Code Structure Capturing and Recommendation

Final Year Project

Session 2022-2026

A project submitted in partial fulfillment of the
requirements for the Degree

of

BS in Computer Science

Department of Computer Science
COMSATS University Islamabad (CUI), Lahore Campus

Project Details

Project ID (for office use)	
Type of project	[✓] Traditional [ ] Industrial [ ] Continuing
Nature of project	[ ] Development [✓] Research & Development
Sustainable Development Goals(SDGs)	[ ] Good Health and Well-Being [ ] Quality Education [✓] Industry, Innovation, and Infrastructure [ ] Gender Equality [ ] Decent Work and Economic Growth [ ] Climate Action
Area of specialization	[✓] Artificial Intelligence (AI) [ ] Blockchain [ ] Cybersecurity [ ] Data Science and Analytics [ ] Game Development [ ] Internet of Things (IoT) [✓] Natural Language Processing (NLP) [ ] Mobile App Development [ ] Web Development

Project Group Members

Sr.#	Reg. #	Student Name	Email ID	Signature
(i)	Group Leader	Zawar Ahmed Farooqi	sp22-bcs-109@cuilahore.edu.pk	
(ii)		Syed Ali Arsal	sp22-bcs-037@cuilahore.edu.pk	
(iii)				

Declaration: The candidates confirm that the work submitted is their own and appropriate credit has been given where reference has been made to the work of others.

Plagiarism Free Certificate

This is to certify that, I am Zawar Ahmed Farooqi S//o Muhammad Zubair Farooqi, group leader of FYP under registration no CUI / SP22-BCS-109 /LHR at the Computer Science Department, COMSATS University Islamabad, Lahore Campus. I declare that my FYP proposal is checked by my supervisor and the similarity index is _____% that is less than 20%, an acceptable limit by HEC. The report is attached herewith as Appendix A. Date: 19 December Name of Group Leader: Zawar Ahmed Farooqi Signature: _____

Name of Supervisor: Dr Junaid Akram Co-Supervisor (if any):_____
Designation: _____ Designation: _____

Signature: _____ Signature: _____

HoD: _____

Signature: _____

ABSTRACT

Software engineers face considerable difficulty and extended time when analyzing code developed by other programmers because program structure and logical elements remain unclear. It becomes difficult to understand how various code components work together because missing documentation together with unfamiliar coding practices. This limitation affects both academic learning environments and professional workplaces where engineers must quickly get familiar with large codebases in order to maintain extend or improve them. When developers do not have clear structure or readable explanations they spend unnecessary time scanning files and connecting the logic manually which leads to delayed progress and increased complexity in routine tasks.

ClarifAI produces Abstract Syntax Trees and Control Flow Graphs to demonstrate the inner structure and the logical flow of the Java programs. Such visual displays assist users to track the order in which the code is executed and find relevant aspects with a lot of certainty. The system further makes use of the CodeT5 model to generate succinct descriptions of each structural unit that helps in comprehending without any knowledge of programming at the expert level. The combination of the diagrams and summaries creates a single solution which bridges the gap between the code navigation and the code comprehension through the provision of an accessible and interactive environment. By formatting Java programs in a more descriptive and well structured manner ClarifAI would be beneficial to software engineers researchers and students who are expected to frequently read given projects. The general idea of the system is the reduction of the time spent on learning the source code, as well as to have a useful assistant, which will help a user navigate through intricate structures and obscure relationships. In this way the tool enhances productivity and promotes learning success and makes codes better accessible to the people operating in the academic and corporate environment.

Acknowledgement

We would like to sincerely thank Dr. Junaid Akram for his unwavering leadership and assistance during this project. His wise counsel, encouragement, and patience have been crucial to the successful completion of this work, from helping us comprehend and fine-tune the project scope at the outset to offering insightful feedback at every stage of development.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

In this chapter, we presents the main idea behind the creation of the system and reasons why the simplification of understanding Java code has become a must need of students as well as professional developers. It identifies which area of the problem the project is intended to resolve and also identifies the issues which come up when the developers deal with unknown or unwritten code. Chapter also introduces the overall aims and scope of the work and the importance of the creation of a tool that would be able to integrate the structural visualization and the automated code summarisation. This chapter establishes a context and lets the reader understand the direction of the project, thus setting the stage of the detailed analysis and design discussions.

1.1 Introduction

Understanding someone else's code has always been one of the hardest things in software development [1], especially when the project is big or when several developers have worked on it over different time periods. Most of the time the code does not have proper documentation, or the comments are outdated or too short to explain what is actually going on. This makes it difficult not only for new team members joining a project but also for students who are still learning how to read and understand Java programs. As software systems continue to grow in scale the challenge of understanding unfamiliar projects also increases especially when developers join ongoing teams or when academic learners must study code created by others. The current development is much dependent on teamwork and regular modifications that imply that program logic may evolve as time goes by and documentation may not be complete. A tool that merely gives syntax highlighting or some basic navigation capabilities like IntelliJ [3] or VS Code [4] is fine to help them edit tasks but they do not assist in a user fully grasping the structure and intent. This has led to developers often taking a lot of time to trace relationships between files on different files to gain insight into how various classes work with one another or how a given feature has been coded. This is further complicated by the fact that the code may have embedded logic or abstract code that cannot be easily viewed using conventional editors.

The issue that we noticed very frequently when working on various projects and it is one of the reasons why we decided to use ClarifAI as a final year project. We decided to create something that would assist software engineers to comprehend the Java code quicker without reading it line after line. Our motivation was simply that we have also struggled many times with confusing files, incomplete documentation, and figuring out how classes are linked with each other. By working on this system, we also hoped to learn more about parsing code, visualizing structures, and how modern models like CodeT5 [2] can turn code into readable explanations. Even though tools like IntelliJ [3] and VS Code [4] give basic structure views, they do not actually explain what the code is doing in plain language. Some tools create diagrams but do not generate summaries. Other tools produce summaries but do not help visualize the structure. So the existing approaches solve only a part of the problem. We basically came up with an idea to combine two different things which is viualization and code summaries to showing the structure and explaining the code in a simple way such that user can interacts visually through the structure and understands different parts of it. ClarifAI focuses on Java code and tries to make it easier to understand by breaking it down into parts. The system interprets the uploaded code after which it constructs the hierarchy of classes, methods and the manner of interconnection. We then produce an AST [5] that provides a graphical concept of how the code is organized. We also tested the CodeT5, which is a code related model, and fine tuned it in Google Colab [6] such that it produces superior summaries of Java functions and classes. The combination of these steps was one of our objectives, and users can upload the code, watch the structure, and read a summary without using different tools.

Recent studies indicate that structural awareness has a significant influence on facilitating the interpretation of codes by machines. Research on AST based learning models including ASTNN [10] and hybrid structural semantic models demonstrate that the syntactic and semantic cues should be combined in order to obtain a better knowledge of program behavior. These results argue in favor of the fact that relevant code understanding presupposes structural understanding and semantic clarity that is usually lacking in actual project. Simultaneously model-based transformer models,

such as CodeT5 [2] point to the opportunities of natural language generation to support the developer by extracting complex logic in the form of readable explanations. These methods imply that a system that combines structural visualization and contextual summarization will be able to lower cognitive load on users who have to read or maintain code that was not created by them.

The ClarifAI is inspired by these research directions and translated to a shipped and easy to use setting which operates directly via a web based interface. The goal is to eliminate the impediments that Java programmers encounter when seeking to comprehend Java programs particularly when they are required to get some immediate insight without necessarily reading hundreds of lines of code. The fact that the code is displayed as a clear structure tree with the help of summaries created with the help of CodeT5 system makes the difference between human level interpretation and raw syntax. In this manner, both students and professional developers will be able to have a more comprehensive picture of the functioning of the program and what its principal elements are and how its internal processes unfold.

We divided the entire project into smaller tasks and collected the side requirements in the stage of development. Then through trial and error testing of each individual model individually on whether it will work in general well with Java summarization. Another important step was formatting the output that we got after parsing Javalang since we needed to make the raw format readable to ordinary users despite the fact that it initially appeared to be disorganized. After that, we labored on assembling all these parts, including combining the parts of parsing and summarization and making sure that the AST is properly presented. We continue to test Java files of various kinds, small programmes and of slightly larger size, simply to observe the behaviour of the system in various circumstances. These activities assisted us in obtaining a clear picture of the project as to whether the project was being driven well or not and whether the features were functioning as we anticipated. To illustrate, we would test whether the parser was recognizing the right classes and methods, whether the AST layout was sensible, and whether the summaries themselves were actually useful to explain the code.In the process of doing so, we also learned some things about minor bugs that do not become immediately obvious like what to do with random syntax errors in uploaded files or how to fix small bugs when the diagram failed to load. It was not ideal at the beginning and we needed to make certain changes here and there depending on what we noticed. This form of trial and error is what assisted us in determining the finished version of ClarifAI.

We then paid attention to the machine learning model that will produce significant explanations of the code. We started by combining the Code Trans T5 based pretrained model that assisted us in trying to find out the effectiveness of an available model to summarize Java programs. In the early tests we found that the model would generate only one generic comment on the complete file no matter what the internal structure of the file was. This implied that a full Java program was given to us, the model will give just a single brief description which will not show the various classes and methods in the code. In order to seal this gap a procedure was devised to isolate the code into smaller rational units in terms of classes and methods. Counting brackets and finding the patterns of naming were some of the techniques that we used to ensure that each part of the file is processed individually. This enabled us to make more specific explanations of individual elements rather than providing the user with a single general explanation. Once we had reached this degree of organization we proceeded to further perfection of the quality of the generated comments and their clarity. For this purpose we fine tuned the model using the CodeXGLUE code to text dataset created by Microsoft because it offered high quality pairs of source code and natural language descriptions. Fine tuning helped the model understand the common patterns structure and behavior of Java methods which improved its ability to generate meaningful summaries for both simple and complex functions. Once the model was able to produce precise outputs than before, we connected the generated comments directly to the AST. This integration let users to interact with different nodes of the AST and see the explanations attached to the selected class or method. Through this approach the machine learning model became an active part of the visual exploration process making it easier for users to understand how each part of the program works and how different components relate to one another.

In the end our expectations from ClarifAI is to make code understanding faster and easier for anyone who works with Java programs. It should help software engineers who want to review or maintain code without wasting time reading everything manually and it should also help students who are still learning how Java structures work. The whole idea is to give one simple place where a user can

upload code, see the structure clearly, and get readable summaries without switching tools or spending hours figuring things out on their own. We believe this makes the process of understanding code smoother and more approachable and it saves a good amount of time compared to manually going through each file line by line.

1.2 Problem Statement

In the professional environment developers often have to deal with and analyze existing software systems. Getting familiar with code written by other developers usually takes an excessive amount of time and becomes frustrating to deal with. Real world projects share the same problem because source code documentation often falls short of providing meaningful comments without adequate consistency in its documentation. Development in large unwieldy code bodies becomes challenging for both new inexperienced and professional developers because they lack visual maps and contextual understanding of the code. In addition, IDEs and code editors exist in the market but users lack tools that link code structures to automated explanations for source code understanding. Current coding tools build either code diagram generators or give fundamental static code inspection capabilities that do not explain component functions. The current market absence of a mapping and explanatory solution for code structure demonstrates an obvious need for a system which combines diagram generation with accelerated understanding capability. In order to facilitate quicker and more precise comprehension of current codebases, a Java-focused tool that connects code structure with automatic, human-readable explanations was needed.

1.3 Proposed Solution

To solve this problem we have developed a tool named **ClarifAI.** ClarifAI is a web-based tool which analyzes Java projects and integrates AI-generated code Comments and structural visualization into a single platform. ClarifAI applies its framework to work with Java code while constructing Abstract Syntax Trees (ASTs) combined with CFG diagrams to present graphical class mappings and object connections as well as code structure organization. Visual components in these representations aid users to travel through complex system architectures which enables them to understand higher-level component interactions within the software framework. ClarifAI employs CodeT5 as its summarization tool which utilizes transformer-based technology to handle code understanding tasks and code generation responsibilities. The tool serves software engineers and computer science students well because they must review, expand or improve existing systems. The visual mapping combined with artificial intelligence-generated explanations enhances code readability as it deepens users' understanding of software design and architecture as well as increase work efficiency for both professionals and students.

1.4 Main Objectives

The main objectives of our proposed system are

- To develop a platform that enables users to examine Java source code through simple directory upload or direct code snippet entry. No prior coding skills or tool experience is necessary because the platform provides easy access to users through its interface.
- To generate structural visualizations of Java code using ASTs and CFG diagrams, allowing users to quickly see classes, objects, and overall architecture.
- To produce contextual natural language summaries of classes and functions using the CodeT5 based model, replicating professional style code documentation.
- To support flexible analysis workflows by enabling both complete project uploads and manual code snippet input based on user requirements.
- To assist software engineering students and professionals in learning, code review, debugging, and system understanding.

- To reduce manual effort and time spent interpreting existing Java codebases through automated analysis and summarization.

1.5 Assumptions & Constraints

This section describes the main assumptions related to the system design as well as the currently existing restrictions on the operational capabilities of the ClarifAI system. These factors specify the anticipated context, behavioral interaction of the users, and the limits of the system.

1.5.1 Assumptions

User Familiarity with Web Tools Software engineers and computer science students are expected to use web-based capability when handling programming outputs for upload and live examination and interaction.

Java Codebase Input The application operates only with Java program source code from its current design standpoint. Users are expected to follow typical Java file organization standards when they upload their Java source code.

Manual Code Input Options Java users can submit their projects by complete folder upload or through direct paste input into the web portal interface. Users need to recognize which input option suits their requirements and they should choose an appropriate method accordingly.

Stable Internet Access No interruptions are expected for ClarifAI users because the application functions as a complete web-based system that requires constant internet service for continuous interaction with real-time summaries and structural diagram visualization.

Interest in Code Understanding The tool can helps the users who want to understand or analyze unfamiliar code while learning from them as well as displaying its summarization and visualization.

Use in Academic and Professional Environments The tools can be used efficiently in educational and professional software teams who had a need to evaluate and implement the code which is written by others.

1.5.2 Constraints

Language Limitation ClarifAI processes only Java programming language through its platform. Only users working with the Java programming language can currently access ClarifAI features because it does not support multi-language text input.

Model Dependency The quality along with appropriateness of the code summaries reacts to the operational efficiency of the CodeT5 model which serves as the underlying foundation. The training purpose of CodeT5 model for code understanding does not prevent occasional imprecise responses in edge cases together with very complicated code situations.

Diagram Generation Limitations The CFG and AST visualization results primarily depend on how accurately the input programming code has been provided. The production of substandard diagram models will result from problematic code formatting and syntactic problems in the input.

No Real-Time Collaboration Shared annotation capabilities as well as real-time project analysis between several users remain unavailable in its current state.

Security and Privacy of Code Uploading proprietary code through users creates an issue regarding both data privacy along with secure management of information. After the processing phase measures should exist to prevent the storage or disclosure of uploaded code.

No Customization of Output Users must deal with fixed summary and diagram detail levels in the current version since advanced users retain limited flexibility to view detailed or specialized findings.

Browser Compatibility The application operates under the assumption of using contemporary browser technology. Users can experience issues during operation when working with either outdated browsers or restricted device settings.

Scalability The platform needs to be able to work effectively to support many concurrent users.

1.6 Project scope

The ClarifAI requires complete web-based tool development to capture Java source code structures and create code summaries which enhance code comprehension with visualization. Users should access an internet based platform through their web browser for submitting Java code projects or pasting code snippets to receive analysis. Moreover, the software should use Abstract Syntax Trees (AST) for automatic structure analysis to break down classes, methods and their relationships. Additionally, CFG diagrams need to be generated in a comprehensible format to display classes with their relationships and object communication. The application will incorporate transformer-based model technology which will generate human-like condensed summaries for both classes and functions. Moreover, A single user friendly interface enables users to see modular diagrams jointed with automatic text summaries. The current version focuses on Java language support to keep up optimal performance and dependability of model operation. The system lets users handle files through a safe web or server environment while erasing all user-related data after each process. Furthermore, the system should demonstrate both responsiveness and cleanliness through its user interface design to ensure effective student learning combined with professional code evaluation. The main target audience will be software engineers either on professional level or students level. The scope of the project does not include other languages instead of java. Also there is no real-time collaboration, code editing, or automatic bug fixing / performance tuning.

1.7 Software Development Lifecycle Model

Since Clarify is research and development based which includes research as the main objectives, we are using Rapid Application Development (RAD) mode. We are using this development life cycle as we have to quickly develop component after doing relative research and make adjustments because the requirements weren't fully clear in the start. RAD prioritizes short development cycles, rapid prototyping, and regular stakeholder feedback over lengthy upfront analysis and design. Since we have to experiment with some technologies and integrate with our selected model in response to supervisor feedback and develop a functional system by certain academic deadline.

We rapidly added functionality during the building phase by combining existing tools and technologies with custom components that we developed ourselves. We have done various experiments with transformer based T5 models to produce code comments. We have also switched from a basic HTML/CSS front end to a Flask-based web application, and integrated Python and Java tools for parsing and analysis. Implementation, testing using sample Java projects, and small adjustments based on the outcomes were all part of each iteration. In the last stage, we focused on getting the system stable enough. We put it on a suitable server, ran full end-to-end tests, and tried it with real-like examples. Working with the RAD model helped us keep moving forward, handle changes in ideas and design, and still finish a complete, working ClarifAI prototype on time.

Chapter 2. Requirement Analysis

In this chapter, the general requirements that were used to design and develop the system will be outlined. It provides a summary of the background research conducted to learn about the available tools and research, and how the results affected the ultimate features of the project. The chapter goes further to outline the functional and non functional requirements that outline what the system is supposed to offer and how it is supposed to perform. These requirements are the basis of future design choices and make sure that the system is in line with the needs of the users as well as the project goals.

