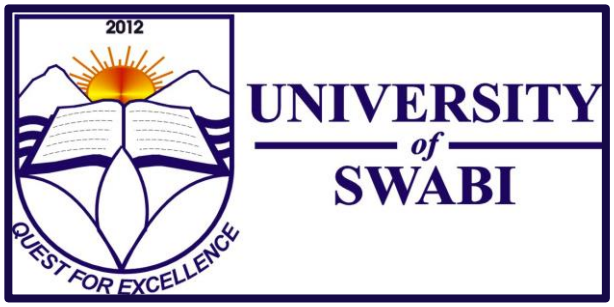
SMART CODER: AI POWERED PROGRAMMING ASSISTANT

BY

SHADAB ALI
SHAH FAISAL KHAN
HASEEB AHMAD

A Thesis submitted to The University of Swabi, Khyber Pakhtunkhwa, in partial fulfillment of the requirements for the degree of

BACHELOR OF STUDIES IN COMPUTER SCIENCE



DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF SWABI,
KHYBER PAKHTUNKHWA, PAKISTAN,
SESSION 2021-2025

TABLE OF CONTENTS

LIST OF TABLES

TABLE OF FIGURES

ACKNOWLEDGEMENT

In the name of Allah, the Most Gracious and the Most Merciful, we start by acknowledging our sincere appreciation to the Almighty for having provided us with wisdom, endurance, and the courage to succeed in this final year project and thesis. This journey has been enriching both intellectually and personally. The successful completion of this project is a major milestone in our university life and would not have been achieved without the help, support, and directions of numerous people and organizations.

We are deeply grateful to our project supervisor, Dr. Anwar Hussain, for his constant support, professional advice, and helpful comments during the evolution of this project. His understanding of artificial intelligence and computer programming laid a solid basis for our work, and his ongoing encouragement kept us engaged and motivated throughout every step of the project. We are particularly thankful for the time and effort he spent reading over our work and assisting us in honing our ideas into a working and effective system.

We also extend our sincere gratitude to the Department of Computer Science, University of Swabi, for giving us a learning-friendly environment, a sufficiently equipped laboratory, and an encouraging faculty that promoted creativity and hands-on investigation. All the skills and knowledge that we gained throughout our study here were very crucial in bringing this project to life.

Finally, but far from least, we would like to thank our friends and families. Their boundless prayers, patience, and moral encouragement were the perpetual source of strength, especially during times of adversity. This success is theirs as much as ours, and we are dedicating this work to them. We also appreciate our fellow students and colleagues, whose debates, ideas, and friendship made this academic experience more rewarding and enjoyable.

SMART CODER: AI POWERED PROGRAMMING ASSISTANT

ABSTRACT

MR. SHADAB ALI, MR. SHAH FAISAL KHAN AND MR. HASEEB AHMAD AND DR. ANWAR HUSSAIN
DEPARTMENT OF COMPUTER SCIENCE,
THE UNIVERSITY OF SWABI, ANBAR SWABI, KHYBER PAKHTUNKHWA, PAKISTAN
JULY, 2025

With the fast-paced environment in software development, having intelligent, integrated tools is now essential. This thesis reports on the design and implementation of SMART CODER, an artificial intelligence-based programming assistant that integrates generative AI, runtime code execution, and a friendly interface to simplify the development process. The aide uses the Google Gemini API for sophisticated code generation, explanation, and debugging functions, so users can map natural language prompts into effective source code. In addition, use of the Judge0 API provides for secure, cross-language code execution within the application itself, providing users with instantaneous feedback and confirmation of effort. This integrated environment is provided by a web app designed to be responsive on the Streamlit platform, with an interactive chat UI that includes file uploads, choice of language, and immediate interaction.

The system aims to solve standard difficulties developers and learners encounter, including abrupt learning curves, broken toolchains, and mental effort debugging intricate code. By integrating various functionality—coding, debugging, running, and explanation—within a single seamless platform, SMART CODER reduces context switching and boosts productivity. SMART CODER supports a vast array of programming languages such as Python, JavaScript, C++, and Java and provides line-by-line, in-depth explanations of both AI code and user-provided code. The software is particularly useful for students, beginner developers, and even professionals aiming to accelerate prototyping, resolve algorithmic issues, or grasp unfamiliar codebases. The natural language interface of the assistant also reduces the entry barrier for novices and self-learners and makes high-quality coding support accessible to them.

The project proves the viability and potential of coupling generative AI with real-time execution environments to develop a solid educational and professional coding tool. Testing and deployment strategies prove the stability, responsiveness, and security of the system, while non-functional requirements ensure compatibility, usability, and data privacy. The results indicate that with such an instrument, the development time can be significantly reduced, quality of code enhanced, and users continuously learn. Ultimately, SMART CODER realizes the potential of AI in transforming the process of code composition, reading, and consumption and becomes a valuable assistant for both academic and industrial settings.

CHAPTER-I

INTRODUCTION

1.1 Project Background and Overview

The software development world is in the midst of a paradigm revolution spearheaded by the explosive growth of Artificial Intelligence (AI) and, more concretely, Large Language Models (LLMs). These new models have shown awe-inspiring capability at comprehending, producing, and reasoning with human language, and this capability directly transfers to the formal realm of programming languages. Code assistants powered by AI have therefore emerged as revolutionary technologies, evolving from mere syntax coloring and autocompletion to provide intelligent, context-aware assistance to developers.

The project provides the creation of a revolutionary AI Coding Assistant, a web-based application that is designed to be an all-encompassing assistant for a wide range of coding tasks. The assistant combines several cutting-edge technologies to build an integrated interactive platform to code, debug, and learn.

The system, in its core, utilizes the Google Gemini API, a strong generative AI model, for natural language processing and generation, explanation, and debugging that are code-related.

One of the distinguishing aspects of this project is its integration with a secure, multi-language code execution platform, the Judge0 Rapid API. Not only can users receive AI-generated code, but they can also run it in real-time in the application, with instant feedback and a real-world validation process. The whole solution is deployed as an accessible, user-friendly web application using Streamlit, a Python framework selected because it can quickly develop interactive data and AI applications. This technology stack—generative AI, remote code execution, and web-optimized interface—is the building block of an incredibly versatile tool destined to accelerate coding efficiency and understanding.

1.2 Problem Description

Despite the existence of development tools, developers of every skill level struggle with chronic problems that hamper learning and productivity. Such problems make up the problem space that this project will address.

1.2.1 Steep Learning Curve for Beginners

For learners and future programmers, the process of learning to code is generally beset with frustration. Navigating intricate syntax, comprehending algorithmic thinking,

and following elusive error messages can be daunting and discouraging. Learning materials, while abundant, tend to be deficient in interactive, personalized feedback to close the knowledge-to-practice gap.

1.2.2 Productivity Bottlenecks for Developers

Senior developers spend most of their time on dull and repetitive tasks, like boilerplate code, environment configuration, or converting logic between languages. Additionally, debugging difficult bugs, particularly in esoteric or legacy codebases, remains the most time-consuming task in software development.

1.2.3 Fragmented Tooling Environment

There are numerous tools to control the development lifecycle in an ad hoc fashion (e.g., linters, debuggers, formatters). Each one of them does so independently, however. A programmer will use one tool to compose code, another to run it, a third to debug, and a collection of web sites to learn about errors or concepts. This fragmentation results in a disconnected process with regular context-switching that breaks focus and makes it less efficient.

1.2.4 Code Comprehension and Maintenance

Understanding code written by others, or even one's own code from the past, can be a significant cognitive load. Without adequate documentation or immediate means of clarification, developers can struggle to make modifications or extensions, increasing the risk of introducing new bugs.

The AI Coding Assistant is conceived as a direct response to these problems. It seeks to provide a centralized, intelligent, and interactive platform that consolidates the functions of code generation, execution, debugging, and explanation, thereby creating a more fluid and supportive development experience.

1.3 Project Objectives

1.3.1 To design and implement an AI-powered code generation engine

This involves leveraging the Gemini API to translate natural language descriptions into syntactically correct and contextually appropriate code across multiple programming languages (including Python, JavaScript, Java, C++, and more).

1.3.2 To integrate a secure, real-time code execution environment

The project will incorporate the Judge0 Rapid API to enable users to instantly compile and run generated or manually entered code, receiving standard output, errors, and performance metrics directly within the application's interface.

1.3.3 To develop AI-driven code analysis and support features

This project has two primary functionalities: code explanation and code debugging. The code explanation aspect is geared toward providing concise, natural language explanations of how a given piece of code works. It analyzes the logic, organization, and purpose of the code in an attempt to educate users about its functionality and behavior.The code debugging feature assists users in identifying and correcting errors in their code. It does this by parsing error messages and examining erroneous code snippets, and then providing suggested solutions for correcting the errors and improving code performance. Together, all these features constitute an end-to-end learning and debugging environment for programming activity.

1.3.4 To develop an interactive and user-friendly interface with file processing features

The assistant will be provided as a web-based application developed using Streamlit and will have an elegant chat-based UI. An integral part of this aim is to incorporate functionality that enables users to upload source code files so that the assistant can analyze, explain, or debug existing codebases.

1.4 Project Scope

The extent of this project is set by the functionalities to be incorporated and technologies to be employed.

1.4.1 In-Scope:

I. **Application Framework:** Creation of a web app using the Streamlit platform.

II. **AI Integration:** Use of the Google Gemini API for all generative AI tasks, such as code writing, explanation, and debugging advice based on the user input and uploaded files.

III. **Code Execution:** Integration with the Judge0 Rapid API to provide execution capabilities for a predefined set of popular programming languages.

IV. **Core Features:** The system has an interface based on chat where users can communicate with the platform in a seamless manner. It accommodates various programming languages through integration with Judge0 API, and users can write, run, and test code in different languages. The system also includes the feature of uploading and running text-based files like .py, .java, .js, and .txt, which helps users work with their existing files easily. The interface further provides AI-generated responses containing neatly formatted code blocks and execution output to provide a silky smooth and informative user experience.

1.4.2 Out-of-Scope:

I. **Model Training:** This project will not involve the training or fine-tuning of a large language model. It is an application of an existing, pre-trained model (Gemini).

II. **Full-Fledged IDE:** The application is not intended to be a replacement for a complete Integrated Development Environment (IDE) like VS Code or IntelliJ. It will not include features such as project management, integrated version control, or advanced static analysis tools.

III. **Local Code Execution:** All code execution is handled remotely via the Judge0 API in a sandboxed environment. The application will not execute code on the user's local machine.

IV. **User Authentication and Data Persistence:** For the purposes of this project, user authentication systems and long-term data storage for chat histories across sessions will not be implemented.

1.5 Target Audience and Beneficiaries

1.5.1 Students and Teachers

Primary target users. Computer science, data science, and related field students can use the assistant as a 24/7 personal tutor to help them understand difficult topics, debug assignments, and explore code interactively. Teachers can use the tool as an instructional tool to demonstrate concepts and provide students with a safe environment to experiment with code.

1.5.2 New Developers and Students of Coding Bootcamps

Individuals who are new to the software development industry or are undergoing intense training will value the assistant's ability to accelerate their learning curve. They can help them overcome typical setbacks in no time, learn professional conventions of coding, and build confidence.

1.5.3 Professional Developers

Even veteran developers may not be aware of their primary languages in all their details, and the assistant can be an effective productivity advantage. It can be used to produce boilerplate code rapidly, code in lesser-known languages, debug tough problems, or succinctly describe a legacy function, thereby saving valuable time and cognitive effort.

1.5.4 Hobbyist Programmers and Lifelong Learners

Coders of hobby or personal projects might find that the tool is useful for trying out new technologies and languages without the cost of a full-scale development environment.

In brief, the AI Coding Assistant aims to provide high-quality coding help to everyone more readily, making the software learning, building, and debugging process quicker, more convenient, and more enjoyable for the learner and the veteran expert alike.

CHAPTER-II

REQUIREMENTS SPECIFICATION

2.1 Existing System Analysis

Contemporary environment of code support tools, whether integrated development environments (IDEs) or standalone programs, offer many facilities to support developers and students. These typically offer syntax highlighting, autocompletion, basic error detection, and version control integration. It appears, however, that a closer examination finds a variety of deficiencies and areas for improvement in existing mechanisms, particularly with advanced, intelligent code support.

2.1.1 Analysis of Current Code Assistance Platforms

There are also other tools like Visual Studio Code, IntelliJ IDEA, and web-based coding tools like Replit or CodePen, which offer basic coding assistance. All of them also offer integration with static code checkers or linters to highlight issues. There are a few newer tools that have started offeringrudimentary AI-based functionality for suggestions, but these are shallow and narrow in scope.

