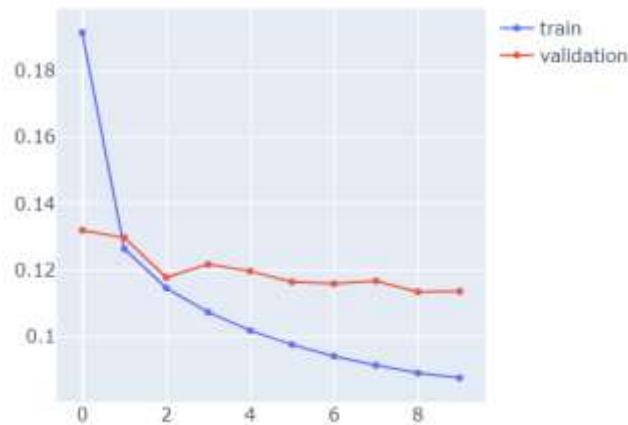


Figure 6.5: CodeBERT train Loss Graph

The CodeBERT train loss graph shows both the training and validation loss over 10 epochs. Initially, the training loss starts high at around 0.18 and rapidly decreases, indicating the model is learning effectively. The validation loss also starts high but remains relatively stable after a slight decline, indicating that the model may be overfitting to the training data. The gap between the training and validation losses widens after the second epoch, suggesting the model performs better on the training set than on unseen data. The evaluation report below in Figure 6.6 summarizes our findings.

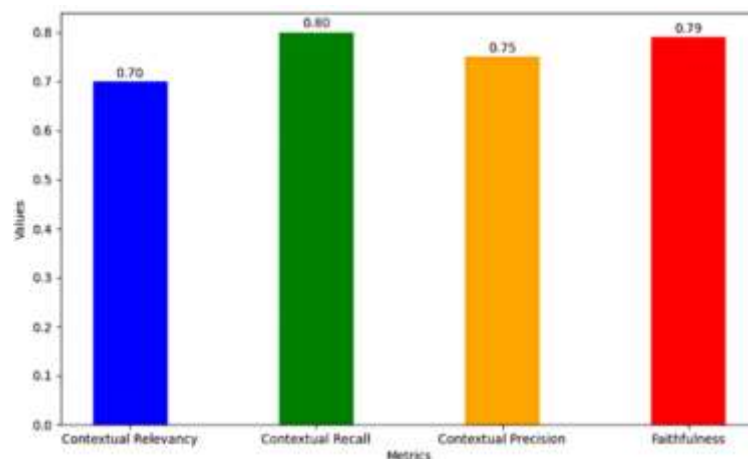


Figure 6.6: Evaluation for CodeBERT

CodeBERT has a Contextual Relevancy of 0.70 or 70%. It means that the results generated by our model captures the main ideas of the original code by 70%. It has a Contextual Recall of 80%. It means that 80% proportion of the relevant information from the original code is included in the results generated by our fine-tuned model. It has a Contextual Precision of 0.75 or 75%. It means that the results are relevant and accurately reflects the original code by 75%. It has a Faithfulness score of 0.79 or 79%. It means that the information in the result is accurate and true to the original code by 79%. The tabular representation of our findings are in Table 6.2.

Table 6.2: Evaluation Table for CodeBERT

Model	Contextual Relevancy	Contextual Recall	Contextual Precision	Faithfulness Score
Code BERT	0.70	0.80	0.75	0.79

6.4 Comparison

After fine-tuning both the Llama3 and the CodeBERT models, we will compare and contrast them to find out which of the two models would be integrated into the code summarization tool. For simplicity, we have concatenated the evaluation graphs in order to find out the comparatively better model shown in Figure 6.7.

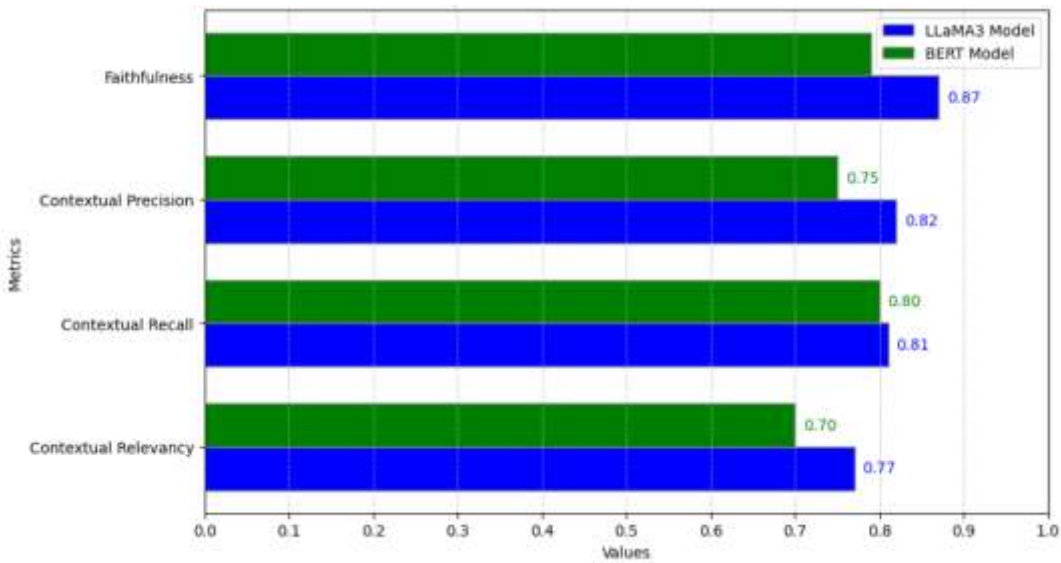


Figure 6.7: Comparative Analysis of Llama 3 and CodeBERT

Llama3 has a Contextual Relevancy of 0.77 or 77% whereas CodeBERT has a

Contextual Relevancy of 0.70 or 70%. It means that the results generated by the Llama model captures the main ideas of the original code by 77% and CodeBERT by 70% respectively. So, there is a 7% difference in their contextual relevancy that makes Llama3 outshine the working of CodeBERT. Llama3 has a Contextual Recall of 81% whereas CodeBERT has a Contextual Recall of 80%. It means that 81% proportion of the relevant information from the original code is included in the results generated by the Llama3 model and 80% by CodeBERT. The difference between them is negligible using this evaluation metrics. So, here, both of the models are equally useful. Llama3 has a Contextual Precision of 0.82 or 82% whereas CodeBERT has a Contextual Precision of 0.75 or 75%. It means that the results by Llama3 model are relevant and accurately reflects the original code by 82%. The results by CodeBERT model are relevant and accurately reflects the original code by 75%. There is a big difference in this evaluation metric and Llama 3 outperforms CodeBERT here. Llama3 has a Faithfulness score of 0.87 or 87% whereas CodeBERT has a Faithfulness score of 0.79 or 79%. It means that the information in the result generated by Llama 3 is accurate and true to the original code by 87%. The information in the result generated by CodeBERT is accurate and true to the original code by 79%. Here, Llama 3 outperforms CodeBERT with a massive difference. To create an effective code summarization tool, we need to select the model which outperforms the other. Llama 3 outperforms CodeBERT with respect to Faithfulness, Contextual Recall and Contextual Relevancy. So, it would be integrated into the code summarization tool for a seamless code summarization experience. Table 6.3 is the tabular representation of our findings highlighting the comparison of our two models.

Table 6.3: Evaluation Comparison of Llama 3 and CodeBERT

Model	Contextual Relevancy	Contextual Recall	Contextual Precision	Faithfulness Score
Llama 3	0.77	0.81	0.82	0.87
Code BERT	0.70	0.80	0.75	0.79

6.5 Libraries and Packages

Here are some of the key libraries and packages that were used:

Tensor Flow: We used Tensor Flow, an open-source deep learning framework, to implement our models. It facilitates operations like data manipulation, model definition, and loss calculation during the training process.

Transformers: The Transformers library gives a thorough set-up of pre-trained models for different Natural Language Processing (NLP) applications. These models are promptly accessible for finetuning, accelerating development compared to training from scratch.

SFTTrainer: SFTTrainer provides functionalities like data preparation, model optimization, and evaluation metrics, all tailored towards supervised fine-tuning on data.

6.6 User Interface (UI)

Home page is shown in Figure 6.8, which introduces the code summarizer and the necessary information a beginner needs to know to use it.