2.1 Literature Review

Over the last several years, many tools and research studies have tried to make sense of large and complex source code bases, mainly because developers still struggle when documentation is missing or outdated. A lot of early work in this space borrowed ideas from natural language processing, where models like BERT [7] and other masked language model approaches were designed to understand text by predicting missing tokens. NLP models were firstly used only for normal languages and not the coding languages but they were later applied to programming languages as well but still many tools treat source code as plain text and this is one of the limitation due to which many models fails to capture the semantic. Due to this many researchers move towards structure aware recognition of code For example, early systems like Code2Vec [8] and Code2Seq [9] used to learn from AST paths but they required high processing power and did not scale well to large codebases. Then comes AST based neural network approach such as ASTNN [10] which tried to preserve syntactic structure more effectively by breaking code into statement level trees.

Some tools that concentrated on documentation and visualization are also there but they are limited as well. Among the mentioned tools, one of them, called SonarQube [11], is being utilized now to capture code smells, code bugs though not much useful to the developers to have a clear picture of the code structure. CFG class diagram can also be generated by diagram generation tools such as PlantUML [12] but they require manual annotation and also fail to provide the explanation of the code. Very popular IDEs like the IntelliJ IDEA, Eclipse, as well as VS Code [4], offer syntax highlighting, navigation and diagrams based on the use of a plug in, nevertheless, do not offer semantic summarization and are overwhelming to novice learners.

The application of deep learning to combine structural and semantic data has been explored in recent research. ComFormer [13] is a syntactic semantic dual attention transformer architecture in order to make better summaries. Other models focused on unified encoder-decoder models. Architectures such as PLBART [14] and CodeT5 [2] are built on NLP systems such as T5 [15] and are particularly good at both code generation and code comprehension. The extensions include additional syntactic and semantic information such as CodeT5+ [16] and AST-T5 [17] and show improved performance, but need substantial computational resources. Structural extraction tools such as srcML [18], Tree-Sitter [19], and Spoon to Java [20] also prioritize the importance of being very precise in capturing the code structure before analyzing it. Even though they do assist developers, not all visualization oriented tools, such as CodeSee [21] and Eclipse AST Viewers [22], are at this point in time able to provide automatic natural language explanations.

In general, it can be seen that the literature demonstrates a well-defined trend in that text-based models fail to do so due to the fact that they disregard structural information, whereas structural only tools are unable to describe how codes behave. ClarifAI fills this gap as it merges both the structure extraction and summarization of Java with a lightweight web-based application that is not heavy in terms of installation and preprocessing.

Table 2.1 summary of related work

Ref no	Study / Tool Title	Research Problem / Objective	Methodology	Shortcomings
[7]	BERT	Improve contextual language understanding through bidirectional masked language modeling.	Masked Language Model (MLM) + Next Sentence Prediction (NSP) pre-training.	Treats code as plain text; does not capture structure or syntax required for code comprehension.
[9]	Code2Seq	Generate sequences (summaries) from structured AST paths.	Path based encoding over AST to sequence decoder.from ASTs and encode them into embeddings, used for prediction tasks.	Summaries are brief and low-detail; struggles with large, real-world projects.
[10]	ASTNN (AST-Based Neural Network)	Improve source code representation by capturing syntactic structure.	Splits code into AST fragments and encodes each fragment with recursive neural networks.	Does not incorporate semantic or contextual information; no visualization support.
[11]	SonarQube	Improve code quality by detecting bugs, vulnerabilities, and code smells	Static analysis using rule engines and CI/CD integration	No semantic explanations; does not visualize structure; not suitable for code comprehension.
[12]	PlantUML	Generate UML and sequence diagrams from textual descriptions or annotated code.	Parses PlantUML DSL to produce diagrams.	Often requires manual input; no automatic summarization or semantic understanding.
[3]	IntelliJ IDEA / VS Code / Eclipse	Provide comprehensive IDE support for code editing, navigation, and refactoring.	Syntax highlighting, type inference, plugin-based static analysis, code navigation.	Overwhelming for beginners, no automatic code summarization, visualization limited.
[13]	ComFormer	Improve code summarization using both syntactic and semantic cues.	Dual-attention transformer over tokens + AST structure.	High computation cost, requires large datasets, no structural visualization.
[23]	Transforme r	Capture long-range dependencies efficiently using self attention.descriptions or annotated code.	Self-attention architecture without recurrence or convolution.	Performs poorly on fine-grained local patterns; not designed for code structure.

Table 2.1 shows the comparsion of literature review we conducted to identify the scope and usage for our solution.

2.2 Research Gap Identification

High Setup Complexity Numerous current solutions, including CodeSee[21] and IntelliJ with plugins, necessitate extensive setup, configuration, or connection with Git or other version control systems. This limits their accessibility for those who are not familiar with such surroundings or for rapid examination.

Limited Summary Generation Some platforms do not automatically produce semantic summaries or explanations for code logic, even though they provide basic code navigation or inline documentation support. Code2Flow and PlantUML are two examples of tools that can show structure but not code behavior.

Single-Focus Toolsets The majority of current platforms focus on either code editing/navigation or structural visualization (such as UML diagrams and flowcharts), but not both. Because of this division, users need to employ a variety of tools in order to fully comprehend the code.

Lack of web-based options Popular tools are frequently complex desktop programs that need to be installed and have dependencies managed. It is uncommon or difficult to find web-based, easily accessible systems that offer both structure and summary.

2.3 Requirements Elicitation

For ClarifAI, requirements were gathered iteratively by combining several techniques rather than relying on a single source. First, we discussed the initial idea with our project supervisor to understand expectations for a final year project and to clarify the main problem: improving comprehension of existing Java code through visualization and summarization. These meetings acted like informal interviews and helped us define the core features that the system must support, such as Java code upload, structural analysis, CFG generation, and automatic summaries.

Next, we carried out a literature review and an existing-system study of tools such as IntelliJ IDEA, VS Code, SonarQube, PlantUML, Code2Flow, CodeSee, and transformer-based summarization models. This document analysis highlighted what current tools already do well and where they fall short, for example the lack of integrated visual structure and natural language explanations in a single web based platform. From these findings we derived additional requirements related to usability, web access, performance, and security.

We also used brainstorming sessions within the project team to convert high level needs into concrete system behaviours. During these sessions we wrote user stories and scenarios, such as a student uploading a Java project to quickly see its class diagram and key method descriptions. Early prototypes of the web interface and summarization pipeline were then shown to our supervisor and classmates. Their feedback on clarity, response time, and ease of navigation was used to refine and reprioritise the requirements.

2.3.1 Functional Requirements

Software Engineering functional requirements convey the description of what the system has to do and how it is expected that each part will conduct itself in a clear and quantifiable manner. These requirements define what the project will be functional and what will be the interaction of the system with inputs of the users and external elements. Through capturing these functions at the early stages of the development process the client and the development team get a common ground of what the system is expected to bring and how it would perform at normal condition.

In the case of this project functional requirements are the basis of all significant workflows such as code input parsing structure extraction summarization and visualization. They also control the design decisions, and all the constituents of the system are aimed at the main task of enhancing the understanding of the code. Every requirement is a feature that the system needs to offer like the ability to accept Java code generating diagrams or be able to give out natural language summaries. All these functions combined make sure that ClarifAI continues to act in accordance with user needs and stay in the right direction as the purpose it was designed to perform at the beginning of the project.

2.3.1.1 Code Upload and Input

First step that includes uploading the code into the system. The user can upload single file as well as whole folder or paste the java code manually into the website.

Table 2.2 Code Upload

FR01-01	The system shall allow users to upload a folder containing Java source code files.
FR01-02	The system shall allow users to paste Java code snippets into a text input area as an alternative to file upload.
FR01-03	The system shall validate the uploaded folder or pasted code to ensure it contains valid Java files.
FR01-04	The system shall only read valid java files

The functional requirements for the code upload and input are listed in table 2.2.

2.3.1.2 Code Parsing and Analysis

Once the code is uploaded the system starts parsing the code to get its organization. The parser examines the syntax and makes sure that the code has legal Java rules. In case syntax error is detected the system will give out a message and halt. In case the code is right the system splits it into meaningful parts such as classes methods and statements. This ductal structure is then utilized in generating AST CFG and subsequent analysis.

Table 2.3 Code Parsing

FR02-01	The system shall parse uploaded Java files using an AST (Abstract Syntax Tree) generator.
FR02-02	The system shall extract classes, methods, and their relationships during AST parsing.
FR02-03	The system shall identify inheritance, method calls, and object instantiations in the codebase.
FR02-04	The system shall store parsed structural information for later visualization and summarization.

Table 2.3 lists the functional requirements for code parsing and analysis.

2.3.1.3 Structure Visualization

The system provides the output structure upon the analysis and parsing of the code. After defining the syntax and elements the system will display a not foil image of classes methods and other specifics hence enabling the user to comprehend the way the code is arranged.

Table 2.4 Visualization

FR03-01	The system shall generate an AST diagram based on the parsed code structure.
FR03-02	The system shall show relationships such as inheritance, association, and aggregation in the diagram.
FR03-03	The system shall provide zoom, pan, and click-to-expand interactions in the AST view.
FR03-04	The system shall allow users to view AST representations of individual classes and methods.

Table 2.4 lists the functional requirements for Structure visualization.

2.3.1.4 Code Summarization

The system produces summaries of all the code parts. All classes and methods are simply explained to make the user realize the meaning of that piece of code. The nature of such summaries is that they enable users to learn the logic as they progressively read all the information.

Table 2.5 Code Summaries

FR04-01	The system shall generate natural language summaries for each class in the codebase.
FR04-02	The system shall generate natural language summaries for each method/function in the codebase.
FR04-03	The system shall display summaries alongside the respective code sections in the UI.
FR04-04	The system shall allow users to copy summaries for documentation or learning purposes.

Table 2.5 lists the functional requirements for code summaries generation.

2.3.1.5 Project Hierarchy Diagram

The system generates a project diagram that is interactive which displays the entire structure of the uploaded project. The users get to learn what the files and components are organized and this helps them navigate easily when handling big codebases.

Table 2.6 Code Hierachy

FR5-01	The system shall generate a tree-based project hierarchy showing packages, classes, and functions..
FR5-02	The system shall display the hierarchy using an interactive visual format.
FR5-03	The system shall highlight currently viewed elements in the hierarchy when selected by the user.

Table 2.6 lists the functional requirements for Project Hierachy Diagram.

2.3.1.6 File/Function-Level Navigation

The system enables users to transfer to other files and other methods with convenience. They are also able to directly jump to any part of the project both in the diagrams as well as in the code view that enhances the speed at which one can explore the code.

Table 2.7 Class/Function-Level Navigation

FR6-01	The system shall list all classes available in the uploaded project.
FR6-02	The system shall allow users to click on a function name to view its structure and summary.
FR6-03	The system shall highlight syntax in the displayed code view to enhance readability.

The functional requirements for the code upload and input are listed in table 2.7

2.3.1.7 Control Flow Diagram CFG

CFG diagrams are created to visulaize the code structure flows and to find any imbuigity in code.

Table 2.8 CFG Diagram

FR7-01	The system shall generate a Control Flow Graph for the uploaded Java source code after successful parsing.
FR7-02	The system shall analyze conditional statements loops and method calls to correctly construct CFG nodes and edges.
FR7-03	The system shall associate each CFG node with the corresponding code block or statement from the source file.

The functional requirements for the code upload and input are listed in table 2.8

2.3.1.8 Saving History of uploaded codes along with their graphs and Tree

All the uploaded code is saved in the history along each user that can be reused.

Table 2.9 History

FR8-01	The system shall store every uploaded source code file for each authenticated user.
FR8-02	The system shall save the generated AST and CFG structures associated with each uploaded code file.
FR8-03	The system shall maintain a history list that allows users to view previously uploaded code files.
FR8-04	The system shall ensure that uploaded code and its related structures are linked only to the respective user account.

The functional requirements for the code upload and input are listed in table 2.9

2.3.2 Non-functional Requirements

Non-functional requirements describe the system behaviour. It represents the quality or attribute of the system.

2.3.2.1 Performance

This part describes the speed at which the system is supposed to react in the process of making summaries and diagrams. The system is not supposed to have long waiting times in order to administer normal Java projects.

Table 2.10 Performance

NFR01-01	The system shall generate code structure diagrams and summaries within 60 seconds for medium-sized Java projects (1,000 lines of code).
NFR01-02	The system shall process code snippets pasted by users with minimal delay.
NFR01-03	The summary generation engine shall maintain a throughput of at least 5 concurrent summarization requests without significant degradation while keeping average response time under 30 seconds.

The non functional requirements for Performance of summaries and diagrams are listed in table 2.10

2.3.2.2 Scalability

This part explains how the system is expected to perform in case of an increase in the size of the project. It will make sure that the application remains functional even when the users upload larger and more sophisticated code bases.

Table 2.11 Scalability

NFR02-01	The system shall be capable of handling increasing project sizes from simple programs (~500 LOC) to large-scale enterprise Java projects (>10,000 LOC).
NFR02-02	The architecture shall support integration with additional language models or visual tools in future without major structural changes.

The non functional requirements for explaining the scalability for increase in the project size are displayed in table 2.11.

2.3.2.3 Usability

In this section, emphasis is laid on the ease with which the system should be understood and used by the students and developers. The design and functionality must be easy and intuitive to learn as it becomes easy to use.

Table 2.12 Usability

NFR03-01	The system interface shall be intuitive for both students and software engineers with basic technical knowledge.
NFR03-02	Tooltips and visual guides shall assist users in interpreting diagrams and summaries.
NFR03-03	Output (summaries, CFG, AST) shall be clearly labeled and organized for quick understanding.

The figure 2.12 displays the non functional requirements for usability of the system.

2.3.2.4 User Friendly

This aspect makes the system have a pleasant and seamless experience. The interface needs to lead the user through the interface so that the user is able to navigate through the code without being confused.

Table 2.13 User Friendly

NFR04-01	The system must allow users to upload folders or paste code snippets easily
NFR04-02	The system should not require the user to install any plugin or packages to run the tool.

The non functional requirements for explaining how user friendly the system is for the users are displayed in table 2.13.

2.3.2.5 Reliability

This part provides the explanation of the stability of the system. It needs to be operating and should give similar results even at large input levels.

Table 2.14 Relaibility

NFR05-01	The system shall maintain 95% uptime during normal operations.
NFR05-02	All code uploaded by users shall be parsed and analyzed accurately, with fallback error handling in case of syntax errors.

The non functional requirements for explaining how reliable system is for the users are displayed in table 2.14.

2.3.2.6 Security

This section specifies the way user code and personal information is to be kept safe. Data should be protected by the system as it is being uploaded and stored.

Table 2.15 Security

NFR06-01	Uploaded code data shall be handled securely without saving user files permanently on the server.
NFR06-02	The system shall not allow remote code execution, and code shall be parsed only in a sandboxed environment.

The non functional requirements for explaining how secure the system is for the users are displayed in table 2.15.

2.3.2.7 Maintainability and Supportability

This part outlines the ease with which the future developers should find it easy to update and maintain the system. The structure has to be articulate such that new members would be able to work upon it without problems.

Table 2.16 Maintainability

NFR07-01	Codebase and documentation shall follow industry-standard commenting and structure to facilitate developer handoff.

2.3.2.8 Availability

This section defines the time in which the system should be available. The platform must be reachable by the users without any lengthy downtime and the disruptions in the service should be reported in a proper manner.

Table 2.17 Availability

NFR09-01	The system shall be accessible 24/7 except during scheduled maintenance.
NFR09-02	A clear message shall notify users during system maintenance windows or downtimes.

The non functional requirements for availability of the system for the users are displayed in table 2.17.

2.3.3 Requirements Traceability Matrix

This section will describe the relationship between each of the requirements and their corresponding design components and test cases. The mapping assists in confirming every requirement and it becomes easy to monitor progress in the process of testing.

Table 2.18 Traceable matrix

ID	Associate ID	Requirement	Use case ID	Test Case ID	Status
1	FR01	Upload Java source code	UC ID 01	TC ID 5.5.1	Pass
2	FR02	Extract and parse the code	UC ID 02	TC ID 5.5.2	Pass
3	FR03	Generate AST Trees and visualize it	UC ID 03	TC ID 5.5.3	Pass
4	FR04	AI-based code summarization	UC ID 04	TC ID 5.5.4	Pass
5	FR05	Project Hierachy Diagram	UC ID 05	TC ID 5.5.5	Pass
6	FR06	Class/Function-Level Navigation	UC ID 06	TC ID 5.5.6	Pass
7	FR07	Display CFG Diagram	UC ID 07	TC ID 5.5.7	Pass
8	FR08	Saving history of uploaded codes and their structures	UC ID 08	TC ID 5.5.8	Pass

The traceability matrix shown in table 2.18 ensures that functional requirement is linked to its corresponding use case and test case. This mapping verifies that all listed requirements are fully implemented and tested as part of the system workflow.

2.4 Use Case Description

A use case description explains how a user interacts with the system to achieve a specific goal. Use case descriptions help clarify system functionality in simple, user-centered terms and ensure that requirements are well understood from the perspective of end users.

2.4.1 Upload Java Code

This use case describes how a user uploads or pastes Java code into ClarifAI so it can be processed by the system.

Table 2.19 Use Case Description 001 Upload Java Code

<table>
<tr><td colspan="3">Use case ID 01</td></tr>
<tr><td colspan="2">Use case name Upload Java Code
Priority High</td><td>Primary Actor Software Engineer
Other Participating Actor None</td></tr>
<tr><td colspan="2">Use Case Summary
Users can upload source code or Folders containing code files.</td><td>Pre-condition
User at the upload interface of ClarifAI</td></tr>
<tr><td colspan="3">Normal Course of Events</td></tr>
<tr><td colspan="3">1. The use case starts when the user navigates to the upload section.
2. Users paste code or upload the file.
3. The code is checked for valid java code.
4. The system stores files temporarily for processing.
5. The system confirms the successful upload and proceed.
6. This use case ends</td></tr>
<tr><td colspan="3">Alternative Path</td></tr>
<tr><td colspan="3">3a. If the file is not valid java file the system prompts a message “Please enter a valid .java file”
3b. If the code field is empty the system prompts the message “Please paste valid Java code”
4a. If folder structure is missing or unreadable, the system prompts: “Upload failed. Please retry”</td></tr>
<tr><td>Conclusion</td><td colspan="2">This use case concludes when theJava code is successfully uploaded</td></tr>
<tr><td colspan="3">Post Conditions</td></tr>
<tr><td colspan="3">The uploaded code is stored, parsed, and ready for structure and summary.</td></tr>
<tr><td colspan="3">Implementation, constraints, and Specifications</td></tr>
<tr><td colspan="3">Use case must support folder uploads through modern browsers.
Use case must support upload of reasonable file size (e.g., <50MB).</td></tr>
<tr><td colspan="3">Use Case Cross References</td></tr>
<tr><td>Includes
File validation</td><td>Extends
Paste code feature</td><td>Exceptions
1a. There is an internet error.
1b. User cancels the upload mid-process</td></tr>
</table>

The use case description for uploading java code is described in table 2.19.