2.1.2 Limitations of Existing Coding Aid Tools

I. **Limited Explanatory Capability:** While inline documentation or hover-over explanations of library operations are available via tools, few of them are able to explain complex code snippets, algorithms, or design patterns in context and detail. This forces users to repeatedly switch context to external documentation or search engines.

II. **Fragmented Support for Debugging:** Debugging activities continue to be largely manual through the use of breakpoints, step-through execution, and variable inspection. Current tools present the debugging environment but little intelligent support in determining the root cause of logical faults or proposing possible solutions beyond minor syntax errors.

III. **Low-quality Code Generation:** Autocomplete and boilerplate code generation are common, but generative AI features to create useful, functional code snippets from natural language descriptions are underdeveloped or nonexistent in most mainline platforms. This greatly inhibits prototyping and feature creation.

IV. **Lack of Multi-Language Unified Support:** While most IDEs are multi-language, the smart behavior (i.e., advanced autocompletion or refactoring) is language-specific and does not leverage cross-language conceptual knowledge.

V. **Integrated Execution Environments Lack:** Users are normally prompted to compile and run code in alternate terminals or environments that result in context switching and a less smooth development process, especially for less common languages or spur-of-the-moment tests.

2.1.3 Student and Developer Coding Support Mechanism Gaps

Student learning and professional developer productivity are disrupted by individual gaps which are:

I. **Closing the Knowledge Gap:** The students are having difficulty with conceptualization beyond syntax. Currently available tools fail to capture the thinking behind why a specific code construct is employed or how an algorithm performs in reality.

II. **Effective Problem Solving:** Debugging long-standing problems consumes a significant amount of time for developers. No AI-driven debugging suggestions mean greater dependence on human auditing and guesswork, which extends development cycles.

III. **Rapid Prototyping:** The absence of robust, AI-powered code generation for specific functionalities means developers must write more boilerplate code, slowing down the initial stages of project development.

IV. **Accessibility for Non-Experts:** Individuals new to programming or exploring new languages face steep learning curves, as existing tools assume a certain level of foundational knowledge.

2.2 Proposed System: AI Coding Assistant

The AI Coding Assistant project aims to address the aforementioned limitations by providing an integrated, intelligent platform for code generation, debugging, and explanation. Leveraging cutting-edge AI and code execution technologies, the system offers a unique value proposition and an innovative approach to coding support.

2..2.1 Justification for Selection of Gemini and Judge0 APIs

Gemini was chosen for its advanced generative AI capabilities, particularly its proficiency in understanding, generating, and explaining code across various programming languages. Its ability to process complex prompts, including multi-modal inputs (though primarily text/code for this project), and produce coherent, contextually relevant code and explanations is paramount. This directly supports the core features of code generation, debugging (by suggesting fixes and analyzing errors), and comprehensive explanations. Gemini's flexibility allows for natural language interaction, translating user intent into executable code and understandable explanations.

Judge0 is selected for its robust, scalable, and multi-language code execution capabilities. It provides a secure sandbox environment for running code, which is crucial for validating generated code, debugging user-provided snippets, and offering real-time execution feedback. With the ability to support a vast array of coding languages, the AI Coding Assistant is designed to assist users of every level. Its speedy and secure API ensures the experience remains smooth and responsive, exactly how users like it.

2.2.2 Unique Value Proposition of AI Coding Assistant

Here are some reasons why the AI Coding Assistant is unique: In contrast to disjointed tools, it offers one platform for writing, explaining, and debugging code, removing context switching and making the workflow more efficient. Going beyond basic syntax, the assistant uses AI to read the intent behind the code, returning more accurate and insightful feedback.

With its integration with Judge0, the assistant can execute and test code in an instant—providing instant feedback and allowing users to correct errors more quickly. Through automating monotonous tasks and offering detailed explanations, the assistant expedites learning for students and increases productivity for professional developers.

2.2.3 Innovative Approach to Integrated Coding Support

The innovation here is bringing Gemini's generative AI together with Judge0's ability to run code, all wrapped up in a simple Streamlit interface. It introduces a highly effective feedback loop that enhances the coding experience for the user. The cycle begins with user input in the form of natural language questions or uploaded code files. Gemini goes on to execute this input by creating code, giving explanations, or even offering debugging fixes. This code that has been given by the user or created is executed using the Judge0 API, which compiles the code and gives output or relevant error messages. Gemini goes on to execute this output to enhance its explanation or suggestions to create an iterative feedback loop. This is particularly beneficial for use cases like game development, where coding and understanding are a trial-and-error process integral to learning and achieving success. Thus, this integrated environment offers a more interactive, responsive, and tutorial-like coding help experience compared to traditional tools.

2.3 Feasibility Analysis

A comprehensive feasibility study has been undertaken to determine the feasibility of developing and rolling out the AI Coding Assistant. This covers technical, economic, and social factors.

2.3.1 Technical Feasibility

The`app.py` code is a successful implementation of key technologies to be used in the project functionality. The code effectively interacts with the Gemini API to create content, using the `google.generativeai` library to make interactions easier and support capabilities like streaming responses. The code is also successful integration with the Judge0 API for executing code in different programming languages. This is done by employing the `requests` module to issue API requests, like posting code and polling for the execution result, and offering a robust and reliable backend. On the frontend side, Streamlit is employed to build a clean and interactive user interface, managing UI elements and session state in a light way. Overall, technical integration of Gemini, Judge0, and Streamlit is very possible, with most of it already in place and working as planned.

The system performance, as expressed in the app.py code, should in general be robust, with some thoughts about API response time. The content generation in Gemini benefits from streaming, which reduces the effects of network and processing latency by returning responses incrementally and enhancing the user's responsiveness perception. Judge0's response time is essential in giving timely feedback, and the script factors in potential delays through the use of a polling mechanism via time.sleep(1), demonstrating an awareness of potential execution queues. Judge0 is generally quick enough for dealing with basic programming issues, and thus it is suited for this purpose. On the client side, Streamlit's architecture makes the app re-run on every user input. While this behavior supports rapid prototyping, it requires careful session state management—already addressed in app.py—to avoid unnecessary re-renders that could negatively affect responsiveness. Overall, the system is expected to perform well, with the main areas for future optimization being API-related latency.

The architecture of the system exhibits high scalability capability through the use of managed cloud services and high availability APIs. The Gemini API, offered by Google, is scalable by design and can process lots of requests, with all scalability being taken care of by the service provider. In the same vein, Judge0 is a commercial quick API service designed for concurrent code execution at scale so that high traffic and simultaneous user requests can be handled well without specialized infrastructure. At the deployment end, the Streamlit application can be deployed on leading cloud providers like Google Cloud, AWS, or Azure and then horizontally scaled as more user

demands come in. In general, the dependency on strong, scalable APIs and adaptable deployment options places the system in a solid position to accommodate future growth and use.

2.3.2 Economic Feasibility

Cost factors for the project revolve around API usage, deployment infrastructure, and staff. Both Gemini API and Judge0 Rapid API practice consumption-based pricing schemes, and the initial development phase can take advantage of free tiers or cheap plans. But for wider deployment, planning in advance to budget for API usage in terms of the anticipated number of requests will be required. The existing `app.py` script rightly has provisions for dealing with API keys, like `GOOGLE_API_KEY` and `JUDGE0_API_KEY`, in line with this necessity. From the deployment perspective, Streamlit's open-source platform has no licensing costs, but running the application on cloud environments like AWS or Google Cloud will come at a cost of compute resources (CPU, RAM) and data transfer. These expenses can be minimized, nonetheless, with effective resource optimization. At the personnel level, the project needs Python developers and basic AI/ML understanding to handle API integration and app logic. At large, the main development costs are incurred in API usage and cloud deployment, which makes the project economically feasible based on available scalable and affordable service options.

The AI Coding Assistant presents a strong value proposition to deliver considerable time and cost savings in several domains. Its AI-powered code generation and debugging capabilities significantly cut the time developers devote to composing boilerplate code and debugging issues, allowing projects to be completed at a faster pace and reducing labor costs. To learners and students, the assistant enhances learning efficiency by providing instant explanations and feedback, which can minimize overreliance on costly tutoring services or lengthy online tutorials. Also, through the unification of various functionalities—writing code, executing it, and debugging—into one single application, it does away with the hassle of working on several platforms, thus conserving the user even more time and effort. In general, the assistant offers good advantages in productivity, learning, as well as in optimizing resources, and as such is a helpful and cost-saving solution for its targeted users.

2.3.3 Social Feasibility

The project provides significant social value through encouraging accessible and successful programming education. Through in-depth code explanations, it enables

conceptual understanding for users, so students are not only understanding how code is written, but why they are doing it in a certain way. Its fast prototyping and smart debugging features speed up skill acquisition by students and professionals, so they can learn faster and solve problems more confidently. Most significantly, the tool democratizes access to expert-level coding support, offering expert-level advice to those who do not necessarily have access to individual mentorship or formal education. Overall, the project is a socially responsible effort that supports learning, accessibility, and proficiency in programming.

The system is well-considered in terms of accessibility as it is designed to be inclusive to users of all levels, ranging from novice users to professionals. Streamlit's simplicity permits the development of simple and uncluttered user interfaces with low cognitive load, evident in the well-organized and styled chat interface presented in `index.html`. The tool also accommodates multiple programming languages through Judge0, leveraging the `LANGUAGE_TO_JUDGE0_ID` mapping in `app.py`, which makes it universal and practical for a global community with varying coding backgrounds. The inclusion of Gemini also makes it possible for natural language interaction, which means that users can interact with the system conversationally, greatly lowering the barrier to entry for new users. Overall, the system is designed with great usability and accessibility in mind, so that advanced coding support is made available to any number of users.

2.4 Requirement Specifications

This section describes the AI Coding Assistant's functional and non-functional requirements and some of the UI.

2.4.1 Functional Requirements (FR)

FR1: Live Code Generation: The system is designed with several important functional requirements to enable it to be functional and context-sensitive code generation effective. It will first generate code snippets or complete functions in natural language user input (FR1.1), allowing simple interaction and reducing the need for precise syntax commands. The code will be presented as an independent, clearly defined code block with proper demarcation from any given comment to enhance readability and user experience (FR1.2). Further, the system is capable of processing an extensive variety of programming languages, i.e., those listed in LANGUAGE_TO_JUDGE0_ID mapping, i.e., Python, JavaScript, Java, C++, C#, Ruby, Go, Swift, Kotlin, PHP, Rust, TypeScript, and SQL, and generic markup and

documentation languages such as HTML, CSS, Markdown, and Scala (FR1.3). Also, it will be capable of incorporating context from external related text or code files when generating new code in such a way that outputs are responsive and relevant to the user's needs (FR1.4). Combined, these requirements provide a strong, flexible foundation for smart code generation.

FR2: Multi-language Code Debugging: The system will have sophisticated debugging functionality with the purpose of assisting users in identifying and fixing mistakes in their code. It will accept user-submitted code snippets or full code files for debugging in all the supported programming languages (FR2.1). Once the code is submitted, the system will identify and beautifully present typical syntax and logical mistakes, allowing users to understand the kind of mistakes (FR2.2). In addition to indicating problems, it will give suggested solutions or improvements, driving users towards more efficient coding practices and best fixes (FR2.3). The system will also execute the user's code and display either the standard output or any error messages, enabling users to observe the runtime output and behavior in real time (FR2.4). These features combined deliver an end-to-end, interactive debugging experience.

FR3: Code Explanation in Depth: The product will feature comprehensive code explanation features that will assist users with their understanding. It will provide rich explanations of the user-submitted code, on a line-by-line or block-level basis, depending on the input complexity and organization (FR3.1). The explanations will cover the purpose of the code, the logic underlying it, and the potential implications or impact, thus allowing users to grasp the functionality of the code as well as the context (FR3.2). In addition to user-code submitted by the users, explanations shall also be made available for any code that is generated by the AI assistant so that transparency and intelligibility in AI-generated output are established (FR3.3). All explanations shall be given in a straightforward and easily understood form, separate from the code block itself, so that it is readable and cognitive load is minimized (FR3.4). This capability is vital for both learning and working purposes, which makes the system a useful tool for understanding code.

FR4: File Upload Mechanism with Secure Files: The system will have file processing functionality to enhance contextual understanding and from users. The system will allow bulk file uploads by the user, and the AI will translate context or process content from the files (FR4.1). It will be able to hold a range of standard text-based file formats, including .py, .js, .java, .txt, .md, .html, and .css, and image formats

like .png, .jpg, and .gif, to allow for flexibility in the input format users can provide (FR4.2). After uploading files, it will present file names and types on the user interface, providing users with transparency and assurance prior to forwarding their requests (FR4.3). In addition, the system will handle all files uploaded securely, such that file data is utilized only for the given interaction and not stored or propagated outside of its functional need (FR4.4). These aspects result in a secure and seamless experience along with developing the ability of the assistant to offer accurate and context-sensitive responses.