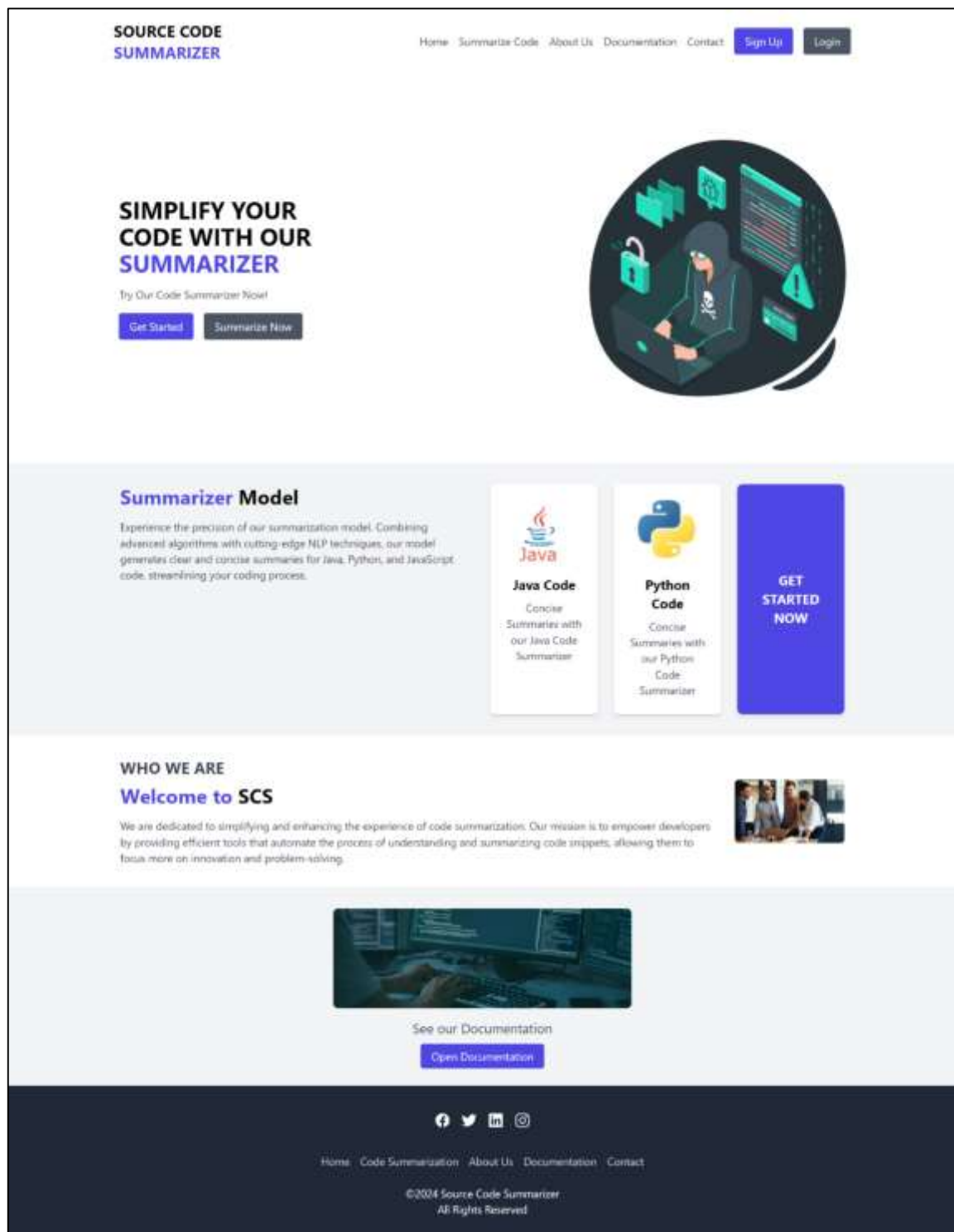
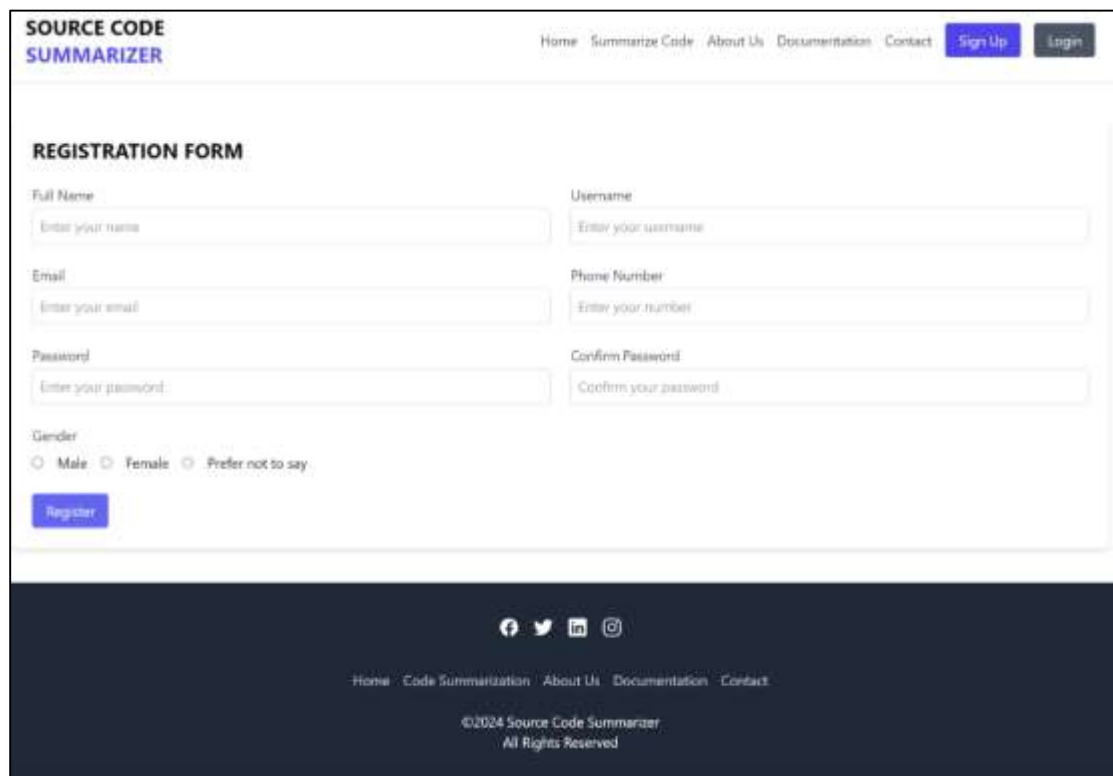


Figure 6.8: Home Page

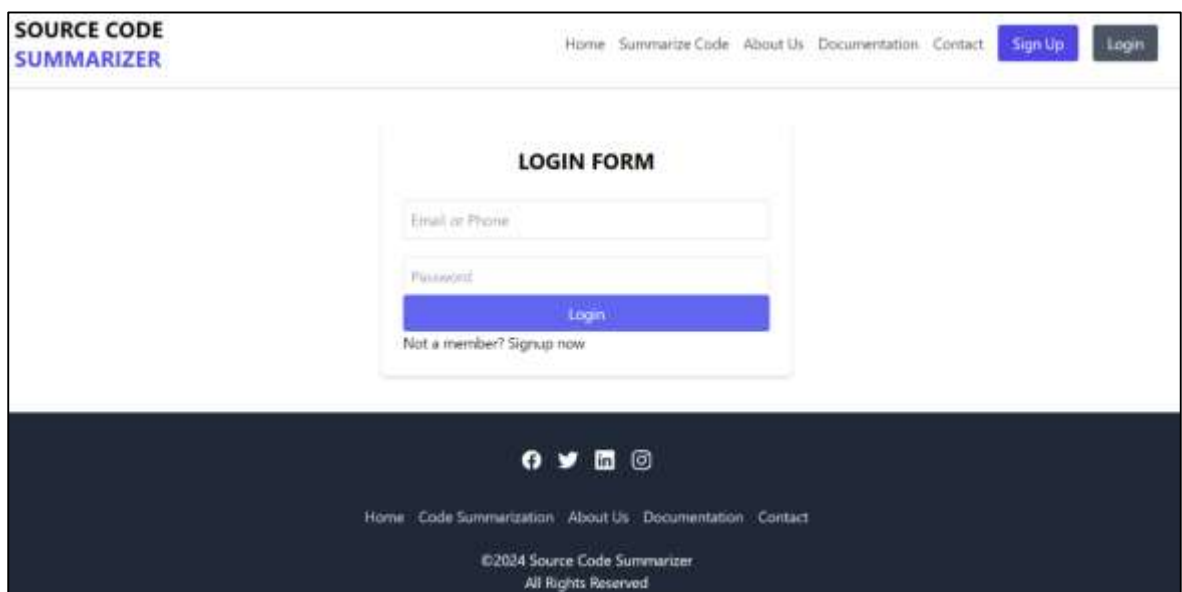
Signup page with all the validation and input fields is shown in Figure 6.9.



The screenshot displays the 'SOURCE CODE SUMMARIZER' website's registration page. The header includes the site logo and navigation links: Home, Summarize Code, About Us, Documentation, Contact, Sign Up, and Login. The main content area is titled 'REGISTRATION FORM' and contains several input fields: Full Name, Username, Email, Phone Number, Password, and Confirm Password. Below these fields is a 'Gender' section with radio buttons for Male, Female, and Prefer not to say. A blue 'Register' button is positioned at the bottom of the form. The footer features social media icons, a secondary navigation menu, and copyright information for 2024 Source Code Summarizer.

Figure 6.9: Sign Up or Registration Page

Login page is shown in Figure 6.10.