2.4.2 Parsing

This use case covers how the system reads the uploaded Java files and parses them to extract structured information from the code.

Table 2.20 Use Case Description 002 Parsing

<table>
<tr><td colspan="3">Use case ID 02</td></tr>
<tr><td colspan="2">Use case name Code Parsing
Priority High</td><td>Primary Actor System
Other Participating Actor Software Engineer</td></tr>
<tr><td colspan="2">Use Case Summary
System pass through the source code once it is successfully uploaded, to extract the structure.</td><td>Pre-condition
User has successfully uploaded valid Java code or pasted it into the system.</td></tr>
<tr><td colspan="3">Normal Course of Events</td></tr>
<tr><td colspan="3">1. The use case starts when the user clicks “Start Analysis” after upload.
2. The system scans each uploaded Java file..
3. The system uses a parser to analyze the syntax and generate an abstract syntax tree.
4. The system parse through source code to analyze classes, interfaces, methods, fields.
5. Parsed information is stored in a structured internal format for further analysis.
6. This use case ends</td></tr>
<tr><td colspan="3">Alternative Path</td></tr>
<tr><td colspan="3">3a. If parsing fails for any file due to syntax error, the system logs the error “Some files couldn’t be parsed due to syntax issues.”
5a. If no valid components are found in code, the system displays: “No valid Java structures found in uploaded code.</td></tr>
<tr><td>Conclusion</td><td colspan="2">This use case concludes when the Java code has been successfully parsed and transformed into AST.</td></tr>
<tr><td colspan="3">Post Conditions</td></tr>
<tr><td colspan="3">Parsed output is saved and used as input for generating visual structure and textual summaries.</td></tr>
<tr><td colspan="3">Implementation, constraints, and Specifications</td></tr>
<tr><td colspan="3">Use case must be available to the user 24 * 7.
Parsing must be completed within a reasonable time frame</td></tr>
<tr><td colspan="3">Use Case Cross References</td></tr>
<tr><td>Includes
File Reading, AST generation</td><td>Extends
Code Summarization</td><td>Exceptions
1a. There is an internet error.</td></tr>
</table>

The use case description for Parsing code is described in table 2.20.

2.4.3 AST Generation

This use case explains how the system builds an Abstract Syntax Tree (AST) from the parsed code to represent its structure.

Table 2.21 Use Case Description 003 AST Generation

Use case ID: 03		
Use case name AST (Abstract Syntax Tree) Generation **Priority** High	**Primary Actor** System **Other Participating Actor** Software Engineer	
Use Case Summary Creation of Abstract syntax tree from parsed source code enabling structural understanding and further processing.	**Pre-condition** The users have uploaded valid java code and it has parsed successfully.	
Normal Course of Events		
The use case starts when the system receives a Java file or code block. The system uses a Java parser to tokenize the code. The parser builds AST representing the syntactic structure of the code. The system stores the AST in a structured format. The system displays generated AST to the user in a tree view with expandable nodes. This use case ends.		
Alternative Path		
2a. If the code contains syntax errors, the system flags the specific line and displays a message.		
Conclusion	AST is successfully generated.	
Post Conditions		
Tree structure is linked to code components for contextual navigation.		
Implementation, constraints, and Specifications		
Use case must be available to the user 24 * 7.		
Use Case Cross References		
Includes parsing	**Extends** Code Summarization	**Exceptions** **1a.** There is an internet error. **2b.** Parser timeout for large files

The use case description for AST Generation is described in table 2.21.

2.4.4 Code Summarization

This use case describes how the system takes selected code elements and generates natural language summaries for them.

Table 2.22 Use Case Description-004 Code Summarization

Use case ID 04		
Use case name Code Summarizations **Priority** High		**Primary Actor** System **Other Participating Actor** Software Engineer
Use Case Summary This use case involves generating natural language code summaries.		**Pre-condition** The system has successfully parsed the code and generated its AST.
Normal Course of Events		
1. The use case starts when the user selects the node from the AST generated. **2.** The system extracts the relevant code block (e.g., class, method). **3.** The code block is preprocessed. **4.** The preprocessed code is sent to the summarization model (e.g., CodeT5). **5.** The model returns a natural language summary. **6.** The summary is displayed near the original code block for reference. **7.** Use case ends.		
Alternative Path		
5a. If the model fails to generate a summary "Unable to generate summary for this code segment."		
Conclusion	Summarization is completed and displayed	
Post Conditions		
Interactive code structure diagram is shown.		
Implementation, constraints, and Specifications		
Model should support Java syntax and produce grammatically correct English.		
Use Case Cross References		
Includes Code Parsing, Tokenization	**Extends** AST Generation	**Exceptions** **4b.** Model error or timeout

The use case description for summarizing java code is described in table 2.22.

2.4.5 Structural interactive hiarachy diagram generation

This use case covers how the system converts structural information into interactive diagrams that show classes and their relationships.

Table 2.23 Use Case Description-005 Structural Diagram Generation

Use case ID 05		
Use case name Structural hiarachy Diagram Generation **Priority** High		**Primary Actor** Software Engineer **Other Participating Actor** None
Use Case Summary This use case enables users to visualize the structure of uploaded Java source code in the form of a tree for the understanding of class hierarchies, attributes, and relationships.		**Pre-condition:** The system has generated the code summary.
Normal Course of Events		
1. The use case starts when the system has successfully generated the code summary. **2.** The system displays tree structure to extract structural elements. **3.** The diagram is displayed in an interactive viewer. **4.** This use case ends.		
Alternative Path		
2a. If the uploaded code lacks sufficient structural elements, an error message is shown: "Unable to generate diagrams from incomplete or unsupported code."		
Conclusion	The system provides a clear structural diagram that helps the user comprehend class relationships and architecture.	
Post Conditions		
A structural diagram is displayed. Users can select any specific node to get its code summary.		
Implementation, constraints, and Specifications		
Use case must be available to the user 24 * 7. Must support key Java OOP elements like inheritance, interfaces, and encapsulation. Diagram generation should complete within a few seconds for medium-sized codebases.		
Use Case Cross References		
Includes	**Extends**	**Exceptions**
UC-003 (AST Tree Generation)	None	**1a.** Code is not parsed correctly.

The use case description for Structural interactive hiarachy diagram generation is described in table 2.23.

2.4.6 Class/Method-Level Navigation

This use case describes how the system enables users to navigate between classes, functions, and methods within the code and generated visualizations for better comprehension and efficiency.

Table 2.24 Class and Method level Navigation

<table>
<tr><td colspan="3">Use case ID 06</td></tr>
<tr><td colspan="2">Use case name Class/Function-Level Navigation
Priority High</td><td>Primary Actor System
Other Participating Actor
Software Engineer / User</td></tr>
<tr><td colspan="2">Use Case Summary
The system allows users to quickly navigate between files, classes, and individual functions/methods in both the code view and visual diagrams, improving readability and code understanding.</td><td>Pre-condition:
The user has uploaded and parsed valid Java code.
AST and/or CFG diagrams are generated successfully.</td></tr>
<tr><td colspan="3">Normal Course of Events</td></tr>
<tr><td colspan="3">1. The use case starts when the system has successfully generated the code summary.
2. The system displays tree structure to extract structural elements.
3. The diagram is displayed in an interactive viewer.
4. This use case ends.</td></tr>
<tr><td colspan="3">Alternative Path</td></tr>
<tr><td colspan="3">2a. If the uploaded code lacks sufficient structural elements, an error message is shown:
“Unable to generate diagrams from incomplete or unsupported code.”</td></tr>
<tr><td>Conclusion</td><td colspan="2">The system provides a clear structural diagram that helps the user comprehend class relationships and architecture.</td></tr>
<tr><td colspan="3">Post Conditions</td></tr>
<tr><td colspan="3">A structural diagram is displayed.
Users can select any specific node to get its code summary.</td></tr>
<tr><td colspan="3">Implementation, constraints, and Specifications</td></tr>
<tr><td colspan="3">Use case must be available to the user 24 * 7.
Must support key Java OOP elements like inheritance, interfaces, and encapsulation.
Diagram generation should complete within a few seconds for medium-sized codebases.</td></tr>
<tr><td colspan="3">Use Case Cross References</td></tr>
<tr><td>Includes
UC-003 (AST Tree Generation)</td><td>Extends
None</td><td>Exceptions
1a. Code is not parsed correctly.</td></tr>
</table>

The use case description for Class and Method level generation is described in table 2.24.

2.4.7 Display CFG Diagram

This use case explains how the system generates and displays the Control Flow Graph (CFG) for a selected Java file or code block to visualize its execution flow.

Table 2.25 Displaying CFG Diagram

Use case ID: 07	
Use case name Display CFG Diagram **Priority** High	**Primary Actor** System **Other Participating Actor** Software Engineer / User
Use Case Summary This use case covers how the system constructs a Control Flow Graph fromthe parsed Java source code and presents it visually to help users understand the logical flow of execution.	**Pre-condition** The users have uploaded valid java code and it has parsed successfully.
Normal Course of Events	
1. The use case starts when the system receives a Java file or code block. **2.** The system retrieves the parsed structural elements from stored AST output. **3.** The CFG module analyzes loops conditional statements branches and method calls. **4.** The generated CFG is formatted and rendered in an interactive viewer. **5.** This use case ends.	
Alternative Path	
3a. If a code block contains syntax errors or incomplete structures the system displays "CFG generation failed ".	
Conclusion	This use case concludes when the system has successfully generated and displayed the CFG diagram for the selected Java code segment.
Post Conditions	
A complete CFG is available for user inspection	
Implementation, constraints, and Specifications	
CFG must be generated within a reasonable time even for large methods. Visualization must support zooming and panning System must accurately map CFG nodes to original code lines Only valid parsed code will produce CFG output	

The use case description for Displaying Control Flow Diagram is described in table 2.25.

2.4.8 Maintaining History

This use case explains how the system stores past submissions so users can later revisit their code, summaries, and ASTs from a history view.

Table 2.26 Use Case Description-005 Structural Diagram Generation

<table>
<tr><td colspan="3">Use case ID 08</td></tr>
<tr><td colspan="2">Use case name History Maintaining
Priority High</td><td>Primary Actor System
Other Participating Actor
Student, SE</td></tr>
<tr><td colspan="2">Use Case Summary
This use case allows the system to save and manage previously uploaded or pasted code by the user, enabling users to view their code submission history and access summaries or ASTs generated in the past.</td><td>Pre-condition:
The user must have uploaded/pasted code at least once.</td></tr>
<tr><td colspan="3">Normal Course of Events</td></tr>
<tr><td colspan="3">1. Users upload or paste Java code via the web interface.
2. The system parses, summarizes, and generates the AST.
3. The diagram is displayed in an interactive viewer.
4. Users can navigate to the "History" section.
5. User selects a previous submission to view its summary and AST.
6. System retrieves and displays the saved data.</td></tr>
<tr><td colspan="3">Alternative Path</td></tr>
<tr><td colspan="3">1a. If code already exists in the history with the same timestamp or content, the system avoids duplicate entries.</td></tr>
<tr><td>Conclusion</td><td colspan="2">The system provides a clear navigation to access the history of codes.</td></tr>
<tr><td colspan="3">Post Conditions</td></tr>
<tr><td colspan="3">The submitted code, along with its generated summary and AST, is stored and retrievable in a structured format.</td></tr>
<tr><td colspan="3">Implementation, constraints, and Specifications</td></tr>
<tr><td colspan="3">Use case must be available to the user 24 * 7.</td></tr>
<tr><td colspan="3">Use Case Cross References</td></tr>
<tr><td>Includes
UC-003 (AST Tree Generation)</td><td>Extends
None</td><td>Exceptions
1a. Code is not parsed correctly.</td></tr>
</table>

The table 2.26 describes the use case description for Maintaining history.

Chapter 3. System Design/Methodology

The chapter explains the organization of the system and the inner workflow of the system. It describes how the user input should be converted into structured representations and natural language summaries and provides the architectural decisions which determine the way the system should behave. The chapter talks of the design layers that the data flow and frontend backend and machine learning interaction. It also states the rationale, as to why the selected technologies have been adopted and the ways they will be useful in attaining the overall project objectives. These details give the chapter a full picture on how the system works both in the design and methodological view.

3.1 Activity Diagram

Activity diagram visually represents the flow of activities, or steps.

3.1.1 Code Uploads

The flow works as the user opens the web interface and gives the input. The user have two options to provide the input, first is to upload java file/folder by clicking upload button and the second is to code the java code manually and paste it into the website code interface. The system excels towards next step if the code is uploaded successfully.

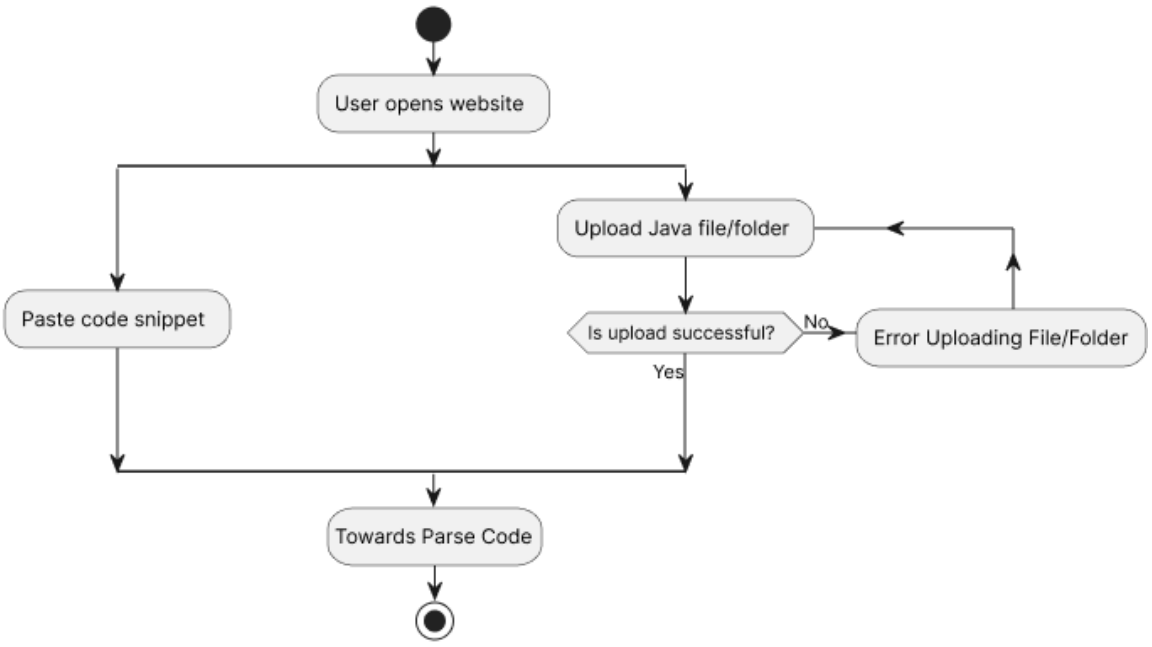

Figure 3.1 Code Uploads

Figure 3.1 shows how the user can choose files or paste Java code and have the system accept it for processing.

3.1.2 Parsing

When the user uploads the code the system parse it to check any syntax errors in it. If the syntax is not correct the use case will end displaying the syntax error message. If the syntax is correct only then the system will perform further operations of AST formation.

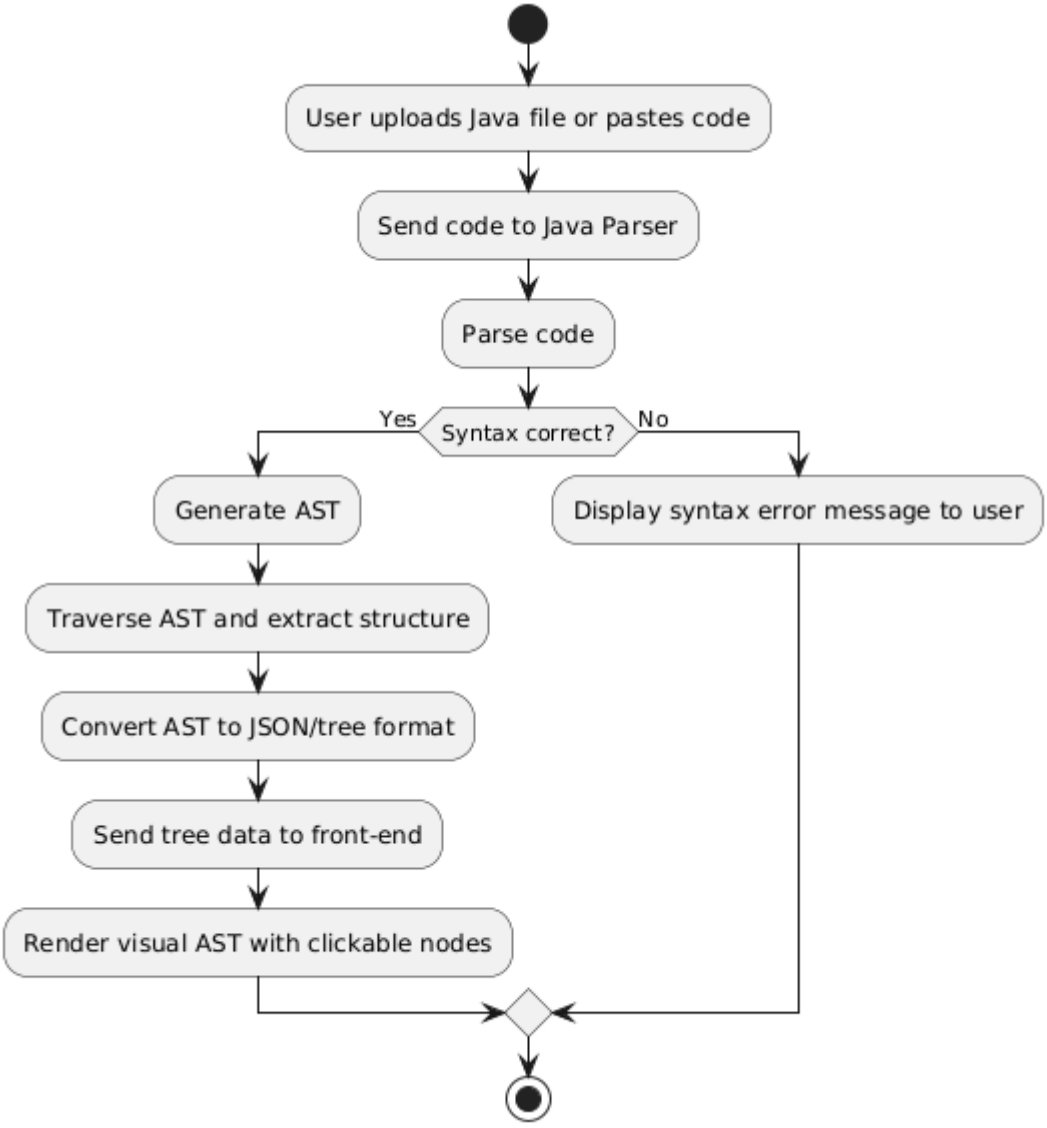

Figure 3.2 Parsing

Figure 3.2 represents reading the uploaded code and converting it into a structured, machine readable format is represented by this activity.