FR5: Code Execution Validation: It is intended to provide secure and effective code execution features as one of its fundamental functions. It will include a mechanism to run code produced by the AI assistant, with the ability to run user-provided code as well, to enable debugging and testing within the platform (FR5.1 and FR5.2). For safety purposes, the execution environment will be sandboxed to block any code that might be harmful from impacting the system or other users (FR5.3). The output of code execution—standard output, error message, compilation error, and run-time status—will be clearly and thoroughly shown to the user to provide complete transparency (FR5.4). In addition, the system should accommodate code execution for multiple languages available through the Judge0 API, like Python, JavaScript, Java, C++, and C# (FR5.5). Such provisions guarantee a strong, secure, and user-friendly interactive coding environment.

2.4.2 Non-Functional Requirements (NFR)

NFR1: Performance: The system is engineered to provide an efficient and responsive user interface, in accordance with certain non-functional requirements. The system will respond to user requests—either for generating or explaining code—within 3 seconds for at least 90% of requests, to provide rapid interaction and sustain user interest, except for extremely complex requests (NFR1.1). In addition, code execution via the Judge0 API will finish and report results in 10 seconds for 95% of the requests, omitting instances of very long-running or heavy-resource requests (NFR1.2). These performance standards are established to allow the system to function as smoothly as possible and to meet users' expectations of speed and dependability.

NFR2: Reliability: The system is designed with reliability and high availability in consideration, as seen in its non-functional requirements. It will have 99% uptime for interactions with both the Gemini API and Judge0 Rapid API to provide constant access to code generation and execution services without frequent interruptions

(NFR2.1). In addition, the Streamlit application will exhibit high system stability, running without crashes or unforeseen errors during 99.9% of user sessions, thus rendering a seamless and reliable user experience (NFR2.2). These needs highlight the system's focus on performance and operation ruggedness.

NFR3: Usability: The system focuses on usability in order to provide support for a wide variety of users, from students to business developers. Its interface should be intuitive and simple to use, so that people with different technical proficiency levels can use the platform with ease (NFR3.1). In addition, the system will also be designed with a strong sense of learnability so that new users can easily grasp its fundamental features and be able to successfully execute their first query within a mere five minutes of initial usage (NFR3.2). Such usability standards will make the platform both accessible and efficient for everyone.

NFR4: Compatibility: The system must provide wide accessibility and a uniform user experience in various environments. Its web interface must be functional and look alike on all predominant web browsers like Chrome, Firefox, Safari, and Edge, and across popular operating systems like Windows, macOS, and Linux (NFR4.1). Besides, the user interface will have a responsive design that changes in a seamless manner as per different screen sizes and devices to ensure that it is usable on desktops, laptops, tablets, and smartphones as well (NFR4.2). These non-functional requirements ensure a smooth and user-friendly experience no matter what platform or device is used.

NFR5: Security & Privacy: The system is developed with robust security and privacy controls to guard user data and secure platform integrity. It will not keep user chat history or contents of uploaded files for longer than the current session without definite user approval, upholding user privacy at all times (NFR5.1). To ensure secrecy of sensitive data, Gemini and Judge0 API keys will be safely stored in environment variables and never be disclosed on the client-side, as properly done in app.py (NFR5.2). The system will also sanitize every user input and code submission to prevent injection attacks and other security issues (NFR5.3). Finally, code execution will take place in a secure and isolated sandbox environment offered by Judge0, so that no malicious or arbitrary code can impact the system or damage other users' data (NFR5.4). These non-functional requirements allow the platform to run in a safe and responsible manner.

2.4.3 UI Requirements

UIR1: Clean, Intuitive Streamlit Interface: The user interface of the system is made to deliver a comfortable and interesting experience, revolving around a chat interface that forms the key way of interaction, much like current messaging apps (UIR1.1). Important input components like the text area, file upload icon, and send button will be clearly visible and readily accessible at the screen's bottom, providing easy and seamless user interaction (UIR1.2). For the sake of clarity and ease of reading, AI-synthesized answers—such as code blocks, descriptions, and regular text—will be visually separable from inputs from users, employing attributes such as alternative background colors and formatting (UIR1.3). These interface design decisions will simplify usability and facilitate communication throughout the application.

UIR2: Responsive Design: The chat interface has been created with responsiveness and adaptability for a smooth user experience across platforms. The chat container will be automatically resized to accommodate varying screen sizes, and there will be vertical scrolling for longer chat histories without compromising usability (UIR2.1). The input field will be permanently located at the base of the screen, holding a fixed position at all times but shifting its layout dynamically on smaller screens to ensure that all interactive elements are visible and usable (UIR2.2). When several files are uploaded, the file preview pills will wrap onto two or more lines when necessary, in effect supporting many uploads without adding horizontal scrolling (UIR2.3). These responsive design elements lead to a simple and user-friendly interface on every screen size.

UIR3: Clear Language and Programming Language Selection: For increased clarity and control in interactions, the user interface will have a visible dropdown or selection mechanism through which users can indicate the programming language context of their query, with options of "Python," "JavaScript," or "Auto-detect" for flexibility and accuracy (UIR3.1). When the system produces code, the resultant code block will show the language clearly using syntax coloring and a proper language tag (e.g., python) so that users can easily identify and read the code shown to them (UIR3.2). Such functionality enhances a better and language-conscious coding experience.

UIR4: Error Message Clarity: The system is intended to graciously handle errors through adequate information made available to users in case an error occurs. All errors that happen during processing by the AI or during code execution will be shown distinctly and briefly on the chat interface, so users are aware without disrupting the continuity of interaction (UIR4.1). These error messages will contain enough detail to

enable users to see the type of problem, e.g., "API request failed" or "Execution error: [Judge0 specific error]," and give actionable feedback for debug (UIR4.2). This will promote transparency and help with a more seamless user experience even in case of a problem.

UIR5: Minimal Cognitive Load for Users: The user interface is carefully designed to encourage simplicity and usability by reducing visual clutter and displaying information in neat, consumable pieces (UIR5.1). Principal interactive components such as buttons and selectors shall have intuitive icons or simple labels, which shall allow them to be easily recognizable and understandable (UIR5.2). In addition, the overall flow of interaction from user input to AI response should be natural and predictable in order to allow for seamless and intuitive user experience that facilitates effective navigation and engagement (UIR5.3). All these design principles combine to provide a streamlined and accessible interface for all users.

2.5 Use Cases

This section outlines various use cases that demonstrate the utility and application of the AI Coding Assistant for its target user base.

2.5.1 Student Learning and Practice

A computer science student is learning a new data structure (e.g., a linked list) in Python. The student asks: "Generate Python code for a singly linked list with insert, delete, and display methods, and explain each part." Delivers Python code implementing a linked list, accompanied by a comprehensive, step-by-step explanation of the class design, method functions, and associated time complexities. The student can then execute the code to observe its behavior in real time.

2.5.2 Professional Developer Rapid Prototyping

A software developer requires a fast solution to extract data from a JSON response received via a web API using JavaScript. The developer uploads a sample JSON file and asks the assistant: “Create a JavaScript function that reads this JSON and retrieves all values under 'product_names'.” Gives the Python code for the linked list and then a line-by-line explanation of the class structure, functions, and their time complexity. The student can then execute the provided code to observe it working.

2.5.3 Programming Language Exploration

An experienced Python developer wants to see how a specific concurrency pattern (e.g., Go routines) is implemented in Go. The developer asks: "Show me an example of using goroutines and channels in Go for a simple producer-consumer pattern." The assistant generates a complete, runnable Go program demonstrating the producer-consumer

pattern with goroutines and channels, along with an explanation of Go's concurrency model. The developer can execute the Go code immediately.

2.5.4 Quick Algorithmic Problem-Solving

A junior developer is given a sophisticated piece of C++ code from a senior developer and has to comprehend its logic. The user asks: "Solve the Two Sum problem in Java, given an array of integers and a target sum, return the indices of the two numbers that add up to the target." The assistant provides an efficient Java solution (e.g., using a HashMap), explains the logic and complexity, and allows the user to execute it with custom test cases. If the user's initial attempt had errors, they could upload it for debugging.

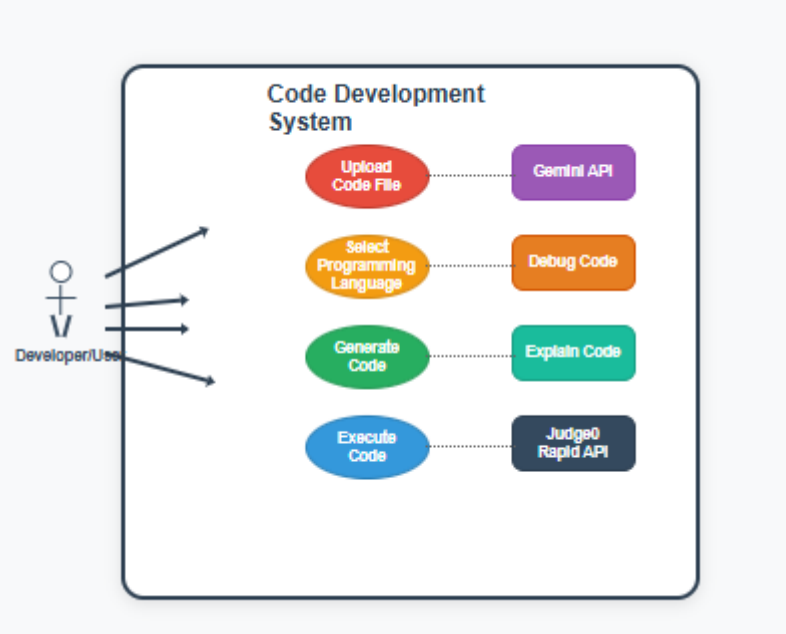


Figure 1: Use Case Diagram

In Figure 1, the Code Development System is illustrated as an interactive platform that assists developers or users throughout various stages of the coding process. The system enables the user to upload a code file, which is then processed using the Gemini API for analysis and suggestions. Next, the user selects a programming language to initiate the debugging process, powered by the Debug Code module. The system further provides a Generate Code tool that aids in generating new code blocks, elucidated and substantiated by the Explain Code part. Lastly, the code can be run on the Judge0 Rapid API.

CHAPTER-III

SYSTEM DESIGN

This chapter explains the system design of the AI Coding Assistant. It includes the architectural framework, component-level design, data structures, and the user interface design, giving a complete plan for the development of the system.

3.1 Architecture Design

It uses a three-tier architecture. The pattern separates the application into a presentation tier, logic tier, and data/service tier. The separation enhances modularity, scalability, and maintainability.

I. **Presentation Tier (Frontend**): It is the interface, developed using the Streamlit framework. It does the work of displaying the chat interface, processing user input, and showing responses by the assistant. The implementation is done according to a contemporary, responsive web design.

II. **Logic Tier (Backend):** This tier is the core of the application, written in Python (app.py). It handles all business logic, including managing the user session, processing inputs, orchestrating API calls to external services, and formatting the data to be displayed on the frontend.

III. **Service Tier (External APIs):** This level includes third-party platforms offering the fundamental smart functionalities. The Google Gemini API is used for any generative operations like code generation, debugging, and explanation. The Judge0 Rapid API is also used to run code snippets in a safe, sandboxed environment that is multi-language compatible.

3.1.1 High-Level System Model

A user interacts with the system through a User Interface, which is a Streamlit web application. This interface communicates with managers for file uploads and chat history to manage user inputs and conversation history. The user's request is then passed to the Core Processing Engine, where a request processor, response handler, and context analyzer work together to manage the query. The engine utilizes two main interfaces to connect with external services for its core functions. One interface connects to Google Cloud Services for generative capabilities, while the other links to the RapidAPI Gateway for executing code securely across various programming languages.