The screenshot shows the 'SOURCE CODE SUMMARIZER' website's login page. The header is identical to the registration page, with the site logo and navigation links. The main content area is titled 'LOGIN FORM' and contains two input fields: Email or Phone and Password. A blue 'Login' button is located below the password field. A link for 'Not a member? Signup now' is provided at the bottom of the login form. The footer includes social media icons, a secondary navigation menu, and copyright information for 2024 Source Code Summarizer.

Figure 6.110 Login Page

The summarizer page in which the summarization, error detection, comment provider and customization functionality is to be performed is shown in Figure 6.11.

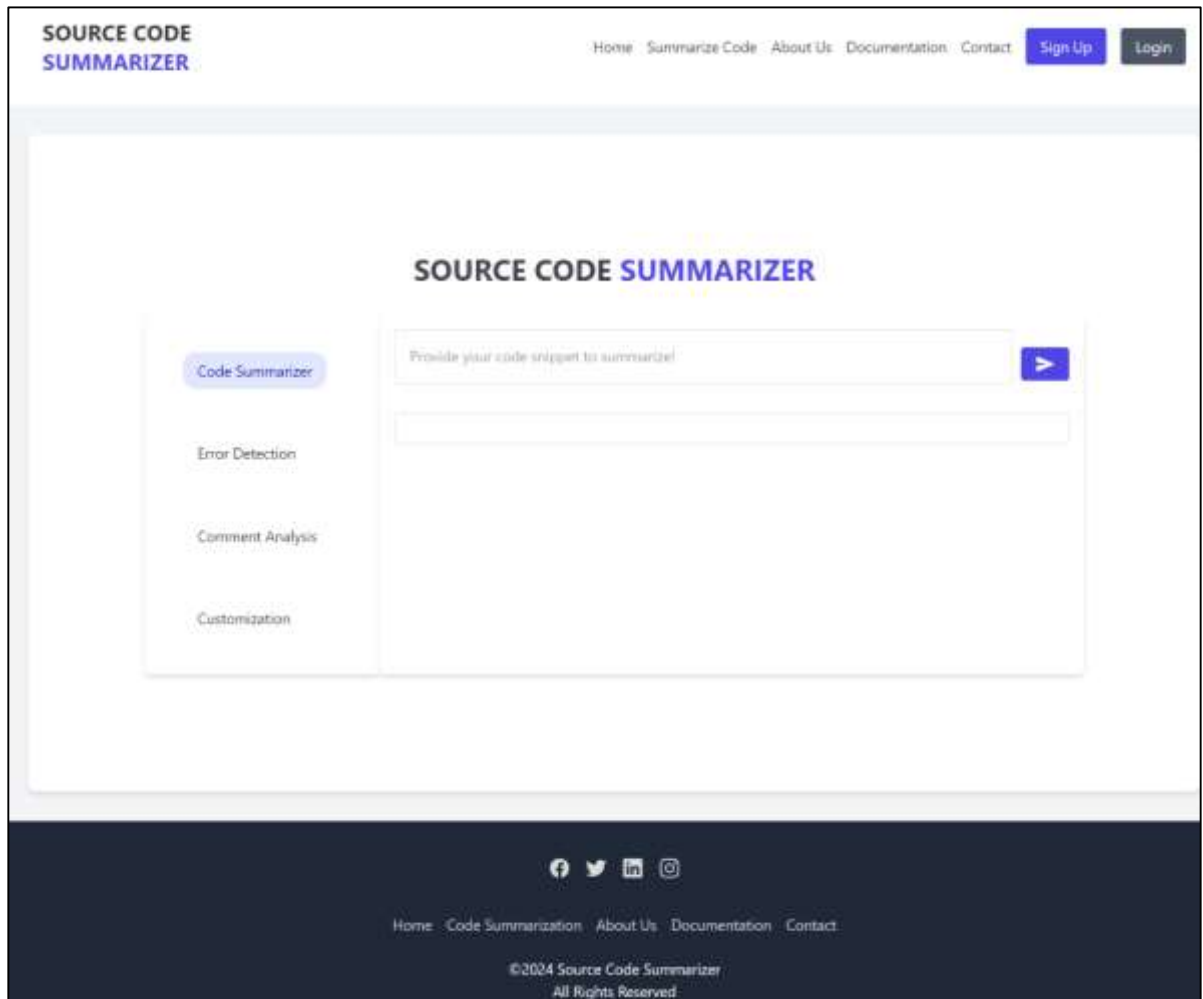


Figure 6.11: Code summarizer Page

The Contact Us Page is shown in Figure 6.12.

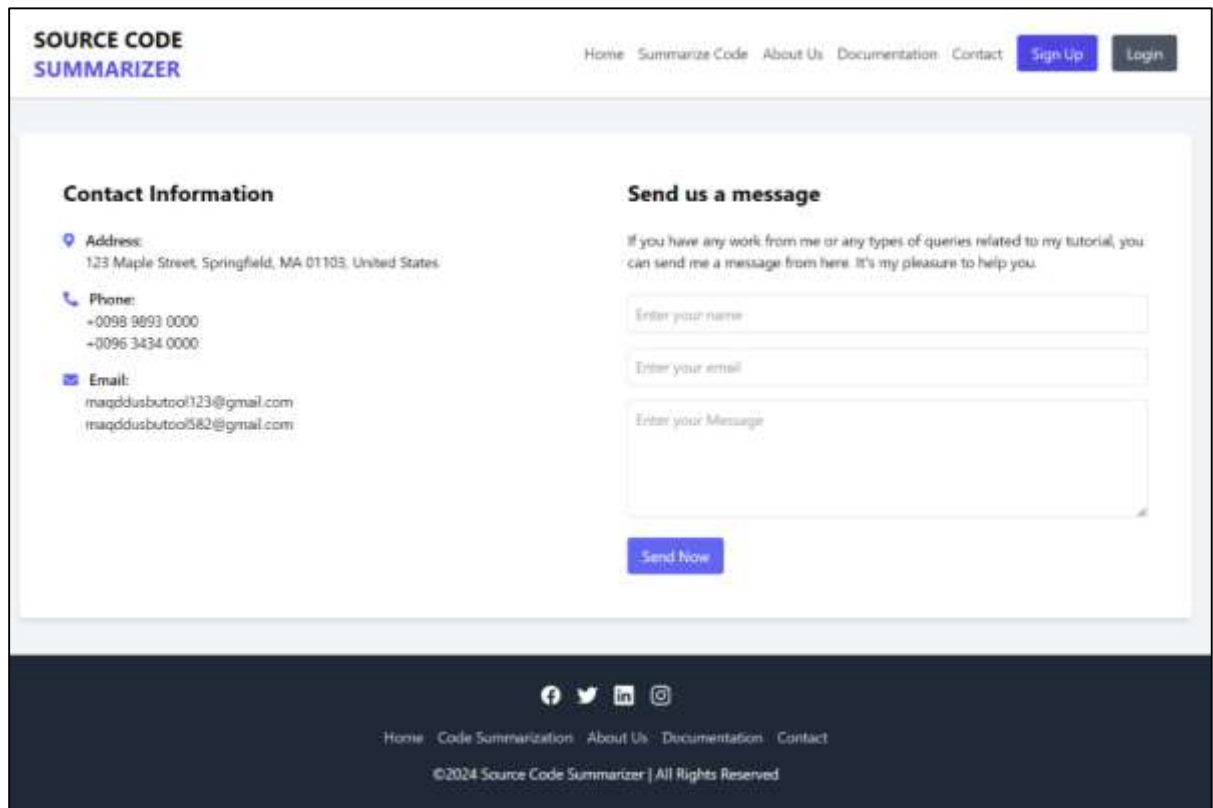


Figure 6.12: Contact Us Page

6.7 Case Study

In this project, we have fine-tuned two deep learning models and have chosen the best model to integrate into our website. Now, after the code summarizer has been developed, we will compare its results to other chatbots like ChatGPT, Gemini, and CodeGPT.

Firstly, we would use our code summarizer to generate summaries. The code to be entered is a simple Java code to add two numbers. Figure 6.13 shows the output of our code summarizer.