3.1.3 AST Display

After the successful upload of correct code, the system process it and display the interactive tree which displays the connections and flow of code. The user can interact with different components of tree by expanding/collapsing its node, see comments along each node and download the tree.

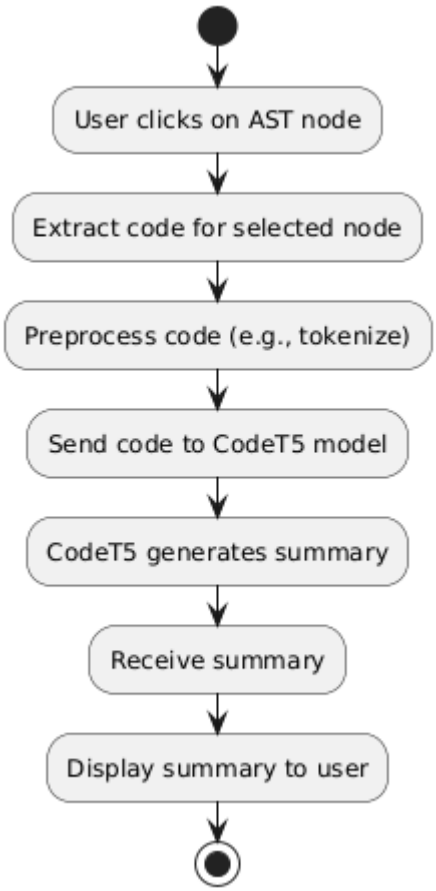

Figure 3.3 AST Integration

Figure 3.3 shows how the user explores the code structure by navigating and interacting with the generated AST nodes.

3.1.4 Code T5 Model Working

The machine learning model uses the code block which are currently fed into in by categorizing methods and classes. The identification of methods and classes are done using multi validation process and then the code blocks goes through encode/decode Transformers that generates the output.

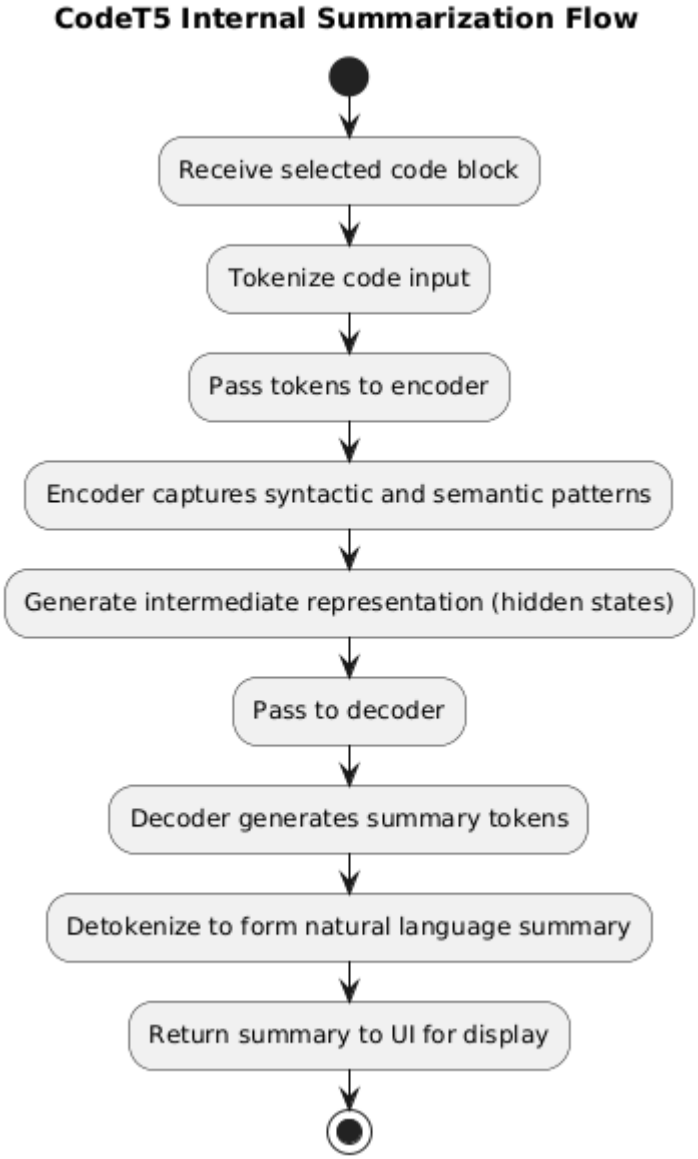

Figure 3.4 Code T5

Figure 3.4 represents how CodeT5 model receives the uploaded code and generates a textual summary in response.

3.1.5 Code Summary

The output from machine learning model is received in form of code summaries, separately for each block. The system validates if the model is working and responded with correct summaries otherwise it diplays error. The generated summaries are then displayed as well as attached with AST tree nodes.

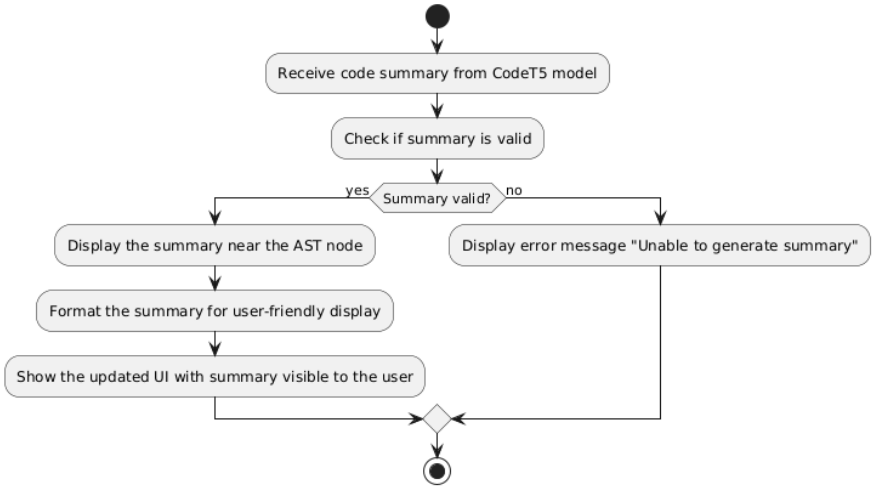

Figure 3.5 Code Summary

Figure 3.5 shows the final display of the generated summary to the user alongside the related code element.

3.2 Software Architecture Diagram

The system follows the layered architectural model which make the clear separation of maintainability and smooth data flow across all the components. The top layer is user interface layer that displays the interactive functionalities like file/code upload, Code editing, Tree visualizing, CFG Diagrams and code summaries. The function of this layer is to facilitate the user through clean and easy to user interface to the user. After this comes the Application layer which handles routing, API endpoints, and request management, ensuring that all frontend requests are processed and delegated to the correct internal modules. Then comes the most important layer which performs AST generation, CFG generation, CodeT5 summarization, and output formatting. Finally comes the Data Layer which maintains storage, including uploaded code, parsed structures, and user history. This layer ensures that previous analyses can be retrieved efficiently without reprocessing, improving overall system responsiveness.

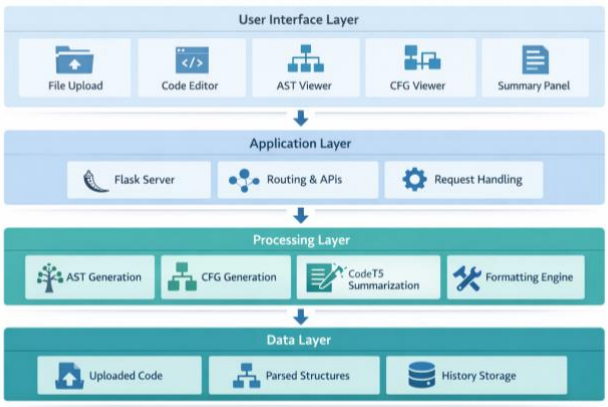

Figure 3.6 Architecture Diagram

Figure 3.6 shows the visual maps of our software system's parts and how they are connected to each other.

3.3 Database Diagram

The database diagram illustrates how ClarifAI manages users, code submissions, and the structural artifacts generated during analysis. All the data including user accounts and authentication details as well as the user's uploaded codes get saved in a Database.

Figure 3.7 Database Diagram

Figure 3.7 shows the database connections of schema in our system.

3.4 Dataset Details

For the dataset part of our project, we mainly worked with two sources that were already established in the code-summarization field. The first one was the CodeSearchNet [24] dataset which is the same dataset used in the CodeTrans work. It contains a large collection of functions from six programming languages including Java. Each Java example pairs a method with its related documentation or comment. The dataset is quite big in size roughly 164k training samples a little over 5k validation, and around 10k test examples just for Java. Since the model we used was originally pretrained on this dataset it already had a broad understanding of how code relates to natural language descriptions which gave us a good starting point.

For fine tuning we picked the CodeXGLUE [25] which is Code to Text (Java) dataset. This dataset also focuses on Java but in a more task specific way. It includes method level code along with paired descriptions. The structure of each entry contains several fields like the repo name, file path, function name, raw code, tokenized versions, docstrings, and some metadata. The dataset is pre cleaned to some extent the code must be valid enough to be parsed into an AST the documentation length is restricted between 3 and 256 tokens non-English comments are removed and any entries containing unusual formatting like embedded images or URLs are filtered out.

3.5 Algorithm/Model Selection

The main goal of our project was to select a model that could comprehend Java code meaningfully and then translate it into natural language summaries without requiring a lot of additional engineering. We did not started with our current model, rather we explored different model which were using different methodologies to generated code comments and summaries. Then to determine the response of Transformer based model we tried a lighter model like T5 base to observe the accuract of its output. There were also some issues in it that includes missing of some important

informations that made us realized that a general NLP model is not enough. We then went on to CodeT5, which is essentially a T5 model that has been trained for programming languages. It is already more knowledgeable about the structure of identifiers and typical code patterns. Since our system relies entirely on producing precise summaries for Java classes and functions, CodeT5 far more closely matched our requirements. Before we fine-tuned it, the version we used had a solid foundational understanding of code and documentation because it had already been pretrained on the CodeSearchNet dataset. We adjusted the model on the Java portion of the CodeXGLUE code to text dataset to better align it with our objective. This dataset is built for summarizing Java methods so it helped the model produce clearer and more useful summaries. The only real issue we faced was GPU limits on Google Colab. Sometimes the runtime stopped or training slowed down because of memory. We adjusted the batch size a few times and restarted sessions when needed. Even with these problems the model trained successfully and the summary quality improved a lot. Before sending anything to the model the Java code goes through a preprocessing step. We parse it with javalang clean the formatting and tokenize it with the T5 SentencePiece tokenizer. These tokens are also used for creating the AST and CFG since both structures depend on the parsed version of the code. This keeps the summaries and diagrams consistent with each other. Inside the Flask application the model is loaded through the Hugging Face Transformers library. We use beam search truncation limits and repetition control so the generated summaries do not repeat or stop abruptly. The model works fast enough for real time use and the accuracy is strong for most Java files we tested.
In the end CodeT5 was the best choice. It gives a balance between good performance and practical usage. It is easy to integrate into our backend and it produces summaries that actually help users understand unfamiliar Java code.

Chapter 4. Implementation

This chapter takes a practical presentation of the development work that was done to develop the system and the way through which each of the major features was developed. It also gives a glimpse of the internals of the logic behind the frontend interface the database integration and the machine learning bits which drive the summarization and visualization functionality. Other key points made in the chapter are the coding techniques and tools that were employed when constructing the system as well as problems that were faced and solutions that were implemented during the process. Describing the implementation choices, this chapter relates the design blueprint to the real working product and demonstrates how the system was introduced in terms of steps.

4.1 Work Breakdown Structure (WBS)

The work breakdown structure demonstrates how the project is broken down into sections that direct the development process. From requirement gathering to deployment, each section reflects a significant phase of the system. In requirements & analysis section all necessary features and constraints are compiled. The architecture and database layout are then planned during the system design phase. The interface and server logic are then the main focus of the frontend and backend development phases. The code processing stage handles AST and CFG creation and formatting tasks. The machine learning integration stage manages model loading and evaluation. The database layer ensures that all stored data is handled safely. The final stage prepares the system for deployment so the project can run smoothly on a live environment.

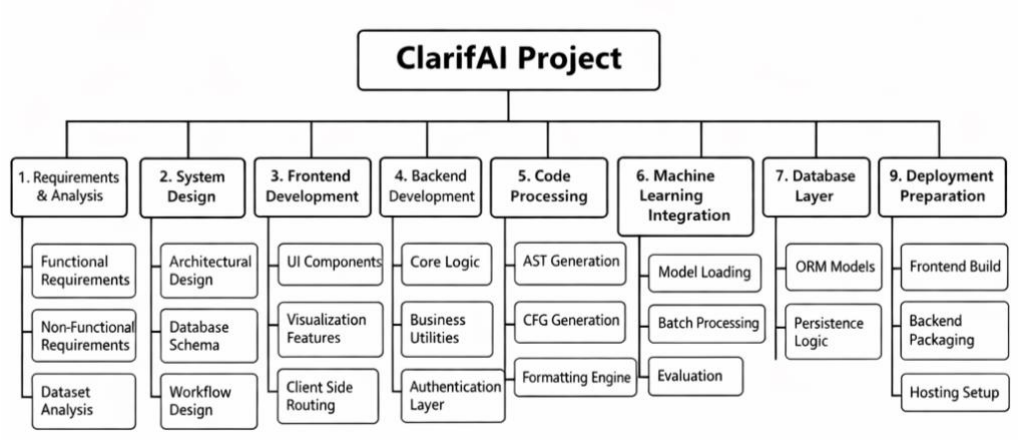

Figure 4.1 Work Breakdown Structure

This breakdown helps show how each part connects and how the project grows step by step from idea to complete working system.

4.2 Team Roles and Responsibilities

We have distributes the roles and responsibilities equally among both member of the group according to expertise of the work associated with the member and then discussed that part with other member for the learning and better understanding of further updates. Here is the mentioned individual tasks.

Table 4.1. Individual Tasks

Team Member	Activity
Zawar, Ali	Initial Research and Planning
Zawar, Ali	Different model testing for summarization.
Ali	Model Training
Zawar	Tree Creation
Ali	Flask Setup
Ali	Web Interface
Ali, Zawar	Component combining and testing
Ali	Model Fine tuning
Zawar	Documentation
Ali, Zawar	Comparison with other systems
Ali	Project Refinement
Zawar	Final Report

Table 4.1 represents the team members responsibilities and their roles for this project

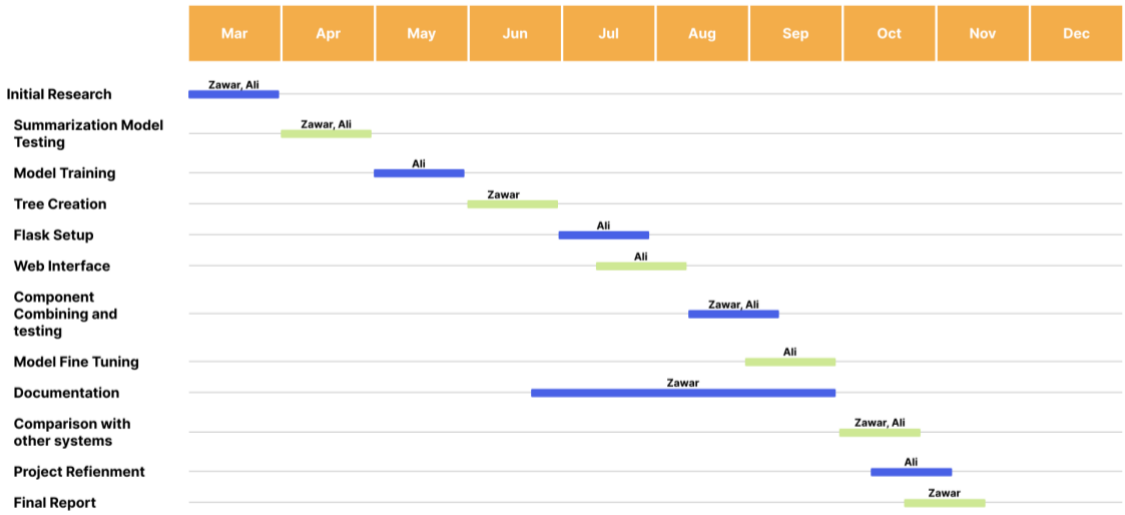

Figure 4.2 Gantt Chart

Figure 4.2 shows the Gantt chart to visually plan, schedule, track, and manage tasks and resources over a timeline, showing what needs to be done, who's responsible, and when it should finish, helping to identify dependencies and monitor progress.