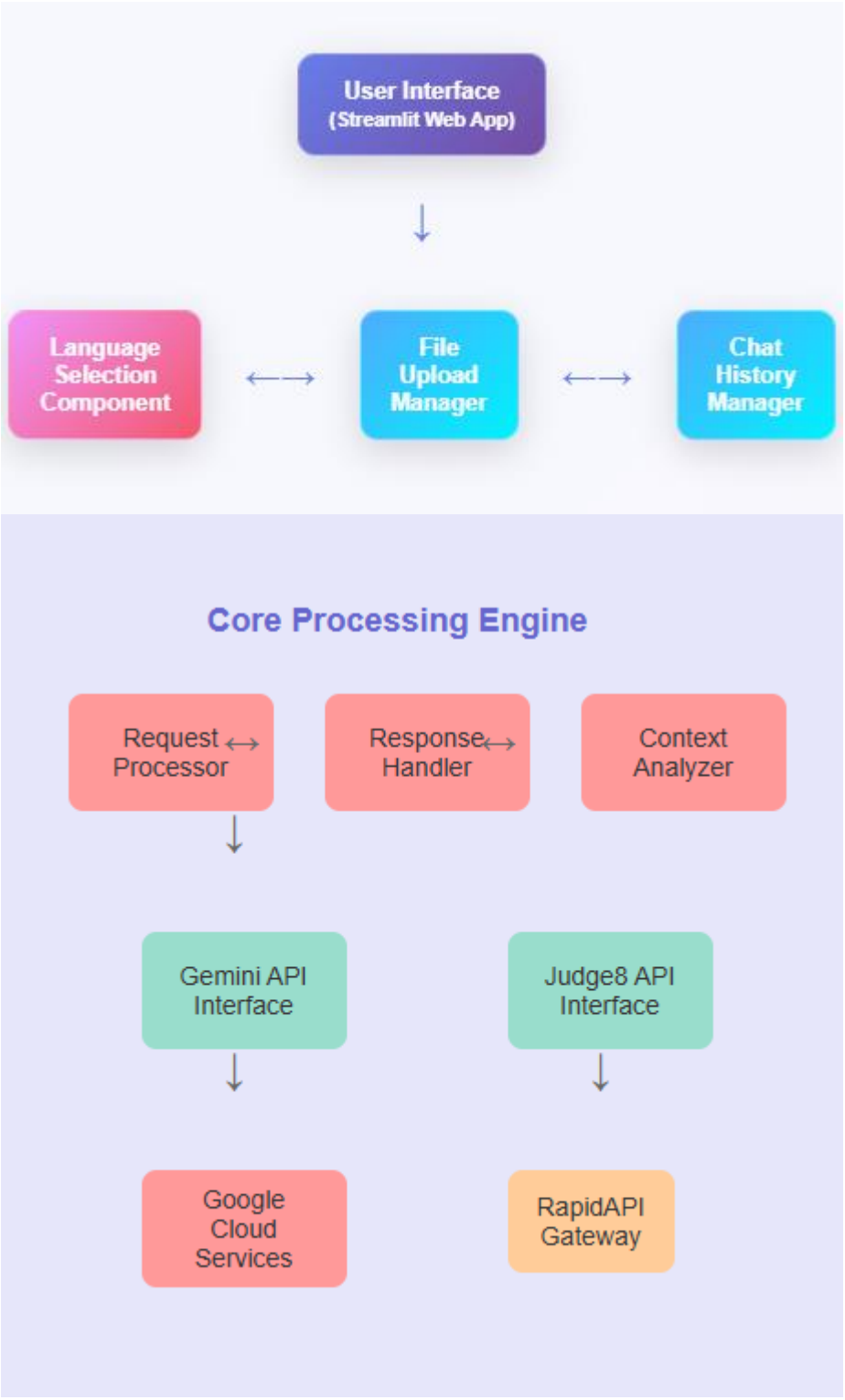


Figure 2: High-Level System Architecture

In Figure 2 Arrows indicate the flow of interaction: A user communicates with the system via a User Interface, which is realized as a Streamlit web application. The User Interface talks to a File Upload Manager, which acts as an intermediary between the

user and the backend entities. The File Upload Manager, in its turn, communicates with the Language Selection Component and the Chat History Manager to manage language choices and keep track of the conversation history.

The request made by the user is passed to the Core Processing Engine, where it is initially processed by a Request Processor. This is the component that handles the request that has arrived and orchestrates the operations that must be executed. Inside the engine, the Request Processor collaborates with a Response Handler and a Context Analyzer in order to properly orchestrate the conversation, making responses contextually relevant.

The Core Processing Engine interacts with external services via two principal interfaces. The Gemini API Interface provides the engine with access to the Google Cloud Services, thus providing the engine with sophisticated generative AI capabilities. The Judge0 API Interface provides the engine with interaction with the RapidAPI Gateway, which provides the system with the ability to run code securely across multiple programming languages.

3.2 Component Level Design

Based on the document, Figure 3 illustrates the key software components of the programming assistant, structured into distinct, interacting layers. The User Interaction Layer handles all user-facing activities and includes the User Interface Component, Session State Management, a File Upload Handler, and a Language Selection Controller. These frontend elements communicate with the central Core Processing Engine, which acts as the application's backend hub.

The engine orchestrates the system's primary functions, which are handled by the core functionality modules. These include the Code Generation & Explanation component and a Code Execution Controller. To deliver these services, the system integrates with two external platforms: Google Cloud Services for its advanced generative features and the Judge0 Rapid API for its secure code execution service.

3.2.1 Component Diagram

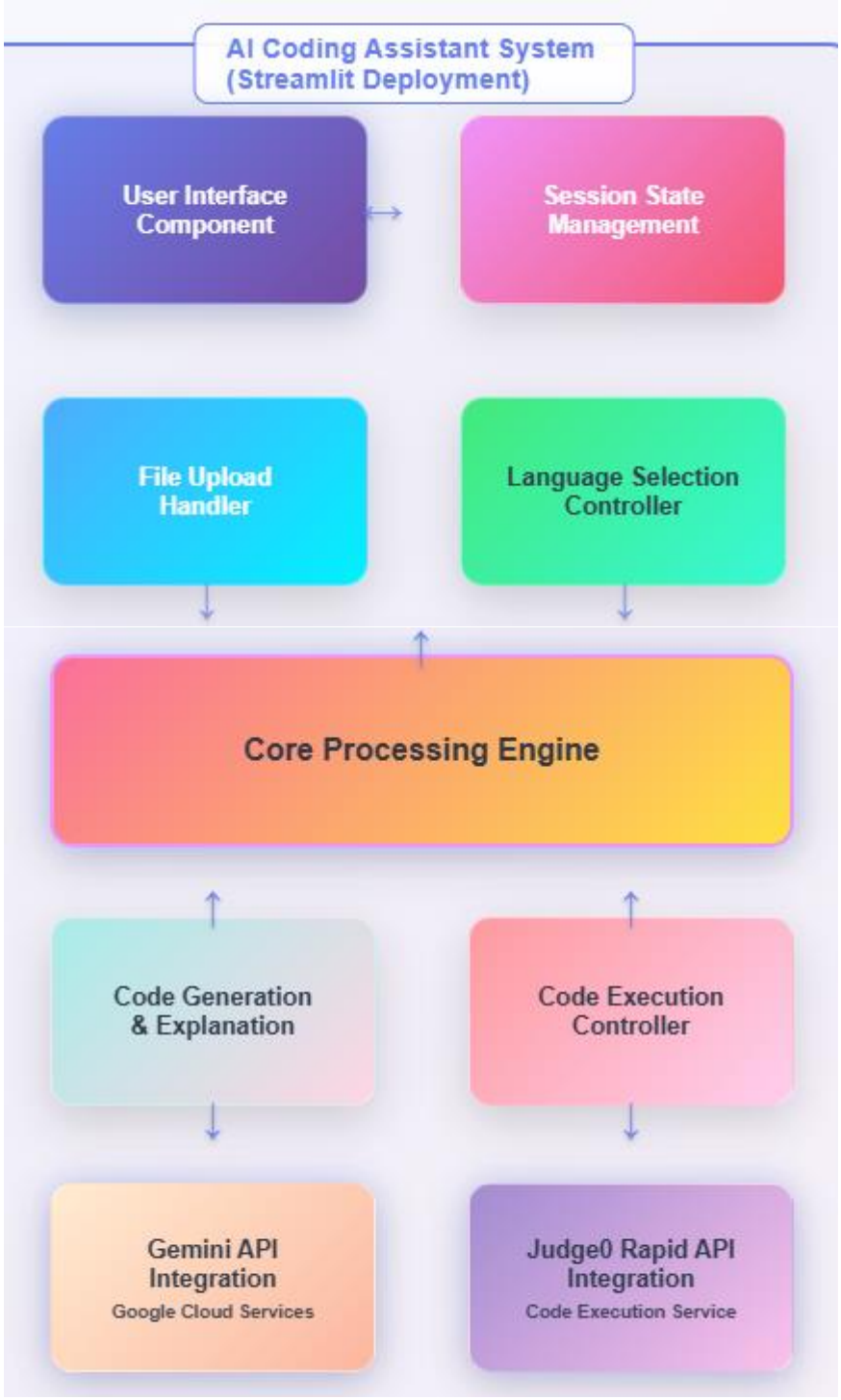


Figure 3: System Component Diagram

In Figure 3 showing the software components within the " SMART CODER: AI POWERED PROGRAMMING ASSISTANT:

:

This system includes several integrated components working together. The User Interface Component is what the user sees and interacts with. Session State Management keeps track of the user's current session and data. The File Upload Handler manages any files the user uploads. The Language Selection Controller allows the user to choose a programming language. The Core Processing Engine serves as the central brain of the system, connecting and managing all other parts. Code Generation & Explanation creates the code and explains what it does. The Code Execution Controller manages the running of the code. Gemini API Integration (Google Cloud Services) connects to Google's Gemini model for AI-powered features. Judge0 Rapid API Integration (Code Execution Service) uses an external service to securely run the code.

3.2.2 Activity Diagram

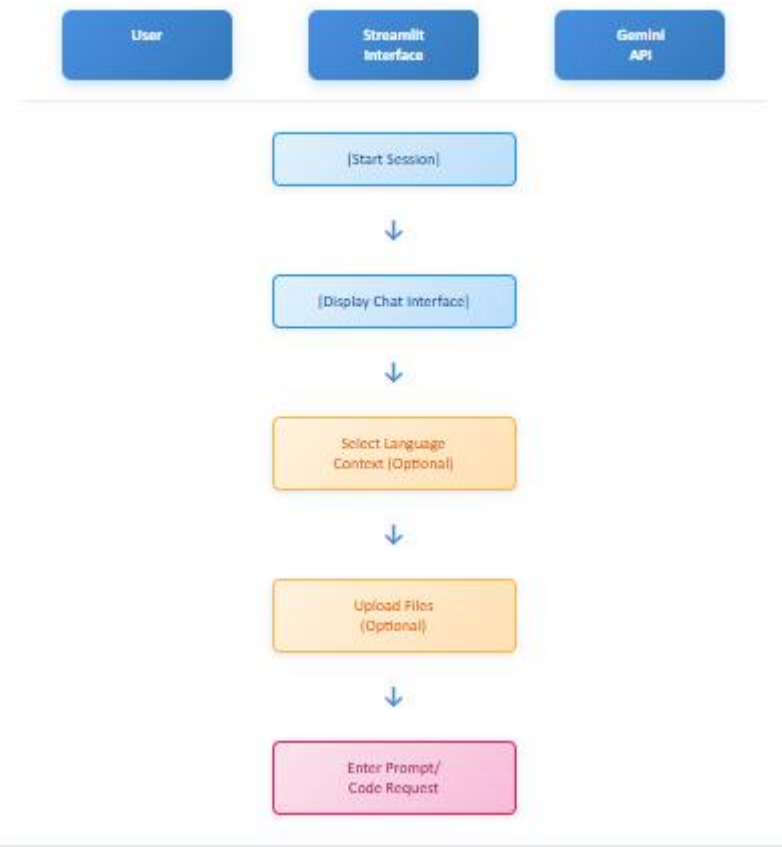


Figure 4.1 : User Interaction Activity Diagram

In Figure 4.1, a step-by-step flow of user interaction with the AI-powered coding assistant through the Streamlit interface is depicted. The process begins when the user initiates a session, which activates the Streamlit interface to display the chat interface. This interface serves as the primary medium through which the user communicates with the system. The user can then optionally select a programming language context to provide better clarity for code-related queries. Also, the system supports the upload of files, which improves the understanding of the context or codebase by the AI model. After setting the context, the user then inputs a prompt or request relating to the code, for instance, requesting code generation, debugging, or explanation. The request is then passed on to the Gemini API for processing. Incorporating optional context and file upload makes the system provide more accurate and contextual responses.