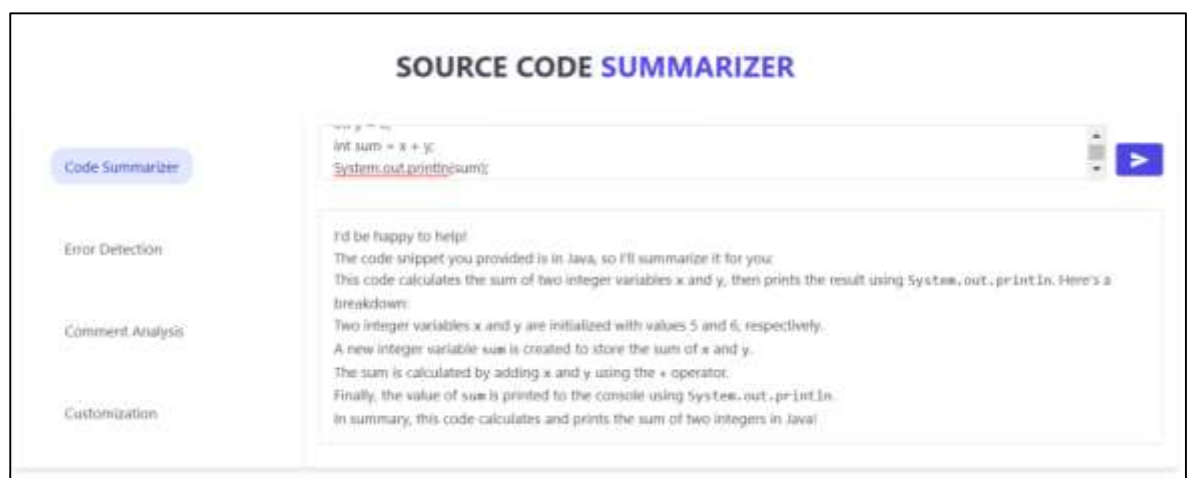
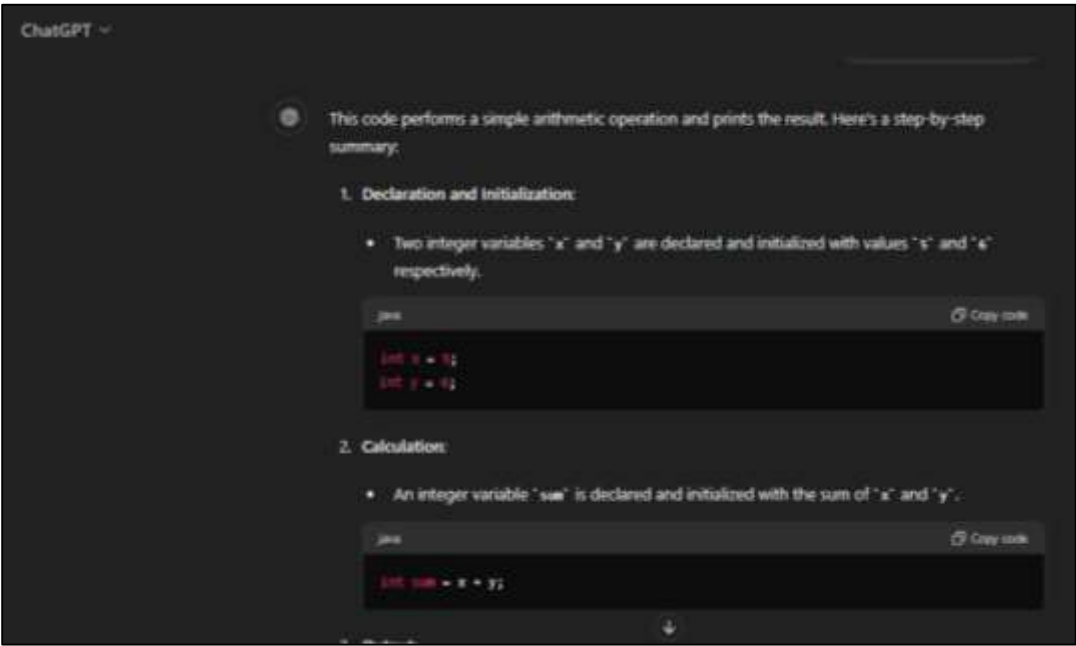


Figure 6.13: Code Summarizer Output

Next, we would give the same code to ChatGPT for summarization as shown in Figure 6.14.

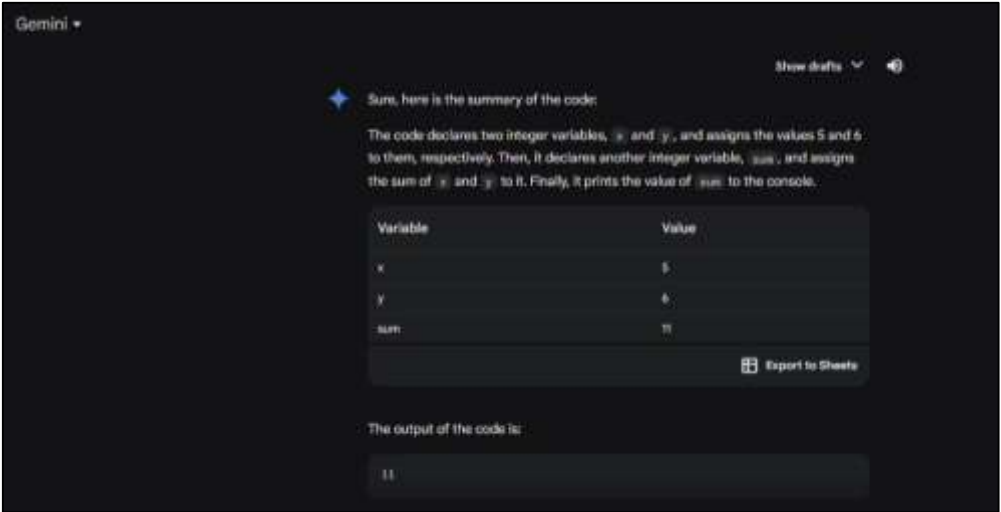


The screenshot shows the ChatGPT interface with a dark theme. The user's input is not visible, but the model's response is. It starts with a summary statement: "This code performs a simple arithmetic operation and prints the result. Here's a step-by-step summary:". This is followed by a numbered list item "1. Declaration and Initialization:" which contains a bullet point stating that two integer variables 'x' and 'y' are declared and initialized with values 5 and 6 respectively. Below this, a code block is shown with the lines `int x = 5;` and `int y = 6;`. The next numbered list item is "2. Calculation:", which contains a bullet point stating that an integer variable 'sum' is declared and initialized with the sum of 'x' and 'y'. Below this, another code block shows the line `int sum = x + y;`. The interface includes a "Copy code" button for each code block and a "Show drafts" button at the top right.

Figure 6.14: ChatGPT Output

ChatGPT provides extensive explanation even if it is asked to provide a summary. It can be challenging for complex and codes spanning hundreds or thousands of lines.

Next, we will use Gemini to generate summaries as shown in Figure 6.15.



The screenshot shows the Gemini interface with a dark theme. The model's response begins with "Sure, here is the summary of the code:" followed by a paragraph explaining the code's logic: declaring variables x and y with values 5 and 6, then declaring a variable sum and assigning it the sum of x and y, and finally printing the value of sum. Below the text is a table with two columns, "Variable" and "Value". The table contains three rows: x with value 5, y with value 6, and sum with value 11. An "Export to Sheets" button is located at the bottom right of the table. Below the table, the text "The output of the code is:" is followed by a code block containing the number 11. A "Show drafts" button is visible at the top right.

Variable	Value
x	5
y	6
sum	11

Figure 6.15: Gemini Output

The summary provided by Gemini is effective but it tends to explain more than just providing a summary. It also creates a table of all the variables and their corresponding values in the code. This visualization is not effective for large codebases where there are hundreds of variables declared and their values changing after every time interval.

Next, CodeGPT was also asked to summarize the same code snippet as shown in Figure

6.16.

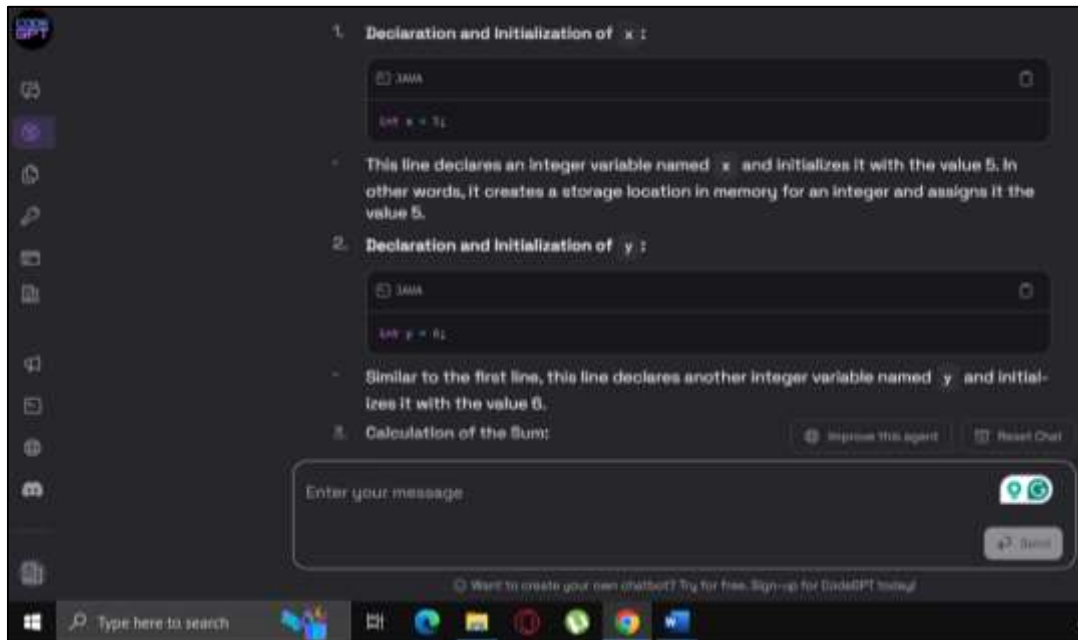


Figure 6.16: CodeGPT Output

Even if CodeGPT is prompted to summarize the code, it provides extensive summary of every instruction. It can be convenient to understand short codes, but it is not effective to understand large codebases.