4.3 Tools and Technologies

We used different languages, frameworks and libraries to get our system working the way we wanted because each part of ClarifAI needs something different. Some tools helped with parsing Java code, some helped with generating summaries using our model, and others for developing the web app. Below is what we used and how it fits into the whole system

4.3.1 Programming Languages

Python is the main language on the backend because it works ideally with machine learning models and libraries. We also used it for parsing Java code, generating ASTs, creating CFGs, running the summarization model, and handling all the logic behind the scenes.

4.3.2 Frameworks and Libraries

We built the web server, routed requests, sent data to the model and returned summaries, ASTs, and diagrams using Flask which is our primary backend framework. Fields, and the general structure can all be extracted from Java source files using Javalang. The extracted data is then cleaned and arranged using our own custom formatting layer to make it readable and compatible with our user interface layout. In order to load and execute our CodeT5 model for producing the natural-language summaries, we also utilized the HuggingFace Transformers library. Tokenization, model execution, and output generation are essentially handled by this library. We used both our own reasoning and the tokens obtained from parsing to create the structural diagrams, which are then utilized to construct both AST and CFG. Javalang is used to create the AST, and Python and simple graph-generation tools are used to create the CFG.

4.4 Implementation Details

ClarifAI was developed using a modular approach, where each component was developed and tested independently before being integrated to produce a complete, cohesive solution. Flask served as our backend framework. Python is used to incorporate the model.

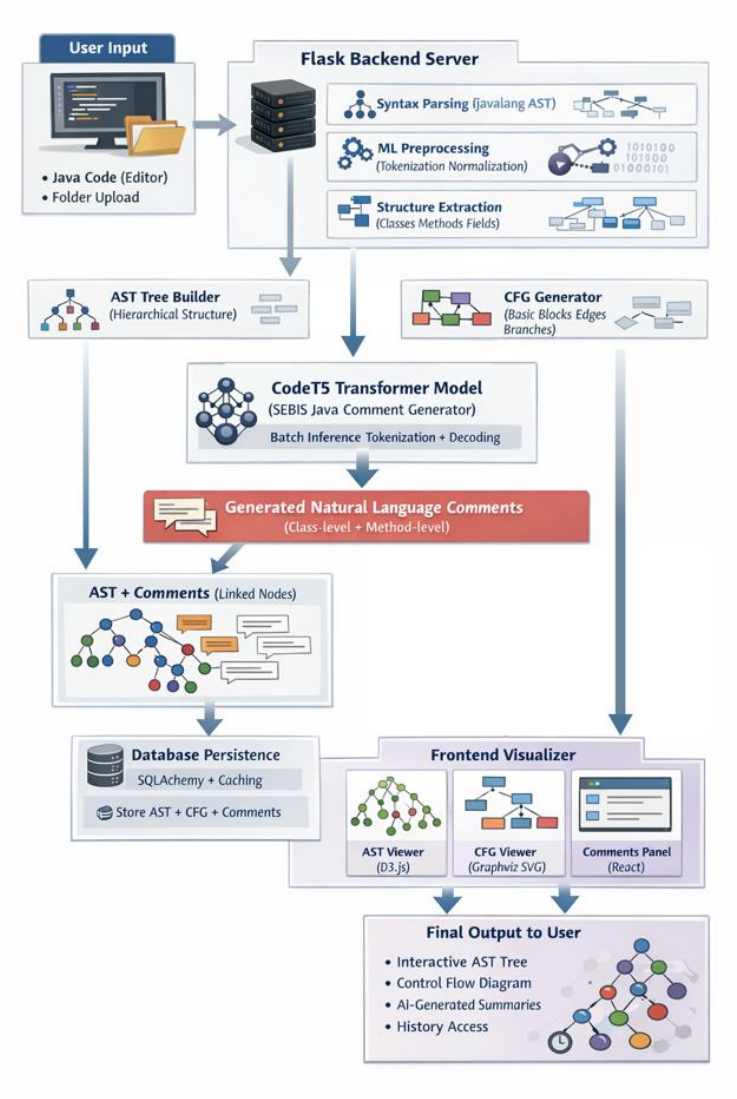

Figure 4.3 System Flow

Figure 4.3 shows the Flow of whole system includes data flow, processing and output.

4.4.1 Flask Integration

ClarifAI backend is effectively located in the middle of the entire system. We developed it with Flask since it is lightweight to develop an experiment quickly and yet has the flexibility to develop as we continued the project as it increased in complexity with the addition of various features. We only required a mere routing layer at the beginning which takes in a Java file and gives out a response but then as we settled on the design we realized that Flask would have to do far more than just that. It was tasked with the coordination between parsing, structure extraction, the CodeT5 model and the front-end interface, almost the traffic controller to the project. The first thing that receives code whenever a user uploads a Java file or a pasted code is Flask. The request is first validated by a basic process in which we test simple aspects like file type, empty input, encoding problems and situations when the folder structure is damaged. Such a step may seem simple but we were reminded at a very young age that a single missing curly bracket can ruin the whole process when parsing is done without checking and the backend will do its best to handle whatever it can before buffering it onward. After code is deemed to be safe, Flask passes the code to the parsing module that invokes

javalang and subsequently the rest of the structural analysis up to structural analysis of the code proceeds.
The model integration component was a little more difficult. The summarization model is an environment that will be operated in the same backend, and due to the large weights, we needed to ensure that the Flask loads the model once when starting the server. To load it on each request would have been to slow the entire system way down. Accordingly the model and tokenizer are stored in memory and Flask provides a clean API endpoint on which a code snippet is entered and the generated summary is given. The manner in which the front end does not have to interact directly with the model as well as unnecessary overhead is avoided. The backend architecture is the way it developed as we experimented with various solutions. It was incredibly basic, but as more features were introduced, Flask became the glue that holds all the moving pieces together, and they all operate in the correct sequence. Although the system has shifted to parsing, structural generation, summarization and visualization, the backend remains quite lightweight since we have attempted to keep tasks separate and simply have Flask coordinate the flow.

4.4.2 Code Parsing and Structural Extraction

One of the most significant components of ClarifAI is the parsing layer as all the other features are reliant on the ability of the system to recognize the raw Java code. Initially we thought that this step would be easy because javalang already has an inbuilt parser but we in the process of coming up with something that would actually work with real world repositories realized that the process of parsing and extracting structure is a lot more complex than we initially thought. The various coding styles, non-standard formatting, missing brackets, nested classes, anonymous functions, and random comments in all cause small problems, which multiply unless the parsing layer is approached cautiously. For the main parsing operation we use javalang which performs lexical analysis and structural breakdown of Java files. It helps us detect classes methods fields inheritance interfaces and other syntactic elements. When the user uploads a folder or a single file the backend sends the content to javalang which tries to convert everything into a parsed structure. Sometimes javalang throws exceptions because the file may contain syntax errors so we wrapped the parsing inside safe blocks that catch failures and return a meaningful message to the user instead of halting the system. Once the raw output comes from javalang the data is not very readable. It contains long nested objects token references and node metadata that do not make sense to a normal user. Because of this we built our own custom formatting engine. This was not part of the original plan but the first time we generated a structural tree we noticed how messy and unorganized the output looked. The formatting layer fixes this issue. It cleans the raw structure organizes class names method signatures field declarations modifiers and parent child relationships into a clear and simple format that the frontend can display without effort. Another challenge was that javalang does not always return relationships in a straightforward parent to children mapping. It might list methods separately from classes or store inheritance information deep inside nested nodes. To solve this we created a structural extraction step that walks through the parsed tree manually reconnects missing links and rebuilds the hierarchy in a way that reflects the real code layout. This same logic later became very useful when we added AST and CFG generation because both rely on accurate class method block and expression mappings.
This extracted structure is then sent to the diagram generation stage that uses it to generate the interactive tree and flow views. We ensured that the formatting process does not omit any critical details since even one absent parameter or an out of the place block would lead to a wrong diagram. Its objective was to generate a stable system that can be easily rendered visually and at the same time be decoded by the front end. The same tokens that are being reused are the ones that are being parsed. We also do not reread the code to generate an AST or CFG and simply pass the structural elements on to the subsequent stages. This saves time in processing and the system is more effective particularly in large multi file projects. Eventually this whole layer would be a combination of automatic parsing and manual assembly. The javalang provides the automated part whereas our custom logic does the reconstruction part of it which makes the extracted structure clean and consistent and is human readable. This layer is referred to as the backbone of the entire system since none of the AST the CFG and the summaries will remain unreliable in case the structure is not right.

Although this section took more efforts than we anticipated it to be is what makes ClarifAI stable as well as reliable.

4.4.3 AST Generation

AST generation element is the key element in translating the interpreted Java source code into a hierarchical map that is structured and reveals the entire syntactic template of the program. Although javalang gives a raw AST the format of its output is crammed with nested code and internal metadata that has minimal usefulness to developers who just need to get a feel of how classes methods and expressions are organized. Due to this weakness we were forced to create our own rebuilding layer over a top of the parser. This layer traverses the original AST only picking out the meaningful nodes of the original AST, making them a class declaration, method signatures, method parameters, control statements, and expressions and then rearranging them into a better human-readable format. The point was to show users a tree that makes some sense instead of displaying them the inner technical workings of the parser. It is also in this reconstruction stage that we are able to normalize node names, consolidate similar constructs and eliminate redundant fragments that only introduce noise. Consequently it produces a clean hierarchical representation of the final AST that is more representative of the real structure of the Java file rather than the implementation of the parser. After preparing the tree structure we make it into a light frontend optimized form that can be drawn immediately in the application interface. The large Java files were one of the initial challenges that we were working with and hence we incorporated some of the interaction facilities such as zooming panning and node collapsing under control. This will ensure the diagram does not take over the screen, and will allow the user the option of only expanding aspects that they need to examine. The interactive action also makes the AST more than a lifeless graphical object. Each time the user clicks on a code snippet, a system will recover the respective code snippet and send it to the summarization engine which then provides a natural language explanation in that context on the sidebar. It is the close relation of structural exploration with the semantic interpretation that causes the AST to be a far more intuitive aid to the understanding of the workings of the code. Practically this is like being in a smart documentation system in which the structure and explanation are co-located. We added the possibility of downloading the AST as high quality SVG or PNG file to enable the reporting and documentation requirements. The files that are exported are optimized in size such that the files are crisp even when incorporated into technical reports research reports or project documentation even when the diagrams are very big. This export option was significant in the sense that most developers desire to have architecture diagrams as part of a design discourse or an academic report. Generally the process AST generation is not only a visualization of the code it also raising the whole analysis process by converting raw source code to user friendly and informative interactive structural map.

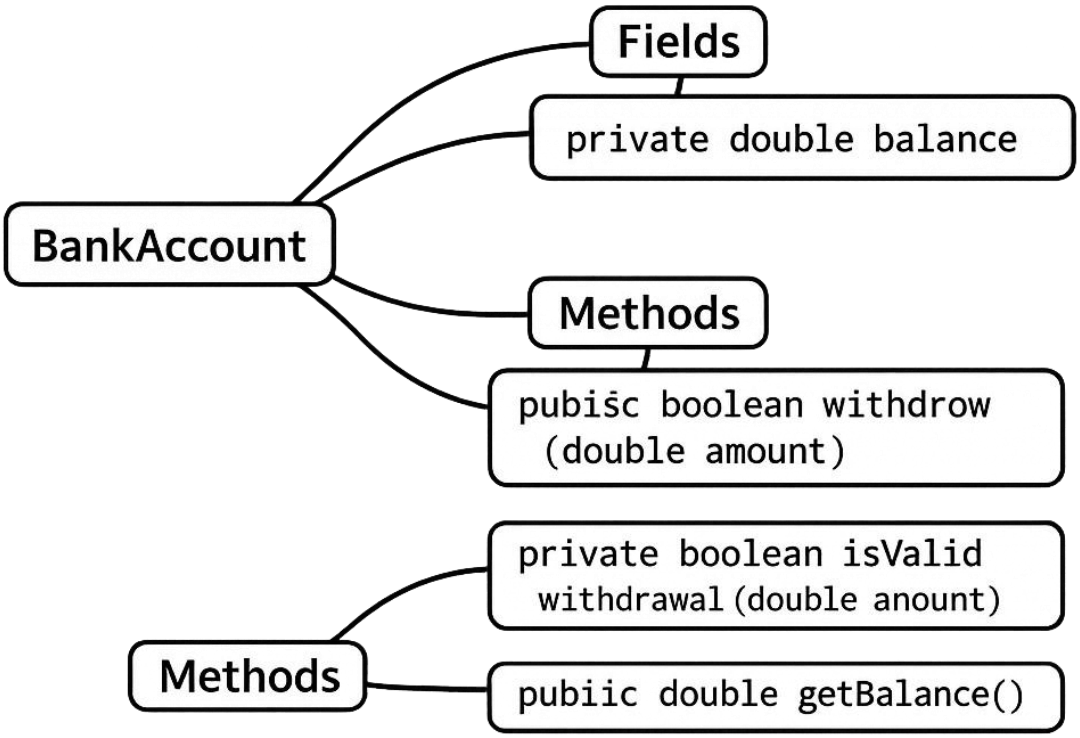

Figure 4.4 AST Graphical View

Figure 4.4 shows the nodes of graphical view of AST Diagrams which are obtained from the root node.

4.4.4 CFG Generation

Control flow generation module is an extension of the system to structural analysis which models the real-life behavior of Java methods. The AST gives an explanation on how the code is written whilst the CFG gives an explanation on how the code is executed and hence it is a very important element in analyzing program logic, possible paths of execution, and complexity of the procedures. Since javalang lacks native support of CFG creation we implemented a new CFG engine. This engine analyzes the output of the parsed result, determines meaningful execution blocks and re-assembles the flow between them in a manner that is true to the Java semantics. It consists of decomposing each method into basic blocks that interpret branches loops jumps and conditionals and then drawing relationships between those blocks to represent all the possible ways the program may execute in the course of its execution. The CFG builder needed development with some more rules in order to address the vast variety of patterns that occur in real Java projects. Nested conditions linked and interlocked logical statements several return points switch cases and loop structures tend to go far beyond the simplicity of a linear graph to the point that the system includes a number of refinement passes that stabilize the resulting structure. Such passes also make sure that the graph will still be readable even when the logic behind it is complex. The output is a CFG representation which is accurate and understandable that reflects actual execution behavior without confusing the user. In the background the generator also clears up the ambiguous control paths, combines identical blocks where suitable and maintains the logical transitions that render the analysis significant particularly in the functions with the branching-heavy logic. After the construction of the execution graph it is converted into a format that is readable by the frontend with each block being linked to a code snippet and metadata. The CFG viewer offers an all-interactive interface, which allows zooming panning and element focusing. Such navigation tools allow one to examine high-complexity methods without losing any context or visualization. Any block that is clicked shows its incoming and outgoing transitions enabling users to follow the flow of control going through the method. The layer of summarization is also linked with each node hence the immediate choice of a block brings to the fore a natural language description elucidating a purpose of that element of logic. Such an

integration will make sure that the users do not merely view the flow but will understand the intent of every branch and block. The system has the capability to save CFG diagrams in high-resolution optimized SVG or PNG files to facilitate reviews of academic documentation design and technical reporting. The resultant exported diagrams can be visually sharp no matter the size of the diagram thus making it appropriate in thesis reports, conference submissions or even internal engineering documentation. These diagrams can be embedded by developers as visual proof of code behavior that can help in onboarding code as well as debugging or analysis of code structure.
The CFG generation module as a whole brings the system to a higher level of more than just the visualization of the system, which gives the system a dynamic execution aware perspective to supplement the AST structure. It provides a connection between syntax and behavior and can provide a clear map on how the program runs, and how decisions on control are taken. The CFG component can be a powerful tool in the quest of any person attempting to know or write formal documentation of complex Java methods through the combination of accurate flow modeling interactive exploration semantic context and professional level exportation.

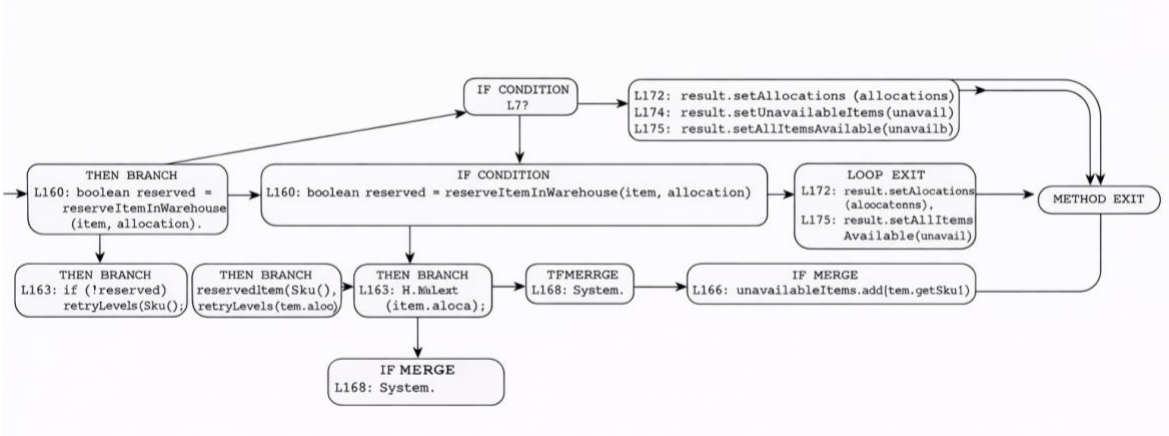

Figure 4.5 CFG Display

The CFG and its logical flows from different components are shown in figure 4.5.