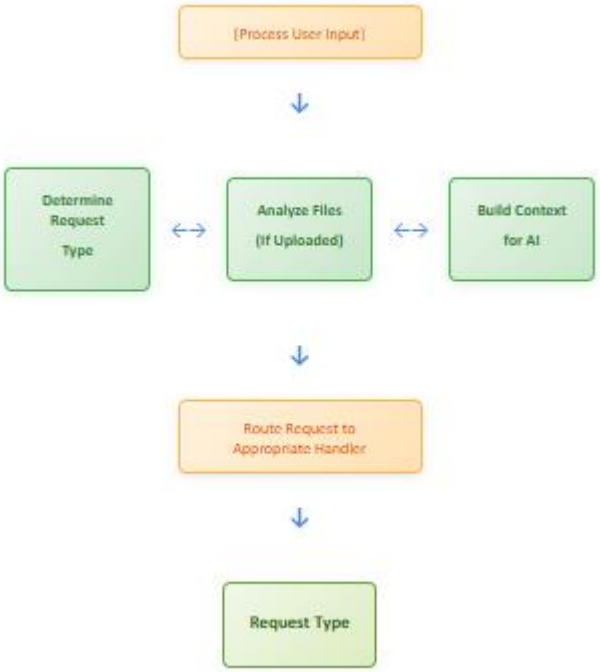


Figure 4.2 : User Interaction Activity Diagram

In Figure 4.2, the workflow of internal processing of user input by the AI coding assistant system has been presented. The process starts from the point where the system takes in and processes the user input, which could be code prompts, queries, or uploaded documents. It first identifies the nature of the request (e.g., code generation, explanation, or debugging). If the files are uploaded, the system goes on to process the files to get relevant information or code snippets. At the same time, it constructs contextual knowledge for the AI model, improving the accuracy of the response. All of these steps are interlinked to make sure that the AI gets all the context required. When analysis and context construction are done, the system sends the request to the respective handler based on the request type found. Lastly, the precise request type is called to ensure a sharp and smart response is provided to the user.

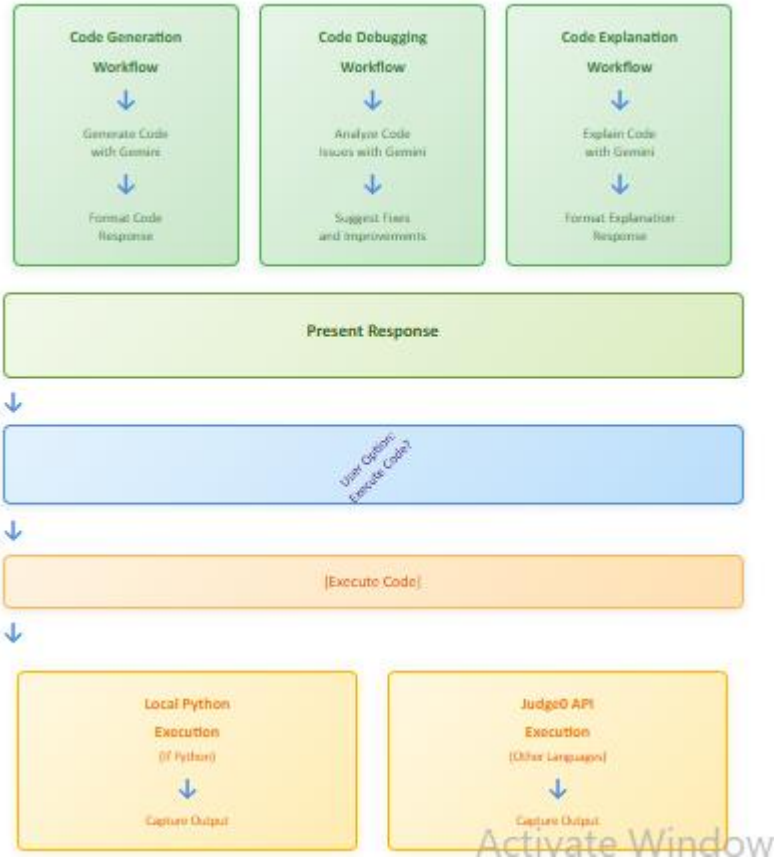


Figure 4.3 : User Interaction Activity Diagram

In Figure 4.3 Code Processing & Execution Flow, the system processes the request through one of the following workflows: Code Generation: Generates code with Gemini and formats the response. Code Debugging: Analyzes code issues, suggests fixes, and formats the response. Code Explanation: Explains code with Gemini and formats the explanation. The system presents the final response to the user. The user has an option to execute the code. If the user chooses to execute the code, the system performs one of the following: Local Python Execution: For Python code, it is executed locally and the output is captured. Judge0 API Execution: For other languages, the code is executed via the Judge0 API and the output is captured.

3.2.3 Sequence Diagram

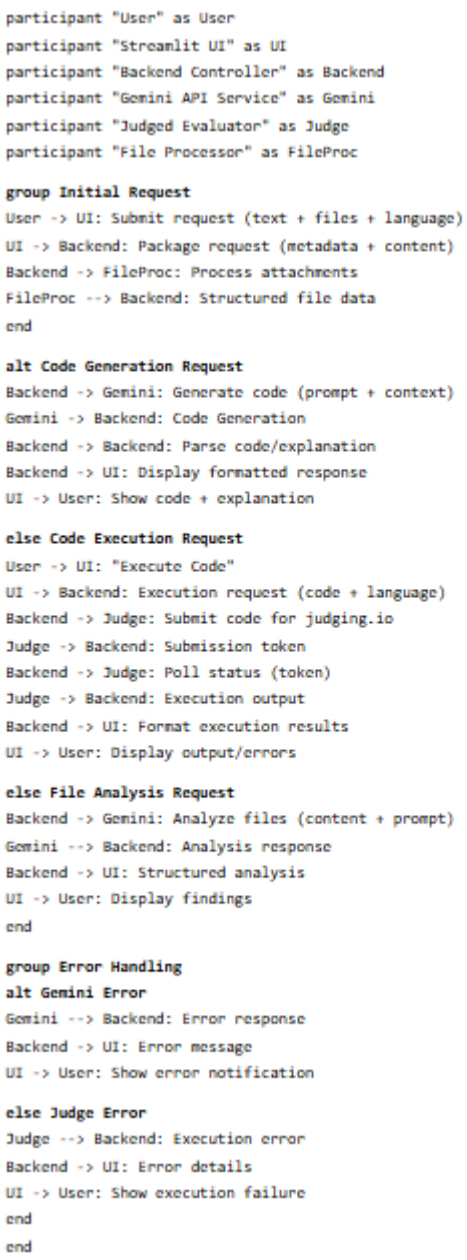


Figure 5: Request-Response Sequence Diagram

In Figure 5 A sequence diagram showing the interaction for a typical user query: The User submits a request (text, files, language) to the UI. The UI sends this request to the Backend. The Backend forwards any attached files to the File Processor for processing. The File Processor returns the structured file data back to the Backend.

If it's a Code Generation Request: The Backend requests Gemini to produce code from the user's prompt and the context given. After Gemini has processed the request, it sends the produced code back to the Backend. The Backend then structures

the response, with or without explanation, and sends it back to the User Interface. Finally, the UI presents the result to the user in an organized and understandable format. If it's a Code Execution Request: When the user hits "Execute Code" on the UI, the UI requests execution to the Backend. The Backend posts the code to Judge0 to be executed, and Judge0 responds with a submission token. The Backend uses this token to repeatedly poll Judge0 until the output of execution is ready. When Judge0 responds with the execution output, the Backend parses the output and passes it to the User Interface, which displays it to the user.

If it's a File Analysis Request: The Backend requests that Gemini analyze the uploaded files together with the user's question. Gemini performs this analysis and sends an analysis response back to the Backend. The Backend then organizes the analysis and passes it to the User Interface, where the results are presented to the user in an understandable format. For Error Handling: When Gemini provides an error response, Backend catches the error and passes a corresponding error message to the User Interface, which shows a notification to the user. In the same way, in case Judge0 faces an execution error, Backend passes the error information to the UI, which shows a failure notification to the user to immediately notify them of any processing problem.

3.3 Data Design

The system does not utilize a persistent database; instead, it maintains state within the user's session. The primary data entities are transient and managed by Streamlit's session state mechanism.

Business Entities

- **UserSession:** Represents a single user's interaction with the application. It contains the entire chat history and the current state of the UI.
- **ChatMessage:** A single message in the chat, either from the user or the assistant. It contains the text, code, explanations, and metadata about any attached files.
- **StagedFile:** Represents a file that has been uploaded by the user but not yet submitted with a prompt.

Entity-Relationship Diagram (ERD)

Figure 6: Conceptual ERD

In Figure 6 showing the relationships between session-based entities:

- **UserSession** (Entity): Contains attributes like session_id.
- **ChatMessage** (Entity): Contains attributes like message_id, role, content, timestamp.
- **Relationship:** A UserSession "has" one or more ChatMessages (one-to-many relationship).

Data Dictionary

Table 1: Data Model Table

Table/Entity	Field Name	Data Type	Description
ChatMessage	is_user	Boolean	True if the message is from the user, False if from the assistant.
ChatMessage	prompt	String	The text content of the user's message.
ChatMessage	code_response	String	The code block generated by the assistant.
ChatMessage	explanation_ response	String	The explanation text generated by the assistant.
ChatMessage	language	String	The programming language detected or selected for the code.
ChatMessage	attached_files_ info	List	A list of dictionaries containing metadata about files attached to the message.
ChatMessage	error	String	Stores any error message that occurred during processing.
UserSession	chat_history	List	A list containing all ChatMessage objects for the session.
UserSession	staged_files	List	A list of uploaded file objects waiting to be sent.
UserSession	code_ execution_ output	Dictionary	A dictionary mapping message indices to their code execution results.
UserSession	selected_ language	String	The language currently selected in the language selection dropdown.

3.4 User Interface Design

The user interface is a critical component, designed to be intuitive, responsive, and efficient.

3.4.1 UI Design Decisions

A minimalist, single-page application (SPA) with a familiar chat-style interface. The design prioritizes ease of use and a clear, uncluttered workspace.

The UI uses a system font stack (-apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto...) for a native feel and optimal readability across different operating systems. Code snippets use a monospaced font (Consolas, 'Liberation Mono', Menlo...) for clarity.

The interface uses a clean and professional color scheme, with a primary blue (#007bff) used for user messages and interactive controls, and light gray (#e9ecef) being used for assistant messages to maintain a high visual contrast. Code blocks are colored with a dark background (#282c34) to achieve high contrast and readability. The layout is made wide and responsive through the use of Streamlit. The primary conversation container takes most of the screen real estate, with the input controls stationary on the bottom for neat and uniform user input throughout the session.

The UI layout was originally created with HTML and CSS, essentially a live prototype to experiment and see the interface in real-time. In the interest of thesis documentation, formal mockups are intended to be drafted using purpose-built design tools like Figma or Balsamiq to provide a cleaner and more standardized representation. The ultimate user interface is deployed via the Streamlit Python library, which provides quick development through direct translation of Python scripts into an operational and interactive web interface.

3.4.2 Screen Layouts

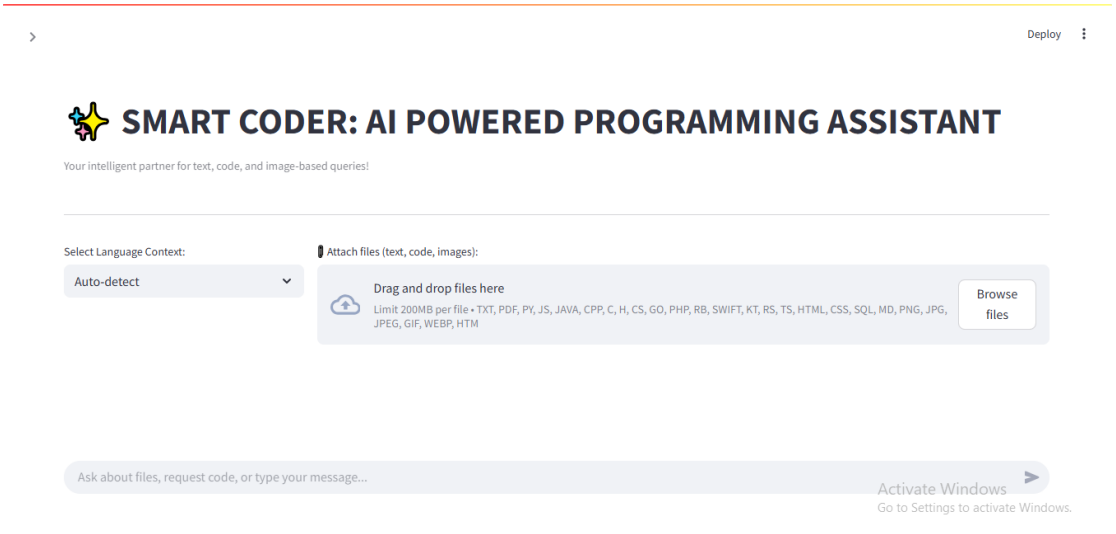


Figure 7: Main Application Screen Layout

In Figure 7 A digital mockup of the fully rendered UI, showing: The chat interface has a conversation design with the user's messages on the right with a blue background and the assistant's messages on the left with a gray background. Every assistant message contains a well-formatted code block with syntax highlighting followed by an "Execute Python Code" button and a text-based explanation.