In short, code summarizer provides concise and to the point summaries and never deviate from its purpose. It provides summaries that are enough to make the users understand the code regardless of the length of the code.

6.7.1 Architecture Comparison

A comparative analysis of the architectures of Llama 3, CodeBERT, GPT-4, Gemini, and CodeGPT was done. Each model was rigorously evaluated using a comprehensive set of metrics. These metrics were chosen to provide a holistic assessment of each model's ability to generate summaries that are not only syntactically accurate but also contextually appropriate and faithful to the original code. Through this detailed evaluation, we aim to identify the strengths and weaknesses of each model and highlight the model that stands out in terms of overall performance and reliability. The metrics used for comparison are as follows:

- The **BLEU (Bilingual Evaluation Understudy)** metric is a widely used method for evaluating the quality of text generated by a machine, particularly in tasks such as machine translation and text summarization. It measures the correspondence

between the machine-generated text and one or more reference texts, focusing on the precision of n-grams (contiguous sequences of words) to assess how closely the generated text matches the reference texts.

- The **ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation)** metric is a tool for evaluating the quality of machine-generated text, particularly in summarization tasks. It emphasizes the longest common subsequence (LCS) between the generated text and reference text to measure their similarity. ROUGE-L captures both precision and recall, focusing on the sequence of words to assess how well the generated text aligns with the reference, thereby providing insights into the fluency and coherence of the summaries.
- The **METEOR (Metric for Evaluation of Translation with Explicit ORdering)** metric is a tool for assessing the quality of machine-generated text, especially in translation and summarization tasks. Unlike metrics that rely solely on exact word matches, METEOR incorporates synonymy, stemming, and word order to evaluate the generated text's alignment with reference texts. By considering these linguistic factors, METEOR provides a more nuanced and comprehensive evaluation, capturing the semantic similarity and overall fluency of the generated text.
- **Contextual relevancy** is a metric used to evaluate the quality of machine-generated text by assessing how well the generated content aligns with the context and meaning of the original input. It measures the extent to which the generated text maintains the same thematic elements, logical flow, and pertinent information as the source material. High contextual relevancy indicates that the summary or translation not only matches the words but also preserves the intent, nuances, and context of the original text, ensuring that the output is meaningful and accurate in its application.
- **Contextual recall** is a metric used to evaluate the completeness and coverage of machine-generated text in relation to the original input. It measures the proportion of relevant and contextually appropriate information from the source that is successfully captured and included in the generated text. High contextual recall indicates that the generated summary or translation retains a substantial amount of the key elements, themes, and details from the original content, ensuring that important aspects are not omitted and the output is comprehensive and faithful to the source material.

- **Contextual precision** is a metric used to evaluate the accuracy and relevance of the information included in machine-generated text in relation to the original input. It measures the proportion of the generated content that is contextually appropriate and aligns well with the source material. High contextual precision indicates that the summary or translation not only includes relevant information but also avoids adding extraneous or misleading details, ensuring that the output is both accurate and pertinent to the original context.
- **Faithfulness** is a metric used to assess the extent to which machine-generated text accurately represents the content and meaning of the original input. It evaluates whether the generated summary or translation stays true to the source material, preserving its factual accuracy, key details, and intended message without introducing errors, distortions, or unsupported information. High faithfulness ensures that the output is a reliable and trustworthy representation of the original text, maintaining its integrity and authenticity.

All the models to be compared were evaluated based on the above evaluation metrics. The results of Contextual Relevancy, Contextual recall, contextual precision, and faithfulness are summarized in Table 6.4.

Table 6.4: Comparison of Various Deep Learning Models

Model	Contextual Relevancy	Contextual Recall	Contextual Precision	Faithfulness Score
Llama 3	0.77	0.81	0.82	0.87
Code BERT	0.70	0.80	0.75	0.79
Chat GPT	0.73	0.79	0.78	0.77
Gemini	0.74	0.77	0.80	0.80
Code GPT	0.76	0.77	0.71	0.82

From Table 6.4, it appears that Llama 3 outperforms other models. To validate this hypothesis, we use more evaluation metrics (BLEU, ROGUE L, and METEOR) as

shown in Table 6.5.

Table 6.5: Comparison of Various Deep Learning Models using BLEU, ROGUE L, METEOR

Model	BLEU	ROGUE L	METEOR
Llama 3	33	45.2	30.5
Code BERT	31	40	26.6
Chat GPT	31.5	42.4	28
Gemini	30.7	42.1	28.3
Code GPT	28.4	40.5	27.1

From Table 6.4 and Table 6.5, it is concluded that Llama 3 outperforms various state-of-the-art deep learning models in case of code summarization. Hence, Llama 3 was chosen for creating the code summarizer.

Chapter 7 TESTING AND EVALUATION

7.1 Testing

Testing is a crucial procedure conducted to assess whether a system or its components meet the specified requirements. It involves executing the system and comparing its actual outputs with the expected outputs to identify any flaws, errors, or missing functionalities. The goal of testing is to uncover discrepancies between the actual and expected results, enabling the detection and resolution of issues. This process yields valuable insights into the system's performance and ensures that it aligns with the desired outcomes.

7.2 Unit Testing

The chosen testing approach for evaluating the application's strength and reliability is unit testing, a white-box testing method. Unit testing focuses on verifying the functionality and non- functionality of individual modules within the code. Its primary objective is to isolate and test each component of the program to ensure their proper functioning. This testing method is efficient in terms of time and aligns well with our development strategy. Below are the test cases and their corresponding results. This strategy is additionally time productive and is best suited to our improvement technique. Following are the test cases and their results.

7.3 Test Cases

7.3.1 Test Case 1 (Input Code Testing)

Testing the system with different types of code snippets in Table 7.1. of varying size and complexity. Verifying that the system can handle different types of code snippets. Testing the system's ability to handle code snippets with different levels of complexity.

Table 7.1: Test Case 1 (Input Code Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test whether Input code is working without any errors.	Test ID: 1
Version: 1	Test Type: Unit

Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Log-in to website, go the summarizer page and paste the code snippet. 3. Input Code Testing performed (whether the website accepts the code or not).
Expected Results: Website detects the uploaded code and generates summary successfully as depicted in Figure 7.1.
Actual Results: Passed



Figure 7.1: Input code test case

7.3.2 Test Case 2 (Sign Up Page Testing)

Sign-up page testing involves evaluating the functionality, usability, and security of the website sign-up process. Testers assess whether users can successfully create accounts, verify email addresses, set passwords, and navigate through any additional steps done in below Table 7.2.

Table 7.2: Test Case 2 (Sign Up Page Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test whether Input fields have validation checks and working without any errors.	Test ID: 2
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Sign-un to website 3. Insert the data into the fields. 4. Check whether the sign-up gives an error of validation when left the fields empty. 	

Expected Results: Signup form detects that input fields are empty and did not let signup action to happen as depicted in Figure 7.2.

Actual Results: Passed

The screenshot shows the 'SOURCE CODE SUMMARIZER' website with a navigation bar containing links for Home, Summarize Code, About Us, Documentation, Contact, Sign Up, and Login. The 'REGISTRATION FORM' is displayed with the following fields: Full Name (placeholder: Enter your full name), Username (placeholder: Enter your username), Email (placeholder: Enter your email), Password (placeholder: Enter your password), Confirm Password (placeholder: Confirm your password), and Gender (radio buttons for Male, Female, and Prefer not to say). A blue 'Register' button is at the bottom left. A yellow tooltip with an exclamation mark icon and the text 'Please fill out this field.' is positioned over the Email field.

This screenshot shows the registration form with the following data entered: Full Name: maqddus butool, Username: maqddus, Email: maqddusaskhab@gmail.com, Phone Number: 4567890678, Password: ***, and Confirm Password: ***. The Gender 'Female' option is selected. A yellow tooltip with an exclamation mark icon and the text 'Please include an '@' in the email address. 'maqddusaskhab@gmail.com' is missing an '@'.' is displayed over the Password field. The blue 'Register' button is at the bottom left.

This screenshot shows the registration form with the same data as the previous one. A dark grey error message overlay is present in the top right corner, displaying the text '127.0.0.1:5500 says Error: Username already registered' and a green 'OK' button. The registration form fields and the 'Register' button remain visible below the overlay.

Figure 7.2: Sign up page validation testing

7.3.3 Test Case 3 (Login Page Testing)

"Login" page testing involves assessing the functionality and security of a feature that allows users to access an application or platform using its credentials as shown in Table 7.3.