4.4.5 Code Summarization with CodeT5

The summarization part in ClarifAI extends the CodeT5 architecture, which is based on the larger CodeTrans family of transformer-based models that are specifically targeting understanding and generation of text related to source code. Whereas CodeTrans can handle other languages and our system can handle only the Java code to text summarization functionality since it is the most closely aligned with the objective of generating parsimonious method and class level explanations. The premise behind this is simple to design but effective in implementation. The model is fed with a snippet of Java usually a method body or a class definition and will give a human readable summary of the code that is intended to capture the intent and functionality of the code.
CodeT5 adheres to encoder decoder transformer architecture that falls into the same architecture as T5 BERT GPT and other current large scale language models. Transformers are based on self attention mechanism enabling them to comprehend long distance relationship between tokens which is particularly significant in Java where meaningful relationships can be located over several lines and nested structures. The encoder parses the Java sequence and acquires syntax and semantics and the decoder creates the summary bit by bit. CodeT5 has much greater context reason ability over control structures and out-of-domain generalization than older RNN or CNN based models. This renders it a good fit to summarization tasks whereby local details and global behavior are of concern.
The pretrained checkpoint made available under the CodeTrans is the version of CodeT5 that is embedded in ClarifAI. It was trained on span corruption with random segments of the code being obscured and the model trained to reproduce them thus learning more structural and logical patterns. In pretraining the model a sequence length of about five hundred tokens was used which is very much compatible with typical Java algorithms and medium sized classes. In fine tuning the model to ClarifAI we used the Java subset of the CodeXGLUE code to text dataset. Google Colab GPUs

were used to do all fine tuning because transformer training cannot be practically executed on a CPU. Our performance gains were substantial despite having only five thousand fine tuning samples because the performance of transformer models increases as quickly as possible, which indicates how models of transformers can evolve through adjustments to new patterns. Single improvement to design of ClarifAI is the option to produce all summaries right after parsing and not during user interaction. After uploading the code and parsing the system retrieves all the classes and methods and submits them to CodeT5 one at a time storing the results in the meantime. This eliminates recidivism of transformer calls whenever a user clicks on a node in AST or CFG. Consequently the interface is also fast since the summary will be displayed immediately when the user scans the diagrams. This will also decrease the computational load in the backend as the model to be used is not run multiple times during navigation but only once when an upload happens. In the process of summarization ClarifAI employs preprocessing control to create high quality outputs. The Java code is cleaned, normalized and tokenized with the T5 tokenizer that divides Java into subword tokens that are easier understood by the model. In decoding the system beam search with four beams is applied to prevent unstable or excessively general outputs and yet provide the model with the necessary flexibility to make meaningful summaries. After the summary has been produced it is then returned to the frontend along with the structural data in order to display the explanation next to the AST or CFG node that it is a part of. Such a combination provides developers with structural as well as semantic understanding simultaneously.The other benefit of this pipeline is enhancement of good integration with ClarifAI parsing and formatting layers. Since the code already has clean organized pieces due to javalang and our custom extractors the model takes regular well shaped input making it a better summarizer. Transformer models are much more effective with an input in a predictable structure and that preprocessing achieved that. The combination of the parsing and the structural extraction as well as the transformer summarization helps to provide the smooth experience of reading the code. Users will not have to read long lines of the Java files anymore. Instead they are able to manoeuvre through clean diagrams and immediately know what each section of the code does.

In general the summarization aspect forms a fundamental aspect of ClarifAI intelligence layer. It creates a bridge between the code and developer knowledge through generating readable and meaningful rules that enhance the diagrams. It also gives a background of the relationship of different segments of a code and the reason why some patterns occur. That is why we are using CodeT5, as compared to lightweight models such as Code2Vec because the encoder decoder transformer architecture provides CodeT5 with a much better conceptualization of the nature of the Java language as a whole, which makes it much more useful as a code comprehension engine.

4.4.6 Frontend Integration and UI Workflow

The frontend part of ClarifAI is basically the layer that makes everything usable because all the backend processing only becomes helpful when it is shown in a clean and understandable way. From the start we tried to keep the interface as simple as possible so students and developers could upload code explore the structure and read the summaries without needing any technical setup. The frontend is built using standard HTML CSS and JavaScript and it communicates with Flask through simple API calls. Whenever a user uploads a Java file or a full folder the UI sends it to the backend and then waits for the processed output. Once the backend finishes parsing the code and generating the structure it sends back a JSON response containing all classes methods fields and their relationships. The frontend takes this JSON and renders it into an interactive tree. This tree is one of the main parts of the UI because users can expand classes click on methods and jump between different parts of the project. Making the nodes interactive took some trial and error since the tree needs to update smoothly without lag especially when the project is large. We used event listeners on each node so that when a user selects a class or method the relevant summary and code snippet appear instantly on the right side.

Once the AST and CFG have been created on the backend they are also transmitted as formatted information which gets converted by the frontend into visual representations. The diagrams have been presented by using dynamic HTML elements rather than heavy libraries which makes the entire system lightweight. On clicking a node within the diagram, the frontend requests the pre generated

summary and renders it in the summary panel without making another call to the model. This causes the workflow to be expeditious and it does not create unnecessary delays. In essence the frontend is just a viewer as all the heavy work has been done previously in the backend. We have also implemented small visual elements of the user interface to enhance usability such as loading indicators as files are being processed or hover effects to indicate that an element is selected. Given that a number of students may be utilizing this tool to learn we made sure to ensure nothing was cluttered without any superfluous features just the bare minimum. This was aimed at ensuring that the diagrams and summaries feel related in such a way that the user of the code can appreciate the structure and the meaning of the code together. Generally the frontend is a mediator between the user and ClarifAI internal logic. It manages uploads and sends requests and receives structured information and displays it in a clean format. Although the technical work is done in the most by the backend, it is the frontend that makes the tool actually usable and easy to navigate. As long as the remaining system was not being confounded by the UI, there would not have been much difference in us dedicating additional efforts towards ensuring that the workflow was easy smooth and intuitive.

4.4.7 Error Handling and Validation Mechanisms

The quality of parsing and summarization is extremely sensitive to the correctness of the input Java code, so a reliable error handling system was required by ClarifAI. Any minor mistake in files uploaded by the users such as brackets being missing, or missing methods or other formatting errors can disrupt the whole processing chain. To control this we introduced layered validation whereby quality of code is verified by the system before it is sent to the parsing or summarizing phases. Small IDE such as an editor is also part of the frontend in which users can correct the errors directly. The editor keeps the highlight at the specific line when the backend finds an error so that the error can be easily debugged.

As we developed, javalang fails on even simple syntax errors and this may lead to the failure of the AST and CFG generation. All the parsing functions were therefore enclosed in safe blocks that are able to trap exception and provide meaningful messages rather than crash. When the parser identifies a file to be broken we halt further processing such as AST generation and comment generation since this part would either deliver inaccurate results or not work at all. Multi file uploads we also provide partial processing whereby even when a file has errors the rest are still processed and shown standardly. Another important part of validation involves protecting the diagram rendering and CodeT5 summarization workflow. Both components require clean well structured input otherwise the output becomes incomplete or confusing. If there is any structural issue it simply warns the user instead of drawing broken diagrams or generating strange summaries. Early checks for invalid file types non Java files or empty folders also help avoid unnecessary processing. Overall the goal was to create a tool that guides the user whenever something goes wrong and keeps the experience smooth by preventing errors from flowing into later stages of the pipeline.

4.5 Screenshots of Prototype / System

Below implementation design created through Figma and implemented in Flask.

4.5.1 Sign up

This screen displays the page to make a new user account in the system. To start using ClarifAI, the user provides simple data to start with.

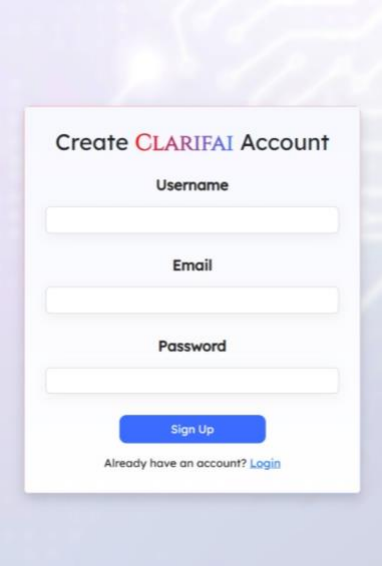

Figure 4.6 Signup

Figure 4.6 shows the signup interface for the website.

4.5.2 Login

This screen allows the existing users to enter their account. Once they log in they are able to explore any functionality of the system.

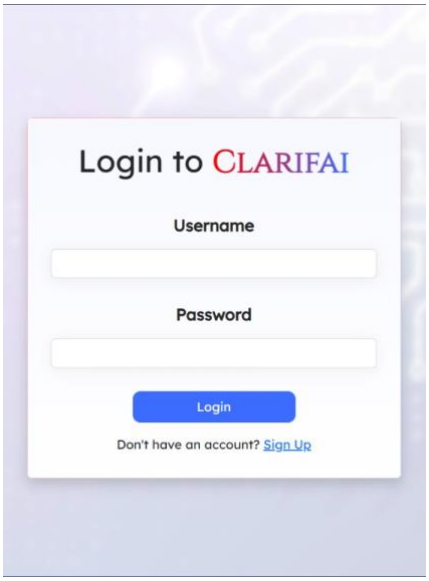

Figure 4.7 Login

Figure 4.7 shows the login iinterface for the website.

4.5.3 Landing Page

It is the page that the user reads after making a log in. It opens access to the key tools and directs the users to the uploading of code or its analysis.

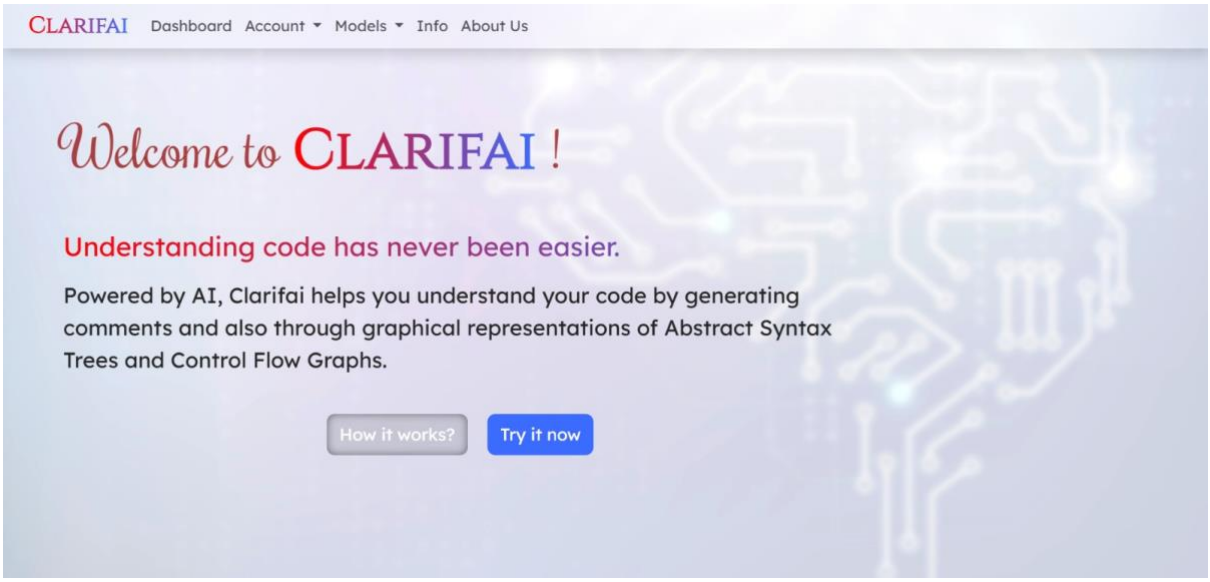

Figure 4.8 Landing Page

Figure 4.8 shows the web interface of the landing page

4.5.4 Code Submission

This screen gives the users ability to paste Java directly to the editor. The system takes the code and gets it ready to analyze it.

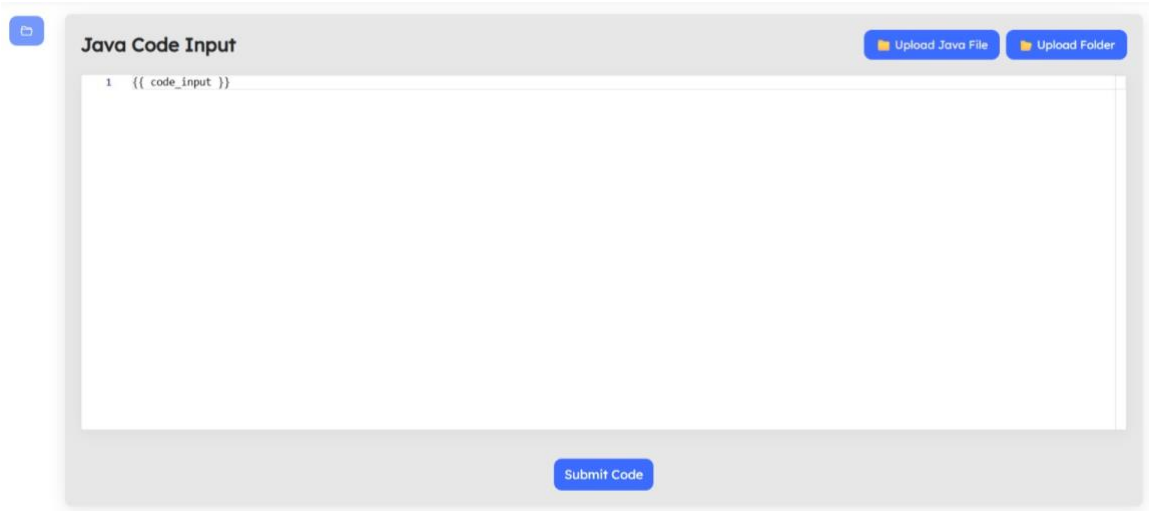

Figure 4.9 Code Submission

Figure 4.9 Shows the interface where user can upload the code manually.

4.5.5 Code History Preview

This screen shows all the files of codes that have been uploaded. Any past submission can be chosen by the user in order to review its diagrams and summaries once more.

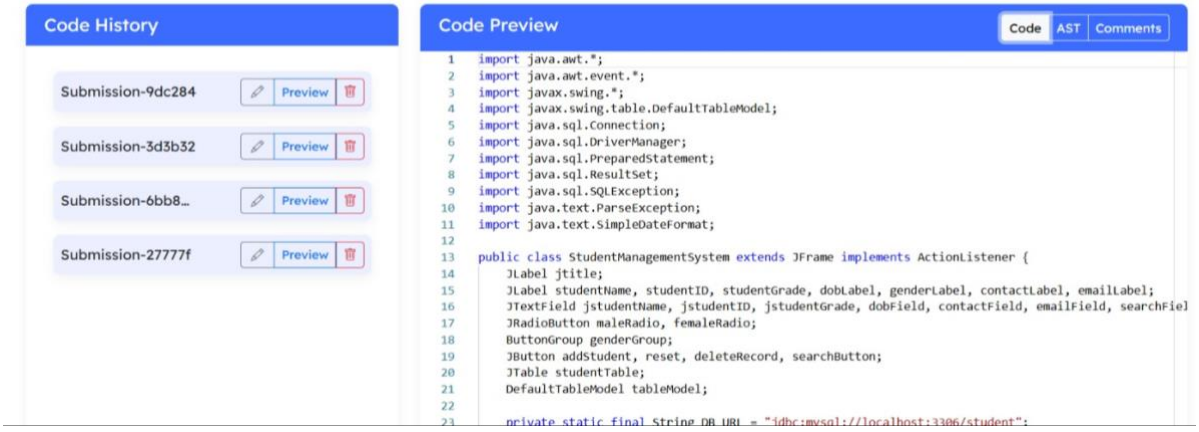

Figure 4.10 Code History

Figure 4.10 shows how the code history preview.

4.5.6 File Uploading

This screen will allow users to upload a single Java file, multiple files or even a whole folder. The system reads and prepares all the files.

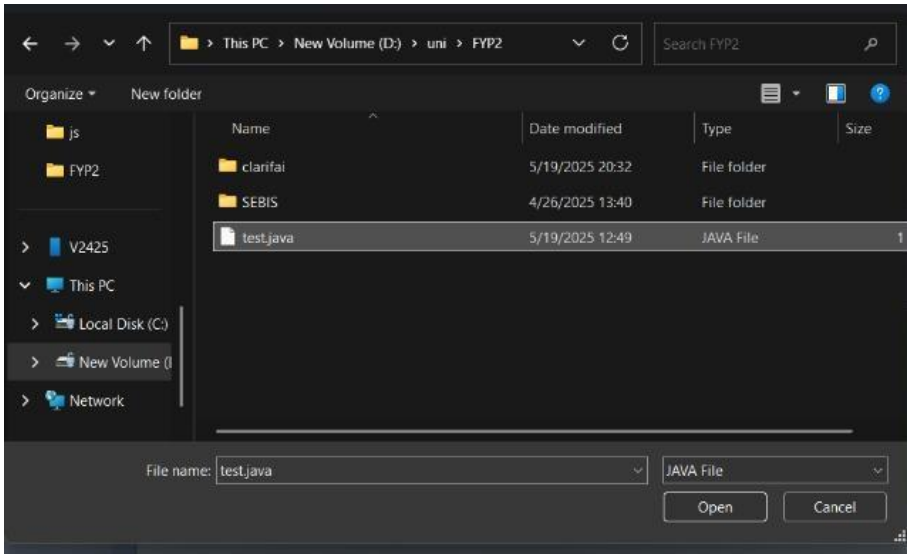

Figure 4.11 File Uploading

When the user clicks on file upload the preview in figure 4.11 opens for user to selects the file/folder to upload.

4.5.7 Comment Generation & AST Page

The AST diagram of the uploaded code is shown in this page. It also displays the overviews of all the components of the structure.