3.4.3 UI Implementation:

The UI is implemented entirely in Python using the Streamlit library. Key Streamlit widgets used are:

- st.set_page_config(): To set the page layout to wide and define the title/icon.
- st.chat_message(): To create the distinct user and assistant message containers.
- st.chat_input(): To provide the primary text input field at the bottom of the screen.
- st.file_uploader(): To handle single or multiple file uploads.
- st.selectbox(): For the programming language selection dropdown.
- st.button(): For the "Execute Code" and "Clear Chat" functionalities.
- st.markdown() and st.code(): To render text explanations and formatted code blocks, respectively.

CHAPTER-IV

IMPLEMENTATION AND TESTING

This chapter outlines the technical environment used for the development of the AI Coding Assistant, details the implementation of its core modules, and presents a comprehensive testing strategy designed to ensure the application's functionality, reliability, performance, and security.

4.1 Implementation Environment

Development of the AI Coding Assistant was done with a current technology stack selected for its strength, adaptability, and speed in development.The whole application is coded in Python. Python's wide collection of libraries for web development, data processing, and API integration makes it the best fit for this project.

The project leverages a few essential technologies in service to its operations. Streamlit acts as the main web application framework, enabling quick web application development of interactive, data-centric web applications from Python scripts. For features based on AI, the Google Generative AI SDK (google-generativeai) is used, giving official support for the seamless interaction with the Gemini API. The requests library, a built-in Python tool, is utilized to make HTTP requests, specifically for sending requests to the Judge0 Rapid API to run code. python-dotenv is utilized to handle environment variables to securely store API keys and other sensitive configuration information.

Visual Studio Code (VS Code) was utilized as the main code editor during development, providing strong support for Python development, effective debugging capabilities, and integrated Git support. Version control was done using Git to keep track of changes and code revisions, and GitHub acted as the central hub for hosting source code, allowing collaborative development and easy project management.

4.1.1 Integration Mechanisms:

- **REST APIs:** The system revolves around integrating two of the most important external REST APIs: the Gemini API for AI logic and the Judge0 API for code execution. Communication is conducted through HTTPS requests with payloads in JSON format.

4.2 Module Implementation

The logic behind the application, although encapsulated in a single script app.py for Streamlit deployment, is logically organized into separate modules, with one module handling a particular set of functionalities.

4.2.1 AI Core Module (Generation, Debugging & Explanation)

This module is the smart brain of the assistant, fueled by the Gemini API. It manages new code generation, existing code analysis and debugging, and creation of rich explanations.

The generate_content_streaming function is the primary method to call when accessing the Gemini API and generating dynamic AI responses. It constructs a very specific prompt that guides the output of the model in an organized and reader-friendly manner. The prompt includes a system directive to print the response with a ---EXPLANATION--- delimiter, giving an explicit separation between generated code and explanation. In addition, it adds the content of any file uploaded, such as text, images, and PDF data, to provide context-carrying importance. The approach also accounts for user-supplied programming language, with the benefit of programming the output to correct syntax and semantics. Finally, it adds the user's explicit question, constituting a coherent and context-carrying input to Gemini. This design supports very relevant and well-formed AI outputs.

The module streams the API response, which is then parsed to split the code_response from the explanation_response using the custom separator.API calls are wrapped in try...except blocks to gracefully handle potential issues like network failures or API errors, displaying a user-friendly error message in the chat interface.

4.2.2 Code Execution Module

This module provides the ability to run code snippets directly from the application. It supports both local execution for Python and remote execution for a wide range of other languages. It leverages the Judge0 Rapid API for all non-Python languages.

The execute_python_code function takes care of running Python code inside a limited exec() context. It safely captures and returns standard output by redirecting stdout, allowing the output to be displayed or logged without impacting the system. The execute_with_judge0 function, on the other hand, communicates with the Judge0 API to execute code in several programming languages. It passes the source code with the relevant language ID to the Judge0 endpoint, periodically queries for the execution

outcome, and deals with various status responses like "Accepted" or "Compilation Error" in a way that supports excellent error handling and feedback.

Source code is taken as a string and the programming language as input. The output is either the standard output (stdout) or a formatted error message from the execution. The module checks if the JUDGE0_API_KEY is configured and handles API request exceptions and non-successful execution statuses from Judge0, returning descriptive error messages.

4.3 Testing Strategy

A multi-layered testing strategy was designed to ensure the quality and robustness of the application. The recommended tools for executing this strategy are Pytest, Selenium, Coverage.py, and Locust.

4.3.1 Unit Testing

Each function in app.py is tested in isolation to verify its correctness. The get_file_details() method was thoroughly tested with a range of different types of files—that is, text, image, PDF, binary, empty, and corrupted files—to validate that it correctly retrieves content and handles errors gracefully. The execute_python_code() method was tested with various cases like code that prints output, code that does not print any output, and code that raises exceptions. Assertions were also added to ensure the restrictive execution environment properly prevents unauthorized access to sensitive modules such as os or sys. In the execute_with_judge0() function, mocking was used through unittest.mock to mimic different responses from the Judge0 API. This facilitated complete testing of different scenarios—successful runs, compilation issues, runtime errors, and API timeouts—without having to make actual network calls. Pytest for writing and running tests; Coverage.py to measure test coverage.

4.3.2 Integration Testing

Focused on verifying the interactions between different parts of the system. Full end-to-end testing was performed for API interaction, including actual requests to the Gemini and Judge0 APIs. The tests checked that prompts were in correct format, sent correctly, and that responses were correctly parsed and rendered in the application. Streamlit's state handling was also fully tested to ensure that filling out the form correctly updates st.session_state. Tests also verified that st.rerun() is invoked at appropriate points to update the user interface and ensure a smooth, responsive user experience.

4.3.3 User Interface (UI) Testing

Automated browser testing was used to simulate real user interactions and validate the frontend's behavior and appearance. The flow of interaction was extensively tested using scripts that mimic normal user interaction, such as entering messages, uploading files, choosing a language from the dropdown list, submitting the form, and applying the "Execute" and "Clear Chat" buttons. The responsive design of the app was also tested to guarantee the user interface follows CSS flexbox and media query concepts set in index.html. Also, error handling was tested to ensure that any execution or API errors are accurately and clearly presented within the chat interface so that users get helpful feedback in case something goes wrong. Selenium was used for browser automation.

4.3.4 Performance Testing

Assessed the application's responsiveness and stability under load. Performance testing covered testing the response time from when a prompt is issued until the initial chunk of the AI's response is available, as well as the overall time needed for the full generation. The execution time was also tested by timing the whole round-trip for code runtime via the Judge0 API. Further, load testing was conducted by emulating concurrent multiple users to reveal any performance limitations and determine if the system can efficiently manage high traffic situations. Locust for load testing.

4.3.5 Security Testing

Focused on identifying and mitigating potential security vulnerabilities. Security testing entailed a number of critical steps to validate the application's strength. Input sanitization was tested by trying to inject malicious scripts into chat input, attempting to test for possible Cross-Site Scripting (XSS) issues. API key security was also validated by ensuring that all API keys are loaded securely from environment variables and are never sent in the client-side code. In addition, the execute_python_code function was thoroughly tested to guarantee that its execution sandbox could not be compromised, essentially stopping arbitrary system commands from running and protecting against illicit access.

4.4 Deployment Considerations

I. **Hosting:** The application is designed for deployment on cloud platforms like **Streamlit Community Cloud**, **Heroku**, or **AWS Elastic Beanstalk**, which provide managed environments for Python applications.

II. **Scalability:** The architecture, being reliant on scalable external APIs (Gemini and Judge0), is inherently scalable. The main bottleneck would be the processing capacity of the web server instance, which can be scaled horizontally on most cloud platforms.
III. **CI/CD:** A continuous integration/deployment pipeline using **GitHub Actions** would be set up to automate the running of all test suites upon each push to the repository, ensuring that only quality, tested code is deployed to production.

CHAPTER-V

DEPLOYMENT

This chapter outlines the comprehensive deployment strategy for the AI Coding Assistant, a sophisticated web application designed to aid developers with code generation, explanation, and execution. The objective is to provide a robust, clear, and actionable guide for seamless implementation, effective user training, and thorough system understanding, ensuring the successful integration of the AI Coding Assistant into a production environment.

5.1 Detailed List of Deployment Tasks

A systematic deployment plan must be used to provide the stability, security, and performance of the AI Coding Assistant. The activities fall under five main stages:

5.1.1 Prerequisite Software Installation

I. **Python 3.8+ Installation:** Install Python 3.8 or greater as the base requirement for running the Streamlit framework and certain of the extra dependencies used within the application. To install, users may either download the appropriate installer from the official website or use a package manager—such as apt on Linux, brew on macOS, or choco on Windows. After installation, the installation may be verified by running the python3 --version command in the terminal, which will display the version of Python installed.

II. **pip Package Manager**: The second thing is to ensure that pip, Python package installer, is updated since it is required to install all project dependencies. To do that, users must run the commands python3 -m ensurepip --upgrade and then python3 -m pip install --upgrade pip. The aforementioned commands will update or install pip to its latest state. The setup can be tested by executing pip --version in the command prompt, which should confirm that pip is installed and updated correctly.

III. **Virtual Environment Installation**: The second action is to create and activate a particular Python virtual environment, which is essential in order to isolate project-specific dependencies from the system-wide Python installation. Isolation avoids version conflicts and allows a reliable development environment. To activate it, run python3 -m venv coding_assistant_env to install the environment. Then, activate it using source coding_assistant_env/bin/activate on Linux or macOS

or .\coding_assistant_env\Scripts\activate on Windows. Successful activation is verified by the terminal prompt indicating the (coding_assistant_env) prefix.

IV. **Required Dependencies Installation**: The next step is to install all the required Python packages stated in the requirements.txt file because they are essential to the working of the application, such as base libraries such as streamlit, google-generativeai, python-dotenv, and requests. While the virtual environment has been activated, the installation can be performed by running pip install -r requirements.txt. A successful installation will complete without outputting any error messages, indicating that all dependencies have been installed properly and are now ready for use.

5.1.2 API Configuration

The procedure is obtaining an API key from Google AI Studio to access the Gemini API, which will be implemented in providing code generation and explanation features in the program. To obtain the key, the users are to utilize the official Google AI Studio documentation in generating API credentials. It is not advisable to hard code the API key within the application code for security reasons; instead, it should be securely stored in environment variables or a configuration management plan in order to prevent exposure and exploitation.

This is an exercise of procuring an API key from the RapidAPI hub for the Judge0 CE API to enable multi-language code execution in the app. Users need to subscribe to the Judge0 CE API through the RapidAPI platform and get their personal API key. Similar to the case of the Gemini API, it is necessary to practice secure methods by not hardcoding the key directly into the application. The key would be kept in environment variables or a secure configuration file to ensure security and prevent unauthorized access.

Then comes creating a .env file in the root directory of the project. This one is employed to store securely environment variables like API keys, keeping sensitive data away from the source code and version control systems. To install it, create a file called .env and insert the following lines, replacing the placeholders with your real API keys: GOOGLE_API_KEY="YOUR_GEMINI_API_KEY"

and JUDGE0_API_KEY="YOUR_JUDGE0_RAPIDAPI_KEY". For security reasons, it is crucial to add .env to your .gitignore file so that it will not be committed to a public repository by mistake and your credentials will not get exposed.

5.1.3 Application Setup

This phase focuses on preparing the application codebase for deployment.The operation requires fetching the application source code from the version system, which holds the full architecture, logic, and design for executing the application. This requires running the command git clone <repository_url>, replacing the placeholder with the repository link. After the cloning process is finished, ensure that the entire project files and directories have been successfully downloaded and are accessible in the cloned folder to confirm that the setup is ready for further development or deployment.

This step is concerned with installing all the necessary Python packages in the virtual environment. Even following the first setup, this step should be carried out in order to keep things consistent—particularly if new dependencies have been introduced or the environment has been re-built. Inside the activated virtual environment, execute the command pip install -r requirements.txt to install all the necessary packages as specified in the requirements file. This guarantees the application contains all the dependencies required to operate properly.

The task is to make sure that the application loads API keys from the .env file securely and reliably for access to external services. The work is assigned to the python-dotenv library. To make sure everything works as expected, ensure load_dotenv() is called at the beginning of app.py, and API keys are retrieved using os.getenv().To test, you can add a temporary line like print(os.getenv('GOOGLE_API_KEY')) in app.py to verify that the correct value is being loaded. Make sure to comment out the line before the actual deployment to keep things secure and avoid exposing sensitive information by mistake.