Table 7.3: Test Case 3 (Login Page Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test whether Login Page is working without any errors.	Test ID: 3
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none">1. Run the Website, using Visual Studio.2. Go to login page of website3. Check whether the Login gives an error of validation when left the fields empty.	
Expected Results: Login page opens successfully as depicted in Figure 7.3.	
Actual Results: Passed	

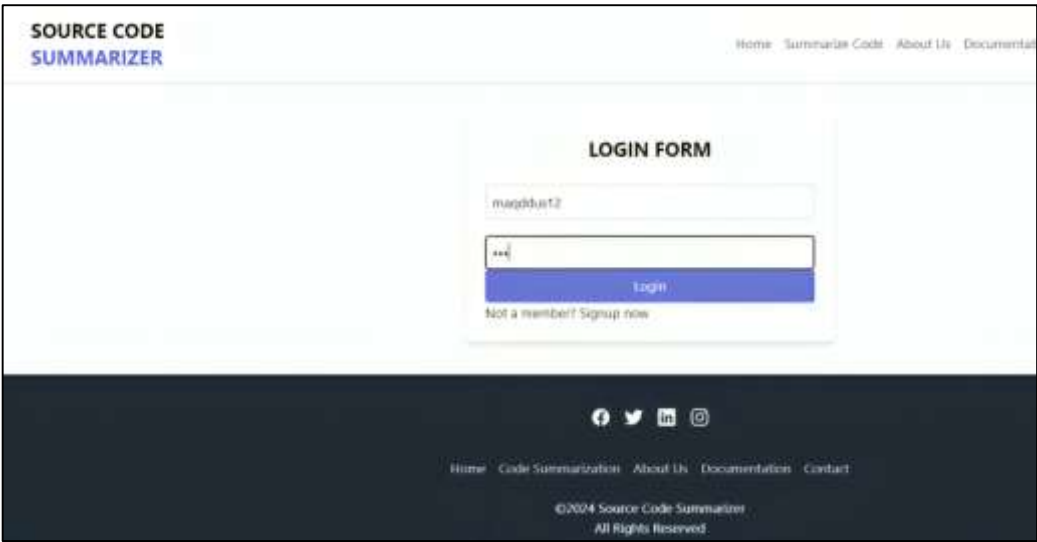


Figure 7.3: Login Page testing

7.3.4 Test Case 4 (Code Summarizer Testing)

Testing the "Code Summarizer" functionality to validate the generation of code summaries depicted below in Table 7.4.

Table 7.4: Test Case 4 (Code Summarizer Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test whether the code summarizing functionality is working or not	Test ID: 4
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the Website, using Visual Studio. 2. Go to login page of website. 3. Login with the public address and password. 4. Enter a code snippet and check whether the summary is generated or not. 	
Expected Results: Code is summarizer functionality is working without any errors successfully as depicted in Figure 7.4.	
Actual Results: Passed	

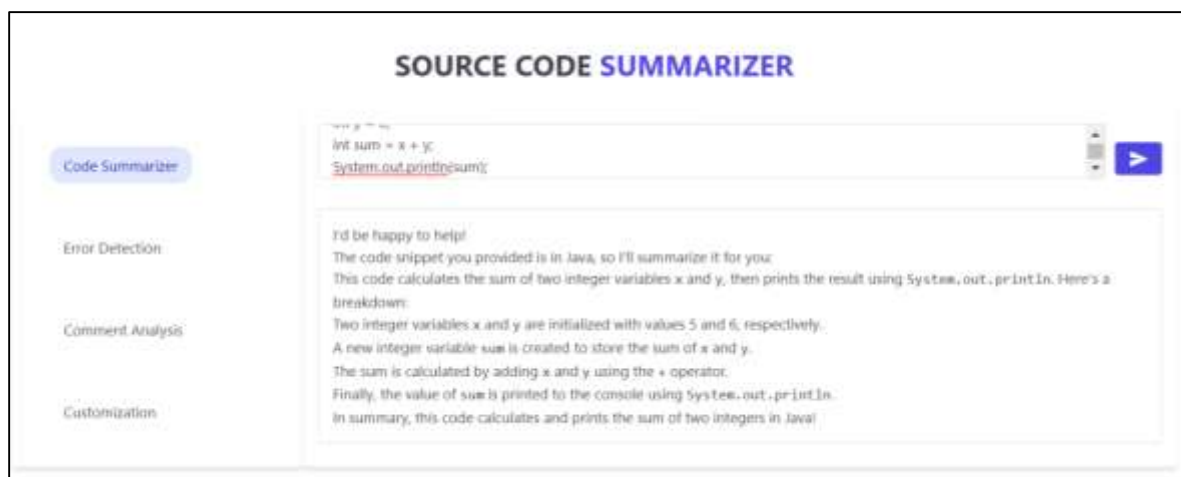


Figure 7.4: Code Summarizer Testing

7.3.5 Test Case 5 (Error Detection Testing)

Error detection is also a functionality added in the summarizer which is shown in below Table 7.5.

Table 7.5: Test Case 5 (Error Detection Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test whether the error detection functionality is working or not	Test ID: 5

Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Check whether the error detection functionality is being performed on the entered code snippet. 	
Expected Results: The error detection functionality is working successfully as depicted in Figure 7.5.	
Actual Results: Passed	



Figure 7.5: Error Detection Testing

7.3.6 Test Case 6 (Comment provider Testing)

Testing the comment provider functionality of the system shown below in Table 7.6.

Table 7.6: Test Case 6 (Comment provider Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test whether the comment provider functionality is being performed or not.	Test ID: 6
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Log-in to website, enter the code snippet, press the button to get results. 	
Expected Results: Comment provider is working successfully and accurately as depicted in Figure 7.6.	
Actual Results: Passed	



Figure 7.6: Comment provider testing

7.3.7 Test Case 7 (Performance and Scalability Testing)

Evaluating the system's performance below Table 7.7 by measuring the time taken for code preprocessing, feature extraction, and summarization. Measure the system's resource consumption, such as CPU and memory usage, during the prediction process.

Table 7. 7: Test Case 7 (Performance and Scalability Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
Objective: To test system's performance that it is working without any errors.	Test ID: 7
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Log-in to website, get results. 3. Evaluate the system's performance by measuring the time taken for code preprocessing and summarization. 	
Expected Results: Application is working successfully and accurately.	
Actual Results: Passed	

7.3.8 Test Case 8 (Usability and User Experience Testing)

Evaluate the system's user interface and assess its ease of use for end users in below Table 7.8.

Table 7.8: Test Case 8 (Usability and User Experience Testing)

System: Code Summarizer Website	Tested By: Mehvish Zahid
--	---------------------------------

Objective: To test system's user interface is working without any errors.	Test ID: 8
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Log-in to website, get results, Accessing all the pages, Signup, logout. 3. Evaluate the system's user interface performance and its ease of use for end users that it is ensuring smooth working of the website. 	
Expected Results: website is working successfully and accurately as depicted in Figure 7.7.	
Actual Results: Passed	



Figure 7.7: Usability and User Experience Testing

7.3.9 Test Case 9 (Contact Us Form Testing)

The Contact Us form testing is shown in below Table 7.9.

Table 7.9: Test Case 9 (Contact Us form Testing)

System: Code Summarizer Website	Tested By: Kainat Farooqi
Objective: To test where the Contact Us form is working or not.	Test ID: 9
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 1. Run the website, using Visual Studio. 2. Log-in to website, got to the Contact Us Form. 3. Fill the required fields in the form and check whether it is stored anywhere. 	

Expected Results: The data to be entered in Contact Us form is not stored anywhere. So, it is not working as depicted in Figure 7.8.

Actual Results: Failed

Figure 7.8: Contact Us Form Testing

7.3.10 Test Case 10 (Social Media Handles Testing)

The testing for social media handles presented in the footer of the website is shown in below Table 7.10.

Table 7.10: Test Case 10 (Social Media Handles Testing)

System: Code Summarizer Website	Tested By: Kainat Farooqi
Objective: To test whether the social media icons are working or not.	Test ID: 10
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> Run the website, using Visual Studio. Log-in to website, scroll down and got to the social media handle icons in the footer (Facebook, Instagram, Twitter). Click on each of the icons to test whether they take us to the social media accounts or platforms. 	
Expected Results: The icons do not take the website to any social media profile. So, it is not working as depicted in Figure 7.9.	
Actual Results: Failed	