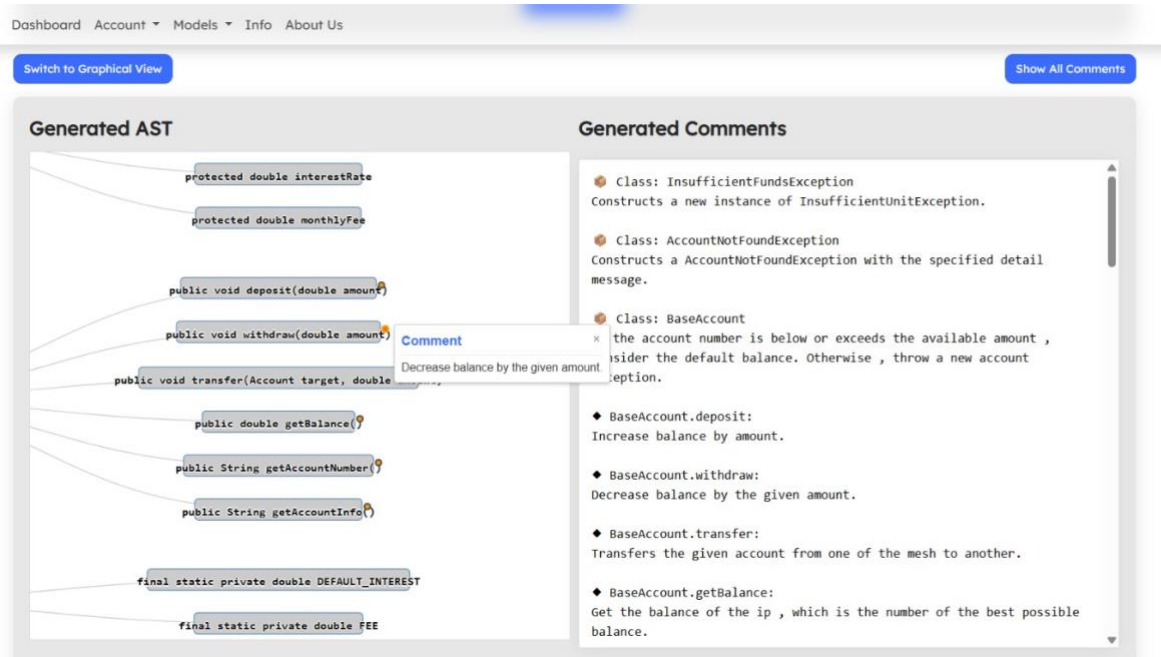

Figure 4.12 AST Graphical view and Generated Comments

Afther the successful parsing and comment generation the Preview of AST tree along with comments are shown in figure 4.12.

4.5.8 Class/Method-wise Comments

This screen displays the overview of a chosen course or procedure. It assists the users to know the functionality of each section of the code.

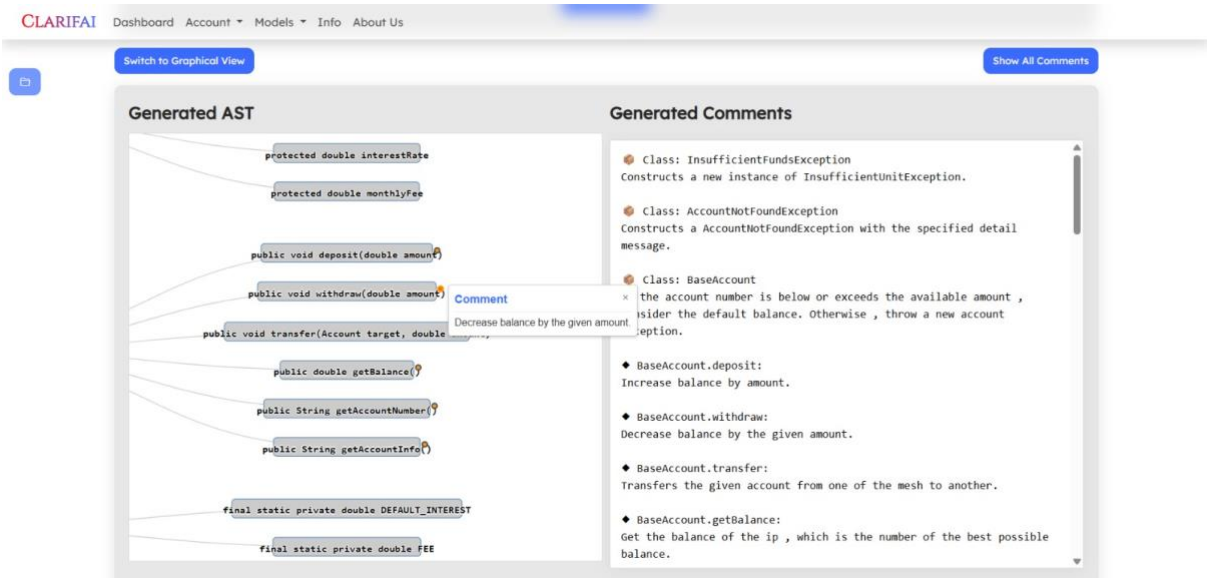

Figure 4.13 Class wise comment

Figure 4.13 Displays the class wise comments along nodes of tree.

4.5.9 CFG Display

Here is the CFG diagram of the chosen method. It assists users in navigating through the execution within the code.

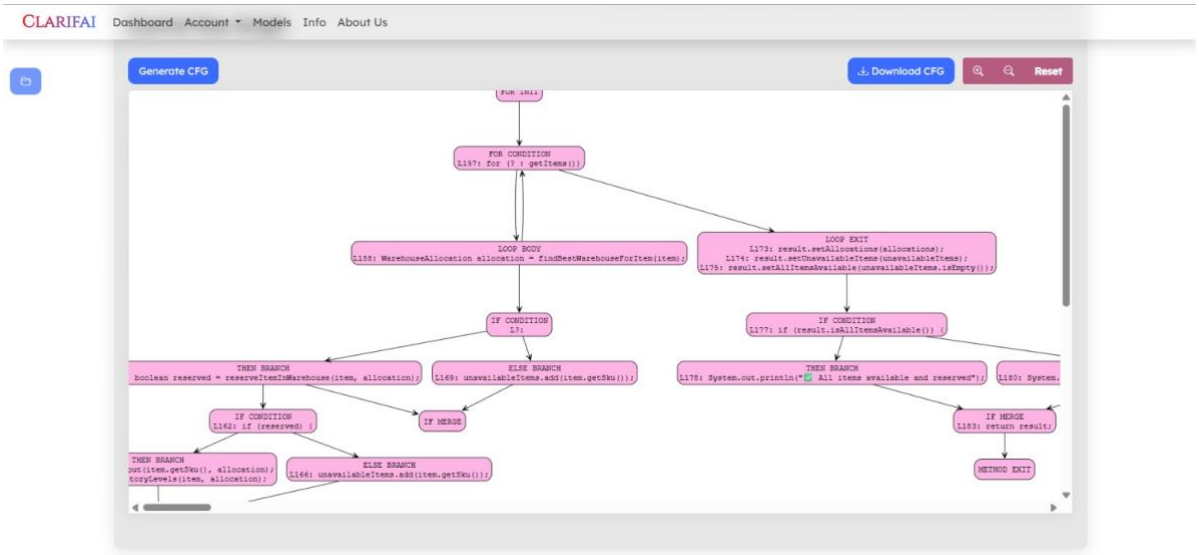

Figure 4.14 CFG Diagram

Figure 4.14 displays the Control Flow Diagram of the uploaded code.

4.6 Challenges During Implementation

During the development of ClarifAI we came across a bunch of practical issues that slowed us down here and there and pushed us to change our approach multiple times. One of the main challenges was getting the Java parsing to work properly. At first the parser would break on random files mainly because of inconsistent formatting and small syntax errors in user code. We had to add our own formatting and cleaning layer just to keep the parser stable. Another big difficulty was with the summarization model. We experimented with various models prior to CodeT5 yet the majority of them generated unfinished summaries or did not take into account the context of the code at all. The fine tuning process was not that simple even after selecting CodeT5 since we were faced with restrictions on the availability of GPU in Google Colab and we had to decrease the size of the dataset which influenced the speed and accuracy of the training.

The integration aspect also consumed more time than allowed. The AST generator and the model needed a considerable amount of trial and error before they worked together without blocking one another. We would experience problems such as slow response time model timeouts and different data format of modules. On the frontend implementation side we had some problems maintaining the tree view uncluttered and easy to read in situations where the codebase had too many classes or too many methods within other methods. And finally, the entire system was to be lightweight meaning that we had to eliminate some of the features which we originally planned to incorporate like the real time collaboration and support of multiple languages.

Chapter 5. Results & Analysis

The findings of the implemented and tested system are reporting in this chapter. It shows the results of the processing pipeline of the code such as AST structures CFG diagrams and produced summaries and evaluates the performance of the system on various cases. Experimental results model evaluation measures and data analyses are also presented in the chapter to establish the rates of the approach effectiveness and reliability. This chapter uses both visual results statistical analysis and performance feedback to give one a comprehensive picture of the strengths and limitations of the system as a whole.

5.1 Experimental Setup

In our experiments, we primarily aimed at the evaluation of the performance of the fine-tuned CodeT5 model in summarizing Java code. All was implemented in Google Colab as it provided us with simple access to GPU, and we were not to concern ourselves with the local equipment restrictions. Our model was trained on the Java part of the code-to-text dataset of CodeXGLUE. Our evaluation metrics were BLEU score, training loss, and validation loss since these are widely used in the code summarization work and provide a rather reasonable vision of the quality with which the model is learning. The batch size was the size that was allowed by Colab GPU and fine-tuning was performed on a smaller number of samples primarily because of the limitations of the GPU memory.

Table 5.1 BLUE Score

Model	**BLEU Score**
Baseline	16.65
Fine tuned Model	13.5552

Blue score for the implemented base model and its fine tuning is showed in Table 5.1.

...

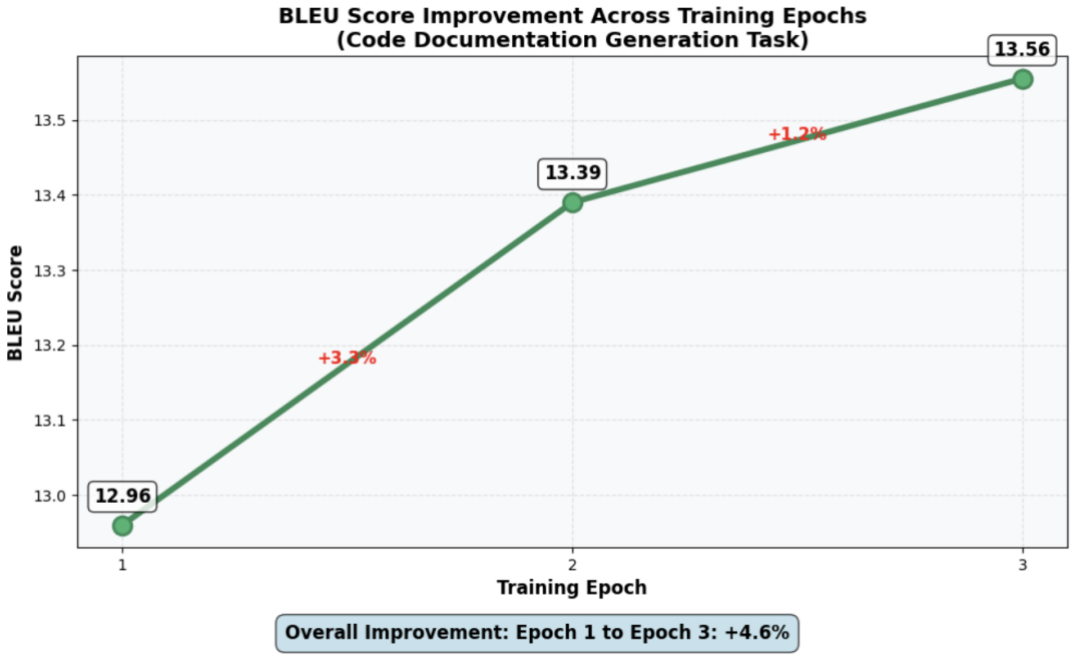

Figure 5.1 BLUE improvement

Figure 5.1 shows the value which demonstrates that the BLEU score was continuously rising as the model was being trained during three epochs. The largest improvement of 12.96 to 13.39 indicates the largest increase in learning and the last improvement of 13.56 indicates that the model stabilized as it fine-tuned. On the whole, the trend proves that the training process was productive and that the model was improving with every step of the data.

5.1.1 Dataset Characteristics and Statistical Distribution

To have a better understanding of what type of data our model was being trained on we also investigated more closely the structure of the dataset itself. We verified the average length of the Java code snippets, the average length of the docstrings and the number of tokens in them. In essence we created four distributions of length of code, length of docstring and the number of their tokens. These graphs were used to make us realize the extent to which there is variation in the dataset and what the model will generally experience during training. This came in handy as it helped in making choices such as what degree of preprocessing we were required to make, how long sequences would be safe to work with, and what type of complexity the model would be working with. Based on the findings, the majority of samples lie in a reasonable range, which implies that they will not be problematic when it comes to their token limits in the model and will not create significant problems when fine-tuning the latter.

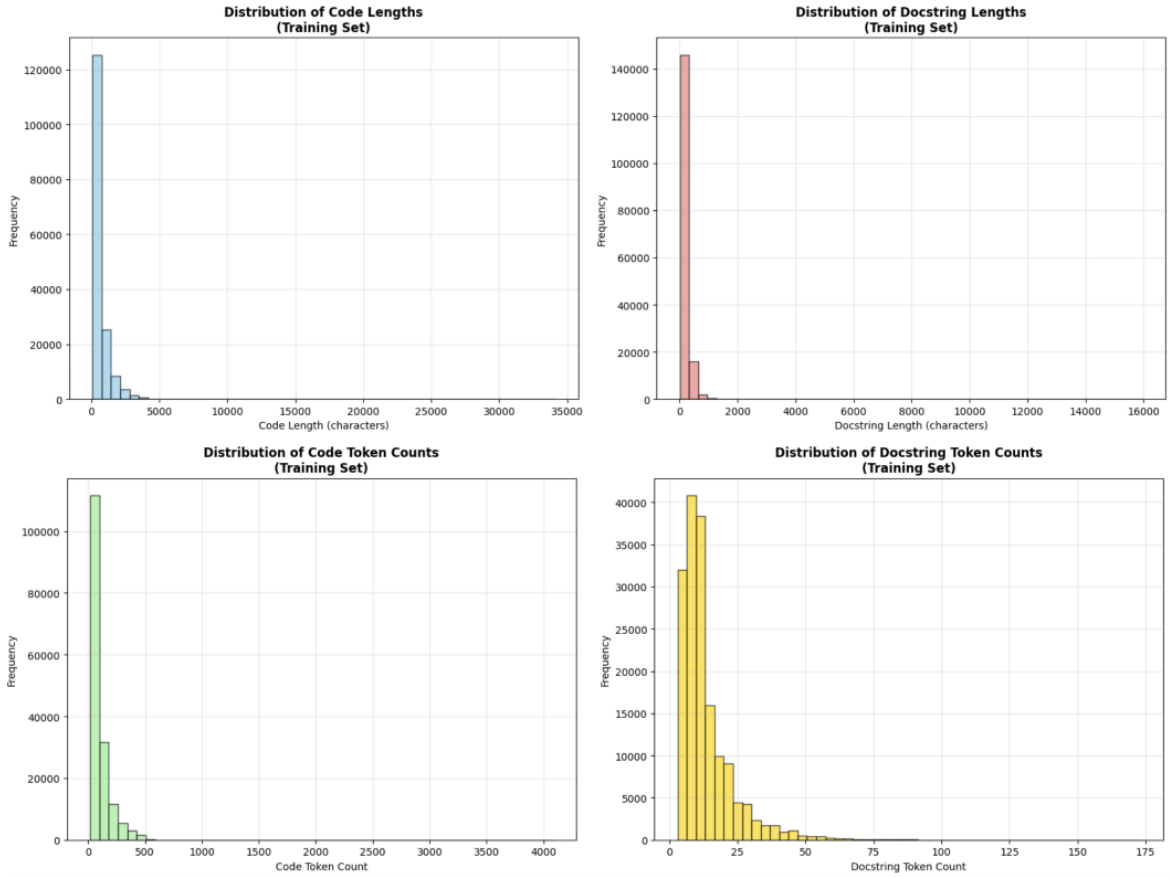

Figure 5.2 Dataset Characteristics and Statistical Distribution

Figure 5.2 displays four distributions of length of code and the number of their tokens that are used to recognize the extent to which there is variation in the dataset.

5.1.2 Code & Docstring Content Analysis

In order to have a feel of the linguistic patterns the model would face in the course of training, we examined the most frequent keywords in Java code, as well as the most frequent keywords in the code written by the developer. This assisted in providing a better idea of the similarity between the vocabulary of the source code and the natural language descriptions that the model should produce. The syntaxes indicate that Java keywords, including if, return, new, and public, have dominated the source code whereas recurrent semantic words of param, return, this, value, and method are found in all the docstrings. This comparison enabled us to confirm that the dataset has a healthy proportion of structural code tokens and descriptive natural-language words, which is relevant in making sure that the fine-tuned model acquires the relationships between the code and text that are meaningful and not a mere syntax memorization.

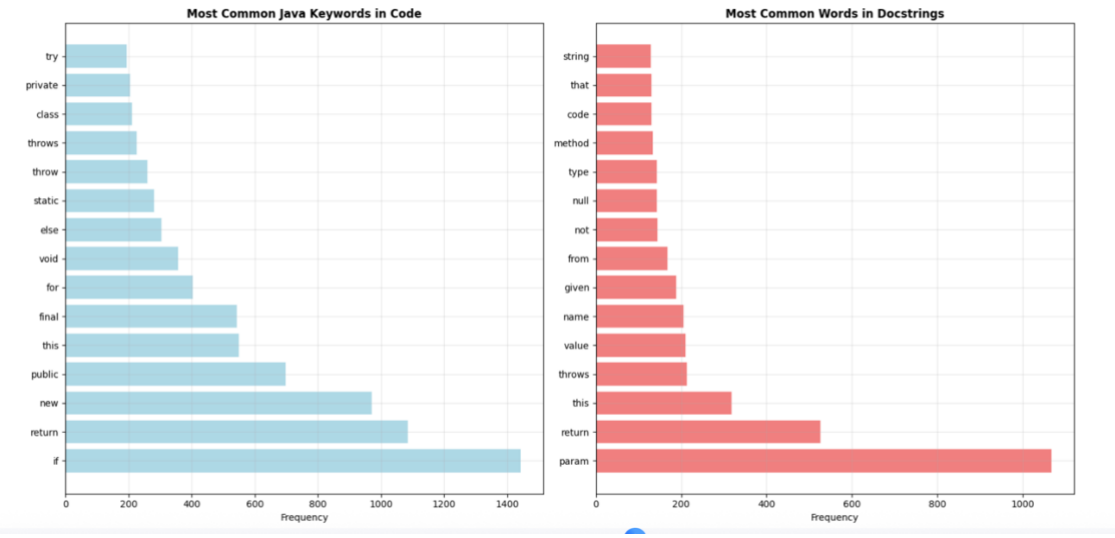

Figure 5.3 Dataset Characteristics and Statistical Distribution

Figure 5.3 displays the use of most commonly used keywords in the dataset for the fine tuning for the model

5.2 Comparitive Analysis with Existing Work

Understanding how ClarifAI performs in comparison with existing tools and research based systems is essential because it illustrates where the project stands and the type of contribution it makes in the broader field of source code comprehension. The problem of understanding unfamiliar code is widely recognized in both academic and industrial environments and many existing tools attempt to provide partial solutions. However most of them address only one aspect of the challenge such as navigation syntax highlighting summarization or structural visualization. They do not provide a complete workflow that integrates code parsing automatic summarization and interactive visualization within a single platform. Most existing editors such as IntelliJ [3] and VS Code [4] focus mainly on code navigation highlighting and basic static inspection. These editors provide essential features for software development and maintenance but their ability to assist in deep comprehension is limited because they do not generate contextual explanations or interactive visualizations of internal code logic. Developers still need to trace relationships manually across files and understand class method and object interactions by reading code line by line which slows down the process and increases cognitive load.