The task is to perform the initial test to confirm that the application is able to establish successful connections with both the Gemini and Judge0 APIs. This is important in identifying connectivity issues or broken API keys early on. To do this, add bare minimum test functions in app.py or another script making simple API calls to both services and confirm a successful return. Having these functions prepared, run them and watch out for the success messages indicating that the connections are successful.This serves to validate that the APIs are accessible and keys are correctly set prior to executing further.

5.1.4 Streamlit Deployment

This phase covers the actual deployment of the Streamlit application.

I. **Local Deployment Testing**: The objective is to execute the application locally to ensure that all functionalities work well in a close-to-production setup. This is important to determine any configuration flaws or integration problems prior to sending the application to the cloud. After enabling the virtual environment, move into the project folder and use the command streamlit run app.py. After starting the application, open it from the given local URL (usually http://localhost:8501) and exhaustively test all major functionalities such as the chat interface, code generation, file upload, and code execution to see if everything is functioning as expected.

II. **Cloud Platform Deployment Options (Heroku, AWS, Google Cloud)**: It requires choosing an appropriate cloud platform and getting the application ready for deployment, which is important to secure scalability, dependability, and widespread access. There are various alternatives to choose from, and each of them supports varying project requirements. For Heroku, an easy and newbie-friendly option, write a Procfile with the following content web: streamlit run app.py, set the Python version in a runtime.txt file (python-3.x.x), and set environment variables using the Heroku dashboard. For AWS, provision an EC2 instance yourself or use Elastic Beanstalk for simpler deployment and management. This entails installing software, copying the code, and executing the Streamlit application. On Google Cloud, the same deployment strategies hold—either configure an app.yaml file for App Engine or deploy a VM instance on Compute Engine. Heroku has simplicity and is good for small projects, but AWS and Google Cloud are more controllable and suitable for large, scalable deployments.

III. **Configure Runtime Environment**: The job is to configure the cloud environment such that the Streamlit application is running properly and open all the time. It is configured to automatically run the app and keep it open without any user intervention. Heroku does this using the Procfile, where the command to open the app is defined. For virtual machines (VMs) such as those in AWS EC2 or Google Compute Engine, using a process manager like systemd or supervisor to run streamlit run app.py as a background process is recommended. Also, set the firewall or incoming rules to allow opening traffic on port 8501, the default port Streamlit uses, to allow the application to be accessed through the web.

IV. **Establish Continuous Integration/Deployment (CI/CD) Pipeline:** The task is to integrate automated build, test, and deployment workflows of the application that help in automating development, eliminating human errors, and providing consistent and reliable deployments. This is achievable with automation tools such as GitHub Actions, GitLab CI/CD, Jenkins, or AWS CodePipeline. These can be utilized to configure triggers for performing some activities—like running tests or deploying to a cloud environment—any time changes are pushed into the repository. As an example, using GitHub Actions, you can have a .github/workflows/deploy.yml file configure how to deploy an application and run tests, so you can have a completely automated and efficient deployment pipeline.

5.1.5 System Validation

Post-deployment verification is essential to ensure the operational readiness of the system and its quality compliance. Work involves executing a full set of test cases so that everything of the application works correctly in the deployed system. It is a significant step to ensure that all essential parts—such as chat interface, file upload, code generation, and execution on various programming languages—are running smoothly and integrated as desired. To achieve this, there needs to be an end-to-end test plan developed with all the user stories and the use cases covered. Automated as well as manual end-to-end tests must be run for a thorough coverage. Verification is deemed successful if all the tests pass and the application is performing as anticipated.

The test is to quantify the responsiveness and utilization of the application under varying levels of load to identify any possible areas of bottlenecks in performance and ensure that it meets given performance targets. It can be done using commands like Apache JMeter, Locust, or even custom Python scripts to create emulation of many different levels of concurrent users and track response times. During running such simulations, system monitoring of CPU, memory, and network usage should be noted. API call latency, code execution time, and page load times in general must form part of key performance indicators, providing clear insight into how well the application scales and performs under stress.

The task is to detect and mitigate potential security loopholes to secure sensitive data—such as API keys and user input—and prevent them from getting accessed by unauthorized parties. This involves carrying out extensive penetration testing, employing automated security scanner tools like OWASP ZAP to use for web attacks

and Bandit to scan Python code, and carefully examining the access controls of the app. Special care should be exercised to deal with secure API key management and robust input validation to avoid common attack vectors. All the vulnerabilities that are detected through such tests should be addressed in advance enough to maintain the application secure and threat-proof.

5.2 Staff Training Requirements

Well-trained staff is needed to empower the efficient use, troubleshooting, and support of the AI Coding Assistant. Training must include technical areas, system-specific tasks, and documentation.

5.2.1 Technical Training

This training is aimed at technical support staff and developers who may need to extend or debug the application.The target audience for this material is relatively inexperienced staff members in Python. The content centers on basic programming principles, including elementary syntax, data structures (e.g., lists and dictionaries), control flow statements (such as loops and conditionals), functions, and basic object-oriented principles. The main goal is to provide the audience with a strong grasp of the central application logic so that they can more confidently navigate the codebase and assist in development or debugging work.

The content audience consists of developers and support personnel who work with the front end of the application. The content covers basic topics in Streamlit, such as its native components, state management through bottom-level st.session_state, layout configuration, and bottom-line best practices for deployment. The bottom line is to demystify the UI/UX framework so that the audience can make little tweaks, improve the user interface, or properly diagnose and solve UI issues as and when they occur.

This material is for technical personnel tasked with the maintenance of the application and the deployment of future upgrades. It discusses integrating the Gemini API and Judge0 Rapid API and how requests are built and responses processed within the application. It also covers error-handling processes for external API calls to provide reliable and fault-tolerant interactions. The aim is to empower the technical team with the know-how necessary to debug API-related problems efficiently and to be able to integrate new external services confidently as the application grows.

This content is aimed at technical staff and power users who wish to get further insight into the underlying operation of the AI assistant. It explores the ideas that lie behind large language models used for code generation, showing how such models take

prompts and generate responses. It also discusses significant security considerations involved in executing code, comparing sandboxed environments with third-party services like Judge0. In addition, it explains how Judge0 handles different programming languages using language IDs and how it interprets its output formats. The objective is to provide a comprehensive understanding of the potential, architecture, and limitations of the AI helper for effective utilization and further development.

5.2.2 System-Specific Training

This training is centered around the practical application and usage of the AI Coding Assistant.

This content is for the use of all users and support staff, with a live demonstration of the entire feature set of the application. This guides the user through the chat interface, language selection drop-down menu, file upload option, and code execution button, providing realistic guidance on each of the components involved. The aim is to familiarize the users with the interface and fully aware of the features of the application so that they can use it effectively and confidently in day-to-day life.

This is geared toward developers, engineers, and technical writers, with focus on best practices in writing effective prompts to maximize code generation from the AI assistant. It explains how to recognize and take advantage of diverse coding styles and how to decipher the explanations of code generated by the AI. The objective is to allow users to attain maximum efficiency and productivity with the AI for code-related tasks, such that they receive accurate, readable, and context-specific results.

This material is geared towards developers, QA engineers, and support personnel, and addresses how the AI assistant delivers explanations of code. It instructs users on how to use these explanations to debug and enhance the resulting code, identifying typical patterns and shapes in the responses of the AI. The aim is to enable the user to interpret, verify, and enhance the output of the AI effectively, facilitating more precise troubleshooting and improved code quality.

This document is for end-users and support personnel, including guidance on how to correct common application issues such as missing API keys, execution timeouts, and abnormal code behavior. It outlines sequential processes for initial diagnosis, such as testing environment variable setups, API response investigation, and checking input format. It also outlines correct escalation procedures for issues not resolved by the initial diagnosis. The objective is to make it possible for first-line support to identify

and solve frequent problems early on, with the least possible interruption of the users and successful issue resolution.

5.2.3 Documentation Review

All personnel that work with the AI Coding Assistant must read the system manual. This content is to be disseminated across all personnel and includes a careful reading of the system manual in its entirety, with emphasis on the sections most relevant to the reader's role. The purpose is to foster a general understanding of the overall goals, format, and operational procedures of the system. By letting each member of the team know how the system operates and how their particular job fits into the big picture, the company is in a better position to encourage more cooperation, smoother operations, and enhanced support between departments.

This content is aimed at the operations and deployment teams and is a detailed step-by-step walkthrough of the deployment checklist and scripts. It captures all the steps required to successfully deploy and upgrade the application, from configuring the environment, installing dependencies, to API key setup and running automated workflows. The goal is to have subsequent deployments and upgrades be smooth and reliable, with little downtime, and to remove the risk of errors occurring during the release process.

The API Key Guidelines are intended for developers and security, providing an in-depth tutorial of best practice secure handling of API keys, including keeping the keys in environment variables and keeping them out of version control. The content also covers the necessity for adhering to rate limits set by the Gemini and Judge0 APIs in order to avoid service disruptions and unexpected billing. Additionally, it covers data handling compliance needs and usage of third-party APIs. The objective is to ensure that API interactions are cost-effective, secure, and in line with organizational policies and external service contracts.

5.3 System Manual Documentation

An exhaustive manual of the system is essential for maintenance, support, and future enhancement.

5.3.1 Introduction

- **Overview of the Project**: The AI Coding Assistant is designed to optimize developer productivity by leveraging artificial intelligence in code generation, explanation, and execution across various languages. Its main purpose is to ease the process of development by providing real-time support, allowing developers to write code more effectively and quickly. This helper specifically aims at

problems such as reducing time spent writing tedious boilerplate code, accelerating the learning curve for new programming concepts or languages, and simplifying debugging. By solving these problems, the helper helps developers solve challenging problems with the AI handling mundane coding tasks.

- **Main Features and Functions**: The AI Coding Assistant has an interactive chat interface that enables users to naturally chat with the system. It also comes with AI-driven code generation functionality that supports multiple programming languages, enabling developers to generate functional pieces of code across different scenarios. The assistant further provides context-specific explanations of code, enabling users to understand confusing logic or alien syntax. The users can upload files that contain code to be analyzed, and the assistant can make suggestions or provide insights based on the content. Apart from this, the assistant also makes use of the Judge0 API to enable real-time, multi-language code execution for the interface. Apart from enhancing accuracy and relevance, it also offers a language selection feature to allow the user to identify the programming language context of the query.
- **Target User Groups**: The AI Coding Assistant is designed to benefit a wide range of users, from beginners to experienced software developers. It can also be an educational study aid for students studying programming as it gives explanations and real-time code support. Technical writers can use the assistant to create correct and clear code documentation, making it easier to write tutorial material. In addition to that, QA engineers can leverage the tool to quickly develop test scripts, taking time off from routine testing and enhancing efficiency during the software development process.

5.3.2 Technical Specifications

The design of the AI Coding Assistant is based on a Streamlit frontend that allows for an interactive and user-friendly web-based interface. Behind the scenes, the Streamlit server automatically takes care of user input, request handling, and communication with external APIs. The two core APIs power the assistant's core feature: the Google Gemini API, which enables natural language interaction through large language model (LLM) capabilities, and the Judge0 Rapid API, which enables real-time execution of code in a multitude of programming languages. To comprehend the system workflow, a level architecture diagram illustrates data exchange between the

user interface, Streamlit application, and the external APIs and how the input is executed, interpreted, and replied with smart results or executable outputs.

The AI Coding Assistant employs two basic external resources: the Gemini API and the Judge0 Rapid API. The Gemini API, provided by Google, supports text and code generation through endpoints that consume structured JSON payloads for system messages, prompts, and user text input. The Gemini API provides generated results such as explanations, code snippets, or language-specific textual output as structured output. To run code, the assistant uses the Judge0 Rapid API, where posting code involves submitting code through a submission endpoint with the correct language ID and standard input. Post submission, a token is returned, which is then used to poll the result retrieval endpoint for receiving the output or error messages. To provide a secure access, both APIs need to be authenticated: the Gemini API by a GOOGLE_API_KEY, and the Judge0 API by a JUDGE0_API_KEY, which both need to be supplied in the request parameters or headers depending on the specifications of each API.

The AI Coding Assistant is designed to deliver high-performance capability with expected response times for code generation usually ranging from 2 to 5 seconds, depending on the complexity of the prompt and API latency. Code execution via the Judge0 Rapid API usually takes between 1 and 3 seconds for most of the languages that are supported as well as average inputs. As for throughput, it can handle multiple concurrent requests, but depending on the network traffic and rate limiting of the APIs, the performance may differ. In light loads, there is low resource usage, with Streamlit server using a balanced amount of CPU and memory—usually around 200–500 MB of RAM and periodic spikes in the CPU, which does not slow down the host environment.