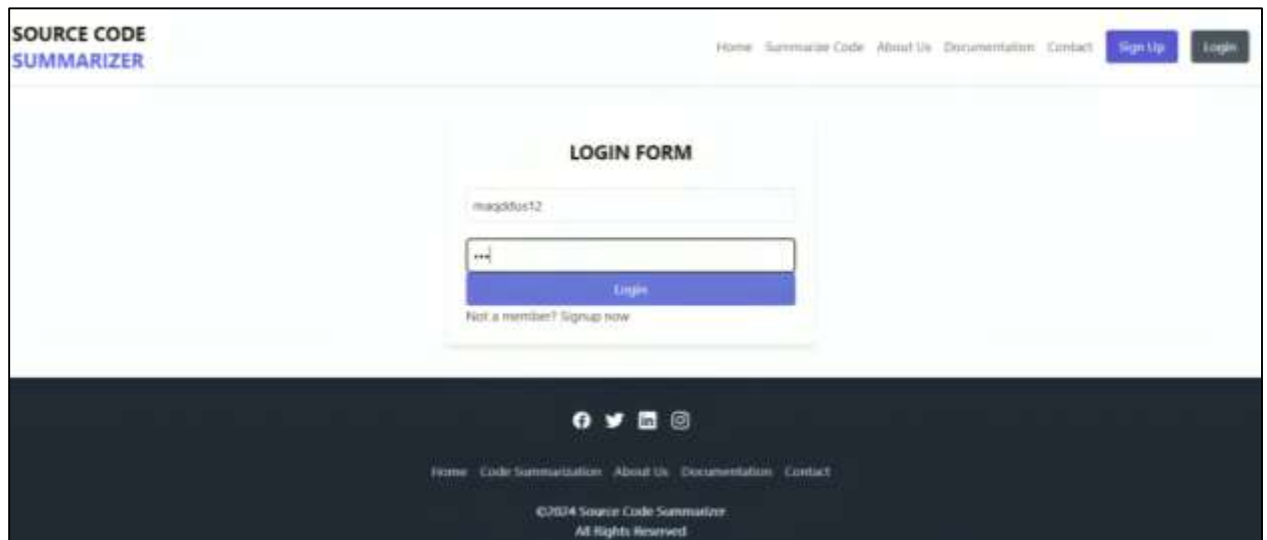


Figure 7.9: Social Media Handle Testing

7.3.11 Test Case 11 (Responsiveness Testing)

The testing for responsiveness website is shown in below Table 7.11.

Table 7.11: Test Case 11 (Responsiveness Testing)

System: Code Summarizer Website	Tested By: Kainat Farooqi
Objective: To test whether the website fit to the mobile and tablet screen or not.	Test ID: 11
Version: 1	Test Type: Unit
Steps to Perform: <ol style="list-style-type: none"> 7. Run the website, using Visual Studio Code. 8. Log-in to website. 9. Decrease the width and height of the browser to check if the website adapts to the decreased dimensions. 	
Expected Results: The website adapts to the decreasing dimensions. So, it is not responsive as depicted in Figure 7.10.	
Actual Results: Passed	

SOURCE CODE
SUMMARIZER

REGISTRATION FORM

Full Name

Enter your name

Username

Enter your username

Email

Enter your email

Phone Number

Enter your number

Password

Enter your password

Confirm Password

Confirm your password

Gender

☐ Male ☐ Female ☐ Prefer not to say

Register

[f](#) [t](#) [in](#) [ig](#)

[Home](#) [Code Summarization](#) [About Us](#) [Documentation](#) [Contact](#)

©2024 Source Code Summarizer
All Rights Reserved

Figure 7.10: Responsiveness Testing

Chapter 8 CONCLUSION AND FUTURE WORK

8.1 Conclusion

The successful demonstration of the code summarizer project, fine-tuning CodeBERT and Llama 3 model and employing the Llama 3 model, highlights its potential to significantly improve developer's code understanding capabilities by providing relevant and effective summaries. By leveraging the power of this model, the project has achieved improved accuracy and relevance in matching the code snippets with the corresponding summaries. The project's success in implementing this model demonstrates the possibilities of leveraging large language models to tackle the complexities of code summarization. By combining the strengths of Llama 3, the project has opened doors for further advancements in this domain. Moving forward, future work can focus on scaling the system and expanding language support. Additionally, incorporating feedback from developers and integrating the code summarizer with popular developer tools can enhance the user experience and increase adoption. Overall, the code summarizer project has made significant strides in improving the efficiency and accuracy of code summarization. With continued development and enhancement, it has the potential to greatly benefit developers by providing them with a powerful and intuitive tool for code summarization.

8.2 Future Work

Our project's success opens up exciting opportunities for future advancements in the software engineering domain, specifically, code summarization. In the future, we aim to elevate our system's abilities and acquire additional resources. Our goal is to broaden our research scope to include the various programming languages contained within the CodeSearchNet dataset. This expansion will allow us to explore and incorporate insights from a wider array of programming languages, enabling a more comprehensive understanding and enhancement of our systems.

REFERENCES

- Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Tom Zimmermann. 2022. Practitioners' Expectations on Automated Code Comment Generation. In ICSE '22: Proceedings of the 44th ACM/IEEE International Conference on Software Engineering.
- Liu, Y.; Sun, X.B.; Li, B. Research on Automatic Summarization for Java Packages. J. Front. Comput. Sci. Technol. 2017, 11, 46–54.
- Eddy, B.P.; Robinson, J.A.; Kraft, N.A.; Carver, J.C. Evaluating source code summarization techniques: Replication and expansion. In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013; pp. 13–22.
- Wang, J.; Xue, X.; Weng, W. Source code summarization technology based on syntactic analysis. J. Comput. Appl. 2015, 35, 1999.
- Ou, M.; Cui, P.; Pei, J.; Zhang, Z.; Zhu, W. Asymmetric transitivity preserving graph embedding. In Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016.
- Divya, P and Aiswarya, V. (2021). Deep Learning: Techniques and Applications. Journal of Emerging Technologies and Innovative Research(JETIR), 8(7), 125–129.
- Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi han, Hongyu Zhang, and Dongmei Zhang. 2021. CoCoSUM: Contextual Code Summarization with Multi-Relational Graph Neural Network. J. ACM 1, 1 (July 2021), 24 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>
- Wei Huang, Xudong Ma, Haotong Qin, Xingyu Zheng, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu and Michele Magno. How Good Are Low-bit Quantized LLAMA3 Models? An Empirical Study. April 2024
- Choi, Y., Bak, J., Na, C., & Lee, J.-H. (n.d.). Learning Sequential and Structural Information for Source Code Summarization.
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Findings of the Association for

- Computational Linguistics Findings of ACL: EMNLP 2020, 1536–1547. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- Gao, S., Gao, C., He, Y., Zeng, J., Nie, L., Xia, X., & Lyu, M. (2023). Code Structure-Guided Transformer for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology*, 32(1). <https://doi.org/10.1145/3522674>
- LeClair, A., Haque, S., Wu, L., & McMillan, C. (2020). Improved code summarization via a graph neural network. *IEEE International Conference on Program Comprehension*, 184–195. <https://doi.org/10.1145/3387904.3389268>
- Su, C. Y., & McMillan, C. (2024). Distilled GPT for source code summarization. *Automated Software Engineering*, 31(1). <https://doi.org/10.1007/s10515-024-00421-4>
- Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., ... Scialom, T. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. <http://arxiv.org/abs/2307.09288>
- Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. <http://arxiv.org/abs/2109.00859>
- Zhang, C., Wang, J., Zhou, Q., Xu, T., Tang, K., Gui, H., & Liu, F. (2022). A Survey of Automatic Source Code Summarization. In *Symmetry* (Vol. 14, Issue 3). MDPI. <https://doi.org/10.3390/sym14030471>
- Zhang, J., Wang, X., Zhang, H., Sun, H., & Liu, X. (2020). Retrieval-based neural source code summarization. *Proceedings - International Conference on Software Engineering*, 1385–1397. <https://doi.org/10.1145/3377811.3380383>
- Zhou, Y., Shen, J., Zhang, X., Yang, W., Han, T., & Chen, T. (2022). Automatic source code summarization with graph attention networks. *Journal of Systems and Software*, 188. <https://doi.org/10.1016/j.jss.2022.111257>

Appendix A

```
%%capture
# Installs Unsloth, Xformers (Flash Attention) and all other packages!
!pip install "unsloth[colab-new] @ git+https://github.com/unslothai/unsloth.git"
!pip install --no-deps xformers trl peft accelerate bitsandbytes
from unsloth import FastLanguageModel
import torch
max_seq_length = 2048 # Choose any! We auto support RoPE Scaling internally!
dtype = None # None for auto detection. Float16 for Tesla T4, V100, Bfloat16 for Ampere+
load_in_4bit = True # Use 4bit quantization to reduce memory usage. Can be False.
fourbit_models = [
    "unsloth/mistral-7b-v0.3-bnb-4bit",    # New Mistral v3 2x faster!
    "unsloth/mistral-7b-instruct-v0.3-bnb-4bit",
    "unsloth/llama-3-8b-bnb-4bit",        # Llama-3 15 trillion tokens model 2x faster!
    "unsloth/llama-3-8b-Instruct-bnb-4bit",
    "unsloth/llama-3-70b-bnb-4bit",
    "unsloth/Phi-3-mini-4k-instruct",     # Phi-3 2x faster!
    "unsloth/Phi-3-medium-4k-instruct",
    "unsloth/mistral-7b-bnb-4bit",
    "unsloth/gemma-7b-bnb-4bit",          # Gemma 2.2x faster!]
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/llama-3-8b-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit,)
model = FastLanguageModel.get_peft_model(
    model,
    r = 16, # Choose any number > 0 ! Suggested 8, 16, 32, 64, 128
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0, # Supports any, but = 0 is optimized
    bias = "none",    # Supports any, but = "none" is optimized
```

```

# [NEW] "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
use_gradient_checkpointing = "unsloth", # True or "unsloth" for very long context
random_state = 3407,
use_rslora = False, # We support rank stabilized LoRA
loftq_config = None, # And LoftQ )
alpaca_prompt = """"Below is an instruction that describes a task, paired with an input that
provides further context. Write a response that appropriately completes the request.
### Instruction:
{}
### Input:
{}
### Response:
{}""""
EOS_TOKEN = tokenizer.eos_token # Must add EOS_TOKEN
def formatting_prompts_func(examples):
    instructions = examples["instruction"]
    inputs       = examples["input"]
    outputs      = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        # Must add EOS_TOKEN, otherwise your generation will go on forever!
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return { "text" : texts, }
pass
from datasets import load_dataset
dataset = load_dataset("yahma/alpaca-cleaned", split = "train")
dataset = dataset.map(formatting_prompts_func, batched = True,)
from trl import SFTTrainer
from transformers import TrainingArguments
from unsloth import is_bfloat16_supported
trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,