Code summarization The research models of code summarization include CodeT5 [2] CodeTrans T5 implemented on the T5 architecture [15] and other transformer based systems have shown excellent results in the generation of textual descriptions of code segments. Such models can input a complete program or even an individual method and output a natural language summary to describe the overall purpose or behaviour of the code. Although they are effective in derivation of readable explanations they are constrained structurally. They tend to perceive the overall program as a block and give no distinction between classes methods or code blocks. They therefore produce generic remarks that fail to give details of the behavior of particular components. This constraint plays a very important part in real world applications where it is important to know how the methods and classes relate to each other. ClarifAI solves this problem by splitting the uploaded code into logical units (based on the class and method boundaries) with such techniques as the count of brackets naming conventions and parsing hierarchies. These segments are then passed to the model individually to produce elaborate class based and method based remarks. By doing this, the system can generate finer and more significant explanation that captures the actual structure of the code as opposed to giving one comprehensive explanation.

The other type of work is work on AST based analysis including ASTNN [10] and early tree structured models including Code2Vec [8] and Code2Seq [9]. The systems are aimed at using the

syntactic structure of code to enhance machine learning processes such as the prediction of a code classification method and code defects. Representing code as a tree gives patterns that are not apparent in plain text models and can be very predictive accurate in research. These systems are however not designed to be explored in an interactive way by the users. Their generated ASTs are then processed internally to be computed and make predictions instead of being represented in a readable form. Users are not able to click nodes or graphically explore the structure in order to get an idea of class relationships, or method callings. ClarifAI builds on such a strategy by providing the AST in an interactive and visually understandable form. Individual users are able to navigate all nodes and see related natural language descriptions of classes and methods that bridges the gap between machine learning understandings and human cognitions. Such a design will enable the AST not only to be a research tool but a useful educational and professional resource.

Diagrams can be generated by many standalone and online diagram generating tools including CodeSee [21] PlantUML [12] UMLGraph and other diagram visualizers that can generate diagrams based on Java source code. These tools aim at visual representation of the output of a program and in many cases generate class diagrams flowcharts or sequence diagrams. Although they are practical when it comes to offering a high level of code organization they lack semantic explanation. The users will be required to decipher the diagrams and derive the behaviour of methods and classes manually. Also such diagrams tend to be plain images that cannot enable one to interact and explore certain nodes and learn more about them. ClarifAI fills this gap by connecting the generated AST and CFGs diagrams to the summaries generated by the fine tuned model. Any node can be clicked on by the user to see the automatically generated description of that code element. This interaction provides an interactive mode of exploration whereby the user is able to explore the structure of the program and at the same time learns the behavior and interrelationship of each component in the program. Such an aspect allows making ClarifAI not only a visualization but a comprehension tool enhancing learning and code review effectiveness.

The necessity of syntactic and semantic information combination, which will allow full understanding, is also highlighted in academic literature. The use of textual context with structural patterns, like ComFormer [13], PLBART [14] and hybrid transformer pipelines, shows that the combination of textual context and structural patterns helps achieve better results in code summarization and code understanding tasks. Although effective these models are usually measured numerically in terms of BLEU scores or prediction task accuracy. Their products are seldom structured in such a way that they can be used directly by developers or learners who need to navigate through new codebases. Similar principles are used by ClarifAI which takes syntactic structure as parsing and AST generation and semantics information as the CodeT5 model and displays the output in an interactive and user friendly interface. By so doing the theoretical gains made in research are translated into practical gains to the actual users.

Conventional documentation generators like Javadoc use annotations to the code that are written by humans. Although these tools may be useful as reference documentation they are not automatic content generating and require the author of the code to write explanatory comments. Projects whose comments are incomplete or old generate an inadequate or distorted documentation. ClarifAI removes this dependence by generating automatic justifications of every class and method without having to be annotated manually. This aspect renders the system robust and applicable even in the poorly documented or the legacy codebases.

Besides, ClarifAI stands out of the past work since it has many capabilities in one platform. The available tools usually offer a single feature like code summarization or AST generation or visualization of the static. The integration of several tools is needed by the users to gain a holistic outlook that complicates the matter and hinders productivity. ClarifAI combines the parsing summarization AST and CFG visualization interactive navigation with an easy to use interface design. This integration enables users to import a project and get structural diagrams and elaborate explanations to the individual components within one environment. The system promotes the processes of both academic and professional developers where unfamiliar programs of Java can be understood and reviewed at a quick pace.

In short, existing editors concentrate on navigation and syntax highlighting [3][4], research based models concentrate on summarization or classifying code and structural diagrams are available on the basis of static visualization [12][21]. ClarifAI incorporates all these features into one convenient

platform that offers structural parsing semantic summation and interactive discovery. This integration enables the users to comprehend the high level architecture and the specifics of the classes and methods. Using these characteristics ClarifAI is a solution that actually fills the gaps of previous tools and models and gives a real user beneficial results by being able to understand and engage with Java code in both an effective and efficient way.
The comparison is divided into two sections given the differences between goals of each model and tool since one is the actual summarization performance which primarily uses BLEU scores and the other is the ability to produce tree or structure. This will assist in illustrating specifically where ClarifAI is more skilled and what deficiencies there remain in comparison with the larger research constructs and the well-established tooling. The results are arranged well in the tables below.

Table 5.2 Model-Level Comparison (Code Summarization)

Model / Tool	**Dataset Used in Paper**	**Reported BLEU Score**	**Summary Quality**	**Strengths**	**Weaknesses**
CodeT5 (Original)	CodeSearchNet	~20.0 – 22	High	Excellent for code-to-text tasks; strong encoder decoder	Requires large datasets; heavy GPU resources
Code2Vec	Java-small / Java-med	~2.5 – 3.2	Very low	Good for method name prediction	Not suitable for full sentence summaries
Code2Seq	Java-small / Java-med	~18–20	Medium-high	Uses AST paths better than Code2Vec	Struggles with long summaries
ComFormer	CodeSearchNet	~23–25	Very high	Dual attention over AST and tokens	Requires long training and strong GPUs
ClarifAI	CodeXGLUE (Java, 5k samples)	~13.55	Medium	Learns fast even on small dataset. integrated AST + summaries	Limited training set, not full size T5 training

Table 5.3 Structural , Tree-Generation Comparison

Tool / System	Type of Structure	Strengths	Weaknesses	Comparison with ClarifAI Tree System
PlantUML	UML Class Diagrams	Good for static diagrams	Requires manual annotations	ClarifAI generates structure automatically from AST
UMLGraph	UML from Java	Accurate diagrams	Fails on incomplete code	ClarifAI handles partial code better
JavaParser	AST Trees	Very precise AST	No visualization	ClarifAI converts AST into user-visible form
CodeSee	Dependency Maps	Great for project-level mapping	Heavy setup; no code summaries	ClarifAI is lightweight and provides summaries + AST
ClarifAI	AST Tree + Custom Formatting	Interactive, automatic, explanation-ready	Not a full CFG editor	More beginner-friendly than all above

In the comparison tables 5.2 and 5.3 , provides a rather good picture of how our system would actually fit in comparison with other methods. We have a much lower BLEU score than the actually large research models but the improvement we received on fine tuning indicates that the model still quickly picks up things despite being trained on much less data. Structurally ClarifAI is a good tool since our tree views and AST views operate directly on the uploaded code and do not require additional IDE extensions or linking entire repositories as is the case with some other tools. It makes it light and even finds time to display the structure in a clean manner. So in general ClarifAI lies in the middle ground in that it will generate fair summaries and correct structural diagrams, but at the same time it remains easy to use without being complicated enough that it would require students or developers to break a complex set up. The combination of that is what makes it applicable to real-life scenarios.

5.3 Discussion on Findings

Having conducted all the experiments and read the data, a few things became clear on how we perform our system in the real world use. Fine tuning also resulted in observable improvement in the model despite it being trained on a very tiny fraction of the available data. The growth in BLEU rating was not very high relative to the huge research papers yet the growth represented the fact that the model is acquiring the patterns in Java code in a sensible manner. This implies that the summarization section did not become as bad as we initially thought despite the lack of resources. The other aspect that was notable was the performance of the system in terms of structure. The AST generation and the tree representation were pretty reliable and failed in the majority of instances when an irregular Java syntax was entered they created an appropriate hierarchy. This component was smoother than some other known available tools that we used previously primarily due to the fact that it is not based on heavy IDE extensions or huge repository configurations. It just accepts any code given by the user and converts it into something that can be easily investigated. This made the system more light and more convenient to the user that desires to learn code quickly without having to install a full environment.

We also observed that the combination of structure and summarization gives a more complete picture of the code compared to using only one part. The summaries help explain what the AST nodes represent and the AST helps the user locate the specific code section that the summary describes. These two components support each other which was one of the main ideas behind our project.

Overall the findings show that ClarifAI is not trying to outperform large research models or replace advanced IDE based tools. Instead it sits in a middle space that is simple enough to use quickly while still strong enough to give meaningful insights about unfamiliar Java code. The results support the

idea that a lightweight mix of structural extraction and transformer based summarization can still be effective even with limited computing power and training data.

Chapter 6. Conclusions

The chapter gives the final reflection of the project and the highlight of the achievements that were made in the development of the system. It re-examines the primary goals and analyzes their degree of attainment as well as shows the overall work contributions. Another subject of the chapter is the wider applicability of the system and how it may be used by learners and developers. Lastly it gives suggestions to the future implementation and directions that can enhance the function of the system to greater heights than it is presently.

6.1 Summary

We wanted to develop a system in this project that can simplify the Java code by summarizing it intelligently and structure analysis. In the initial stages of our work, we concentrated on researching the already available tools and the research papers in the field of software engineering and machine learning to be able to construct the solution that would not look unrealistic to the real developers. Based on these works we created ClarifAI that unites code parsing AST generation CFG construction with transformer based summarization into a single workflow. The project was aimed at simplifying reading of new source code by students and developers and providing students and developers with an effective visual and textual representation of the functioning of every aspect of the program.

During the development phases ClarifAI proved that a lightweight environment could provide a sound support to users who deal with the Java projects. The parsing phase was found to be both reliable and the reassembled AST had the capacity to point out significant structural features like classes methods and expressions in a clean tree presentation. This graphical representation of the code did not require the user to read extensive and complicated source code in order to navigate the design. The CFG generation feature provided an extra degree of insight in presenting the flow of execution inside each of the functions. In a structured graph, the users were able to view decision points loops and returns which facilitated easier interpretation of the control flow. These diagrams were all that made a complete structure pattern of the uploaded Java code.The semantic layer of ClarifAI was given through the summarizing component. The CodeT5 model that we refined using a Java specific dataset generated readable and informative summaries of both the classes and methods. The fact that the BLEU scores increased after fine tuning was an indication that the model was picking up useful patterns despite the small size of the dataset. This confirmed the ability of transformer based models to adapt within a short time, and they are apt in code focused tasks. The connection between the generated summaries and the structural nodes also made the system stronger as it allowed its users to click any aspect of the diagram and see instantaneously its explanation. This made ClarifAI more interactive and more helpful to users who desire to obtain fast insights without having to conduct a manual analysis.One of the biggest strengths of the system is the fact that all components are well integrated. The backend takes care of the CFG generation and summarization in an efficient manner and the frontend presents diagrams and summaries in straightforward interactive format. Other features like the expansion node through zoom and downloading of high quality diagrams were also added to enhance the user experience. ClarifAI does not have a complicated installation process and heavy development environments meaning that students and computer programmers can easily use it on any web browser. This ease of use renders it both scholarly and educational applicable and to software engineering novices that require more rapid onboarding.

In the testing phase we noted that ClarifAI always assisted users to navigate and understand Java files that would have taken long time to read. It was of particular benefit to classes with a large number of students since it is difficult to visualize the structure. Other areas that can be enhanced in the future that were noticed in the project include the inclusion of more programming languages to enhance CFG accuracy in un-conventional control pattern and more training data to enhance

summarization. These are the observations, which demonstrate that the main concept of ClarifAI is powerful and can still be developed as required. The project in general achieved the primary goals. ClarifAI provides a transparent and reachable method of examining Java code by visual representations and natural language definitions. It decreases the amount of manual work enhances the speed of learning and offers a favorable atmosphere in learning the new programs. The AST CFG + AI based summarization resulted in a useful and informative tool. The project shows how the current code intelligence methods can be implemented into a user friendly system that makes the developers easier and more confident in understanding the software. In this project we set out to build a system that makes Java code easier to understand by combining structural extraction with automatic code summarization by exploring the related tool and research papers. Throughout the development and testing stages ClarifAI showed that even a lightweight setup can still give meaningful help to users who deal with unfamiliar source code. The parsing and AST generation parts worked in a steady and reliable way and the structural diagrams made it easier to see how different parts of the code are connected.
At the summarization aspect the CodeT5 model was more successful following fine tuning and the increase in BLEU score demonstrated that model does learn helpful patterns even with a small dataset. The combination of the two features makes the results more complete since the structure and the summary complement each other. The net system proved to be convenient to the students and developers who desire swift learning without installing extensive tools and environments.In general the project achieves its primary objective which was to offer an easy and very accessible means of reading Java code via visual representation and natural language descriptions.

6.2 Recommendations for Future Work

There are several directions in which ClarifAI can be extended in the future. One important improvement would be to support more programming languages so the system is not limited to Java. Another possible upgrade is to add deeper analysis features such as control flow graphs or data flow tracking so users can understand the logic inside the methods in more detail.
The summarization model can also be improved by training on a larger dataset or by experimenting with newer transformer models that might produce clearer or more precise explanations. A more advanced user interface can also help especially if we add searchable diagrams or real time interactions with the AST.
Finally a cloud based or distributed version of the system could allow heavier models to run faster or support larger projects without slowing down. These improvements can make ClarifAI more powerful and bring it closer to being a complete code understanding assistant.

Chapter 7. References

[1] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey,," *Conference of the Centre for Advanced Studies on Collaborative Research,* 2002.

[2] Y. Wang et al, "CodeT5: Identifier-aware pre-trained encoder–decoder models for code," 2021.

[3] Jet Brains, "Intellij Idea," [Online]. Available: https://www.jetbrains.com/idea/.

[4] S. Iyer, "Visual Studio Code," 18 November 2025. [Online]. Available: https://code.visualstudio.com/. [Accessed 2025].

[5] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, in *Compilers: Principles, Techniques, and Tools*, Boston, MA, USA: Addison-Wesley, 2006.

[6] E. Bisong, "Google colaboratory," Building machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners, Berkeley, CA: Apress, 2019.

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *Proc. NAACL-HLT,* p. pp. 4171–4186, 2019.

[8] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang,* vol. 3, pp. 1-29, 2019.

[9] U. Alon, R. Zilberstein, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *ICLR,* 2019.

[10] J. Zhang, X. Wang, H. Zhang, J. Chen, and K. Wu, "Aston: AST-based neural network for source code representation," IEEE Trans. Software Engineering, 2020.

[11] SonarQube, "Continuous inspection platform," 2024. [Online]. Available: https://www.sonarqube.org/. [Accessed 23 Jun 2025].

[12] PlantUML, "Open-source UML diagram generator," [Online]. Available: https://plantuml.com/. [Accessed 23 Jun 2025].

[13] W. Y. e. al., "ComFormer: Code summarization with dual attention," arXiv preprint arXiv:2108, 2021.

[14] W. Ahmad et al., "Unified pre-training for program understanding and generation via PLBART," ACL, 2021.

[15] C. Raffel et al. , "Exploring the limits of transfer learning with a unified text-to-text transformer,," JMLR, 2020.

[16] Y. Wang. e. al., "CodeT5+: Open code large language models for code understanding and generation," arXiv:2305.07922, 2023.

[17] H. Liu et al., "AST-T5: AST-aware pretrained transformer for programming languages," arXiv:2310.12345, 2023.

[18] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the analysis of source code," *ICPC,* p. 37–146, 2013.

[19] Github, "Tree-sitter: A parser generator tool and incremental parsing library," 2018.

[20] R. Pawlak et al., "Spoon: A library for analyzing and transforming Java code," *Software: Practice and Experience,* vol. 46, no. 9, p. 1155–1179, 2016.

[21] CodeSee, "Interactive codebase maps," 2024. [Online]. Available: https://www.codesee.io/. [Accessed 2 Nov 2025].

[22] E. Foundation, "AST View," Eclipse JDT Documentation, 2024.

[23] A. Vaswani et al, "Attention is all you need," Advances in Neural Information Processing Systems, 2017..

[24] M. Husain, A. Choudhury, S. Wu and J. Hamilton, “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search,” *arXiv preprint arXiv:1909.09436*, 2019.

[25] D. Lu *et al.*, “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation,” *arXiv preprint arXiv:2102.04664*, 2021.

Appendix A

*Include here the 1st page of Turnitin Report (**MANDATORY**)*

Every supervisor has his/her own Turnitin account. If not, then supervisors are requested to get the account from Library as soon as possible.

Appendix B

Include here the 1st page of AI Report generated through Turnitin ***(MANDATORY)***

Every supervisor has his/her own Turnitin account. If not, then supervisors are requested to get the account from Library as soon as possible.