```

```

train_dataset = dataset,
dataset_text_field = "text",
max_seq_length = max_seq_length,
dataset_num_proc = 2,
packing = False, # Can make training 5x faster for short sequences.
args = TrainingArguments(
    per_device_train_batch_size = 2,
    gradient_accumulation_steps = 4,
    warmup_steps = 5,
    max_steps = 60,
    learning_rate = 2e-4,
    fp16 = not is_bfloat16_supported(),
    bf16 = is_bfloat16_supported(),
    logging_steps = 1,
    optim = "adamw_8bit",
    weight_decay = 0.01,
    lr_scheduler_type = "linear",
    seed = 3407,
    output_dir = "outputs", ), )
# alpaca_prompt = Copied from above
FastLanguageModel.for_inference(model) # Enable native 2x faster inference
inputs = tokenizer(
[
    alpaca_prompt.format(
        "Continue the fibonnaci sequence.", # instruction
        "1, 1, 2, 3, 5, 8", # input
        "", # output - leave this blank for generation!
    )
], return_tensors = "pt").to("cuda")

outputs = model.generate(**inputs, max_new_tokens = 64, use_cache = True)
tokenizer.batch_decode(outputs)

```


Appendix B

```
import glob
import json
import os
from typing import List
import numpy as np
import pandas as pd
import tensorflow as tf
import transformers
from sklearn.model_selection import GroupKFold
from sklearn.utils import shuffle
from tqdm.notebook import tqdm
RANDOM_STATE = 42
MD_MAX_LEN = 64
TOTAL_MAX_LEN = 512
K_FOLDS = 5
FILES_PER_FOLD = 16
LIMIT = 1_000 if os.environ["KAGGLE_KERNEL_RUN_TYPE"] == "Interactive" else None
MODEL_NAME = "microsoft/codebert-base"
TOKENIZER = transformers.AutoTokenizer.from_pretrained(MODEL_NAME)
INPUT_PATH = "../input/codesearchnet"
paths = glob.glob(os.path.join(INPUT_PATH, "train", "*.json"))
if LIMIT is not None:
    paths = paths[:LIMIT]
df = (
    pd.concat([read_notebook(x) for x in tqdm(paths, desc="Concat")])
    .set_index("id", append=True)
    .swaplevel()
    .sort_index(level="id", sort_remaining=False))
df_orders = pd.read_csv(
    os.path.join(INPUT_PATH, "train_orders.csv"),
    index_col="id",
    squeeze=True,
```

```

).str.split()
df_orders_ = df_orders.to_frame().join(
    df.reset_index("cell_id").groupby("id")["cell_id"].apply(list),
    how="right",)
ranks = {}
for id_, cell_order, cell_id in df_orders_.itertuples():
    ranks[id_] = {"cell_id": cell_id, "rank": get_ranks(cell_order, cell_id)}
df_ranks = (
    pd.DataFrame.from_dict(ranks, orient="index")
    .rename_axis("id")
    .apply(pd.Series.explode)
    .set_index("cell_id", append=True))
df_ancestors = pd.read_csv(
    os.path.join(INPUT_PATH, "train_ancestors.csv"), index_col="id")
df = (
    df.reset_index()
    .merge(df_ranks, on=["id", "cell_id"])
    .merge(df_ancestors, on=["id"]))
df["pct_rank"] = df["rank"] / df.groupby("id")["cell_id"].transform("count")
df = df.sort_values("pct_rank").reset_index(drop=True)
features = get_features(df)
df = df[df["cell_type"] == "markdown"]
df = df.drop(["rank", "parent_id", "cell_type"], axis=1).dropna()
df.to_csv("data.csv")
with open("features.json", "w") as file:
    json.dump(features, file)
df = shuffle(df, random_state=RANDOM_STATE)
for fold, (_, split) in enumerate(
    GroupKFold(K_FOLDS).split(df, groups=df["ancestor_id"])):
    print("=" * 36, f"Fold {fold}", "=" * 36)
    fold_dir = f"tfrec/{fold}"
    if not os.path.exists(fold_dir):
        os.mkdir(fold_dir)
    data = tokenize(df.iloc[split], features)

```

```

np.savez_compressed(
    f"raw/{fold}.npz",
    input_ids=data["input_ids"],
    attention_mask=data["attention_mask"],
    features=data["features"],
    labels=data["labels"],)
for split, index in tqdm(
    enumerate(np.array_split(np.arange(data["labels"].shape[0]), FILES_PER_FOLD)),
    desc=f"Saving",
    total=FILES_PER_FOLD, ):
    serialize(
        input_ids=data["input_ids"][index],
        attention_mask=data["attention_mask"][index],
        features=data["features"][index],
        labels=data["labels"][index],
        path=os.path.join(fold_dir, f"{split:02d}-{len(index):06d}.tfrec"), )
RANDOM_STATE = 42
N_SPLITS = 5
TOTAL_MAX_LEN = 512
BASE_MODEL = "microsoft/codebert-base"
GCS_PATH = KaggleDatasets().get_gcs_path("ai4code-codebert-tokens")
EPOCHS = 10
LR = 3e-5
WARMUP_RATE = 0.05
VERBOSE = 1 if os.environ["KAGGLE_KERNEL_RUN_TYPE"] == "Interactive" else 2
try:
    TPU = tf.distribute.cluster_resolver.TPUClusterResolver()
    tf.config.experimental_connect_to_cluster(TPU)
    tf.tpu.experimental.initialize_tpu_system(TPU)
    STRATEGY = tf.distribute.experimental.TPUStrategy(TPU)
    BATCH_SIZE = 64 * STRATEGY.num_replicas_in_sync
except Exception:
    TPU = None
    STRATEGY = tf.distribute.get_strategy()

```

```

    BATCH_SIZE = 4
print("TensorFlow", tf.__version__)
if TPU is not None:
    print("Using TPU v3-8")
else:
    print("Using GPU/CPU")
print("Batch size:", BATCH_SIZE)
for i, (train_index, val_index) in
enumerate(KFold(n_splits=N_SPLITS).split(range(N_SPLITS))):
    if TPU is not None:
        tf.tpu.experimental.initialize_tpu_system(TPU)
    train_filenames = np.ravel(
        [ tf.io.gfile.glob(os.path.join(GCS_PATH, "tfrec", str(x), "*.tfrec"))
          for x in train_index ])
    steps_per_epoch = count_samples(train_filenames) // BATCH_SIZE
    train_dataset = get_dataset(train_filenames)
    val_filenames = np.ravel(
        [tf.io.gfile.glob(os.path.join(GCS_PATH, "tfrec", str(x), "*.tfrec"))
         for x in val_index])
    validation_steps = count_samples(val_filenames) // BATCH_SIZE
    val_dataset = get_dataset(val_filenames, ordered=True, repeated=False, cached=True)
    with STRATEGY.scope():
        model = get_model()
        total_steps = steps_per_epoch * EPOCHS
        warmup_steps = int(WARMUP_RATE * total_steps)
        optimizer = transformers.AdamWeightDecay(
            learning_rate=WarmupLinearDecay(
                base_learning_rate=LR,
                warmup_steps=warmup_steps,
                total_steps=total_steps, ),
            weight_decay_rate=0.01,
            exclude_from_weight_decay=[
                "bias",
                "LayerNorm.bias",

```

```

        "LayerNorm.weight", ],)
    model.compile(loss="mae", optimizer=optimizer)
    metrics = model.fit(
        train_dataset,
        steps_per_epoch=steps_per_epoch,
        validation_data=val_dataset,
        validation_steps=validation_steps,
        epochs=EPOCHS,
        verbose=VERBOSE,
    ).history
    model.save_weights(f"model_{i}.h5")
    break
metrics = pd.DataFrame(metrics)
go.Figure(
    data=(
        go.Scatter(x=metrics.index, y=metrics["loss"], name="train"),
        go.Scatter(x=metrics.index, y=metrics["val_loss"], name="validation"), ),
    layout=dict(
        width=600,
        title_text="Model loss",
        xaxis_title_text="Epoch",
        font=dict(size=16), ),)

```