Streamlit applications tend to scale by having numerous instances of the application sit behind a load balancer, which sends user traffic evenly to provide responsiveness and reliability. This approach gives the system the capacity to handle increased demand by horizontally scaling resources when needed. To handle external API rate limits—like those of the Gemini and Judge0 APIs—mechanisms such as retry and exponential backoff are implemented. These avoid short-term rate limit failures from leading to user disruption and allow the system to recover seamlessly from request throttling. Additionally, when deployed on cloud platforms like AWS, GCP, or Azure, platform-specific scaling features such as auto-scaling groups, serverless containers, and managed load balancers can be utilized by developers to scale resources automatically on real-time utilization and traffic patterns.

5.3.3 Installation Guide

To install the AI Coding Assistant, go through a step-by-step installation process starting with cloning the project repository using Git. Before that, have installed all the prerequisite software on your system like Python (version 3.8 or later), pip for package management, and virtualenv for creating isolated environments. After setting them, navigate to the project directory and install a virtual environment using virtualenv venv and activate it with source venv/bin/activate on Unix or venv\Scripts\activate on Windows. With the virtual environment activated, install all the Python packages needed using pip install -r requirements.txt, which will download and install the necessary packages for use in the application. Once you are done with these steps, you can start the Streamlit app via the command streamlit run and open the interactive web interface of the AI Coding Assistant.

Explanation of requirements.txt and how to update it (pip freeze > requirements.txt).

In order to safely store API credentials, add a .env file in the project root. This file must contain environment variables like GOOGLE_API_KEY and JUDGE0_API_KEY with each key-value pair on a new line in the format KEY=your_api_key_here. These variables are then automatically imported by the app to authenticate Judge0 and Gemini API requests. Never commit the .env file to public or shared version control systems such as Git since it holds sensitive information. Avoid accidental exposure by having .env added to the .gitignore file, protecting your API keys from being pushed into public or shared repositories.

5.3.4 User Guide

I. **Feature Exploration**: **Chat Interface**: How to type messages, read responses. **Code Display**: How generated code is presented. **Code Explanation**: How to find and interpret the AI's explanation.

II. **Code Generation Workflow**: Tips for writing clear and specific prompts. Examples of successful code generation requests. How to iterate on generated code.

III. **Language Selection Process**: How to use the dropdown to set the language context. Impact of language selection on AI's output. Understanding "Auto-detect" behavior.

IV. **File Upload Mechanism**: How to attach files using the "📎 Attach files" button. Supported file types and their handling (text, code, images, PDFs). How the AI uses file content for context. How to remove staged files.

V. **Code Execution**: How to click the "Execute Code" button. Understanding execution output (success, errors, compilation issues). Limitations of the execution environment (e.g., internet access, specific libraries).

5.3.5 Troubleshooting

I. **Common Error Resolution**: **"Failed to initialize Gemini model"**: Check GOOGLE_API_KEY in .env, network connectivity to Google API. **"API request failed (Judge0)"**: Check JUDGE0_API_KEY, network connectivity to Judge0 API, Judge0 service status. **"Execution error"**: Review generated code for syntax errors, logical issues, or unsupported operations in the execution environment. **Streamlit UI issues**: Clear browser cache, restart Streamlit server.

II. **API Connection Issues**: Steps to verify API keys. Checking network firewalls and proxy settings. Consulting API documentation for service status.

III. **Performance Optimization Tips**: Reduce reliance on excessively large file uploads. Optimize prompts for conciseness. Monitor resource usage on the deployment platform.

IV. **Maintaining the Application**: Regular dependency updates (pip install --upgrade -r requirements.txt). Monitoring logs for errors and unusual activity. Planning for AI model updates and API version changes.

5.4 Automatic Implementation and Deployment Scripts

Section 1: Automated Deployment Script for AI Coding Assistant

```
#!/bin/bash
# Automated Deployment Script for AI Coding Assistant

echo "Starting automated deployment for AI Coding Assistant..."

# --- 1. Virtual Environment Setup ---
echo "Setting up virtual environment 'coding_assistant_env'..."
python3 -m venv coding_assistant_env

# Check if activation script exists and activate
if [ -f "coding_assistant_env/bin/activate" ]; then
    source coding_assistant_env/bin/activate
    echo "Virtual environment activated (Linux/macOS)."
elif [ -f "coding_assistant_env/Scripts/activate" ]; then
    source coding_assistant_env/Scripts/activate
    echo "Virtual environment activated (Windows compatible)."
else
    echo "Error: Could not find virtual environment activation script."
    exit 1
fi
```

Section 2: Dependency Installation

```
# Upgrade pip within the virtual environment
echo "Upgrading pip..."
pip install --upgrade pip

# --- 2. Dependency Installation ---
echo "Installing project dependencies from requirements.txt..."
if [ -f "requirements.txt" ]; then
    pip install -r requirements.txt
    if [ $? -ne 0 ]; then
        echo "Error: Failed to install dependencies. Please check requirements.txt"
        exit 1
    fi
else
    echo "Error: requirements.txt not found. Please ensure it exists in the project root."
    exit 1
fi
echo "Dependencies installed successfully."
```

Section 3: Environment Configuration and Application Launch

```
# --- 3. Environment Configuration ---
echo "Configuring environment variables..."
if [ -f ".env.example" ]; then
    if [ ! -f ".env" ]; then
        cp .env.example .env
        echo "   Created .env file from .env.example. IMPORTANT: Manually update API keys in"
        echo "   Open .env and replace YOUR_GEMINI_API_KEY and YOUR_JUDGE_RAPIDAPI_KEY."
    else
        echo ".env file already exists. Skipping copy. Ensure API keys are updated."
    fi
else
    echo "Warning: .env.example not found. Please create .env manually with your API keys."
fi

# --- 4. Streamlit Application Launch (for testing/local deployment) ---
echo "Launching Streamlit application (for local testing/development)..."
echo "Access at http://localhost:8501 (or as provided by Streamlit after launch)."
streamlit run app.py

echo "Deployment script finished."
echo "For cloud deployment, refer to the 'Cloud Platform Deployment Options' in Chapter 5."
```

Figure 8: Automatic Implementation and Deployment Scripts

In Figure 8 Save the script above as deploy.sh in your project's root directory. Make Executable: chmod +x deploy.sh Run: ./deploy.sh Manual Step: After the script runs, it will prompt you to manually update the .env file with your actual GOOGLE_API_KEY and JUDGE0_API_KEY. This is a critical security step and cannot be automated securely within a public script. Cloud Deployment: This script is primarily for local setup and testing. For cloud deployment, follow the specific instructions for your chosen platform (Heroku, AWS, Google Cloud) as outlined in Chapter 5.1.4.

5.5 Deployment Considerations for the AI Coding Assistant

Beyond the checklist and training, several critical considerations impact the long-term success and maintainability of the AI Coding Assistant.

5.5.1 Implement Robust Error Handling

Ensure the Streamlit application gracefully handles and displays user-friendly error messages instead of crashing. This includes errors from API calls (e.g., rate limits, invalid keys), file processing, and code execution. Implement comprehensive logging (e.g., using Python's logging module) to capture application events, warnings, and errors. Logs are invaluable for debugging and monitoring. Store logs securely and review them regularly. Use try-except blocks extensively around API calls, file operations, and code execution logic in app.py to catch exceptions and provide informative feedback.

5.5.2 Ensure Secure API Key Management

Storing API keys in .env files and accessing them via os.getenv() is the minimum secure practice. Never commit these files to version control. For production deployments, consider using cloud-native secrets management services (e.g., AWS Secrets Manager, Google Secret Manager, Azure Key Vault) to store and retrieve API keys securely, adding an extra layer of protection.Ensure that the deployed application's service account or IAM role has only the necessary permissions to access required resources and APIs.

5.5.3 Provide Comprehensive Logging

Implement structured logging (e.g., JSON format) to make logs easier to parse and analyze with log management tools. Define clear policies for how long logs are retained and where they are stored, balancing debugging needs with storage costs and compliance. Integrate logs with monitoring systems (e.g., Prometheus, Grafana, CloudWatch, Stackdriver) to create dashboards and set up alerts for critical errors or unusual activity (e.g., high error rates, unusual API usage patterns).

5.5.4 Design Scalable Architecture

Plan for horizontal scaling of the Streamlit application by deploying multiple instances behind a load balancer. This distributes incoming traffic and improves fault tolerance. Design the application to be as stateless as possible, especially concerning user sessions. While st.session_state manages per-user state, ensure that critical data can be persisted externally if session continuity across different instances is required. Be aware of the rate limits imposed by Gemini and Judge0 APIs. Implement retry mechanisms with exponential backoff for transient errors. For high-volume usage, explore higher-tier API plans if available.

5.5.5 Create Backup and Recovery Mechanisms

Ensure the project repository is hosted on a reliable platform (e.g., GitHub, GitLab, Bitbucket) with appropriate backup strategies. Regularly back up configuration files, including the .env file (stored securely, not with source code) and any cloud-specific configurations (e.g., app.yaml, Procfile). Develop a disaster recovery plan outlining steps to restore the application in case of outages, data corruption, or other catastrophic events. This includes restoring from backups and redeploying.

5.6 Critical Warnings

Adhering to these warnings is paramount for the security and stability of the AI Coding Assistant.

I. **NEVER expose API keys publicly**: This includes hardcoding them in client-side code, checking them into public repositories, or exposing them in public-facing logs. Always use environment variables or dedicated secrets management services.

II. **Implement strict input validation**: Sanitize and validate all user inputs to prevent injection attacks (e.g., prompt injection, code injection attempts) and other vulnerabilities. This is especially crucial given the AI's ability to generate and execute code.

III. **Regularly update dependencies**: Keep all Python packages and system libraries up-to-date to patch security vulnerabilities and benefit from performance improvements. Use pip list --outdated and update regularly.

IV. **Conduct periodic security audits**: Regularly review the application's code, configurations, and deployment environment for security weaknesses. This can include automated scans and manual penetration testing.

CHAPTER-VI
CONCLUSION AND FUTURE WORK

6.1 Project Overview

This project aimed to develop a sophisticated AI-based software development assistant with the purpose of automating and making the contemporary coding process more efficient. The central goal was to develop a comprehensive platform that combines code generation, debugging, explanation, and execution under one single interactive setting. Using an advanced technology stack, we created a tool that acts as a smart collaborator for developers, teachers, and learners alike. The application, hosted on the Streamlit platform, offers a smooth user interface where users can upload existing codebases, communicate with an AI model through natural language, and be provided with executable, well-documented code in an array of programming languages. This effort is one of the important strides toward an even more intuitive and productive paradigm of software development that minimizes the cognitive burden on the programmer and directs it to higher-level thinking and creativity.

6.2 Technological Integration Achievements

The success of this project will depend on the strong and strategic integration of multiple state-of-the-art technologies,each serving a crucial function in the operation of the platform.

6.2.1 Gemini API

Our application is founded on Google's Gemini API, which offers the fundamental intelligence behind our set of features. Its sophisticated capabilities in natural language processing and code comprehension allow the platform to precisely understand user requests for creating new code, debugging difficult errors, and offering precise, concise explanations of complex code segments. The effective incorporation of the Gemini API is the foundation of the functionality of the tool as a general and savvy assistant.

6.2.2 Judge0 Rapid API

In order to offer an engaging and interactive experience, we have incorporated the Judge0 Rapid API, a high-performance and secure code execution platform. This ensures that users are able to execute the AI-generated code directly within the app for an incredibly broad range of programming languages. This real-time feedback mechanism is essential for iterative development and learning, making the tool no longer a static code generator but a live coding platform.

6.2.3 Streamlit Deployment Platform

The entire application is built upon the Streamlit framework, which proved to be an excellent choice for rapid prototyping and deployment of a data-centric and interactive web application. Streamlit's architecture enabled the creation of a clean, responsive, and user-friendly interface, complete with essential features like file upload functionality and real-time updates to the chat interface, all with a remarkably efficient development cycle.

6.2.3 Multilingual Programming Language Support

A key achievement is the platform's extensive support for a diverse range of programming languages. This was made possible by the combined power of the Gemini API's vast training data and the language-agnostic nature of the Judge0 execution engine. This feature makes the tool highly versatile and applicable to a wide audience of developers with varying technical backgrounds.

6.3 Key Technical Contributions

This project makes several unique contributions to the field of AI-assisted software development tools:

6.3.1 Unified Development Lifecycle Integration

The platform's most significant innovation is the creation of a cohesive and end-to-end development lifecycle within a single interface. Users can seamlessly transition from conceptualization (describing a feature in natural language) to implementation (AI-generated code), verification (immediate execution), and comprehension (code explanation). This unified workflow drastically reduces context-switching and streamlines the development process.

6.3.2 Context-Aware AI Interaction

We introduced an advanced context-passing mechanism that improves the performance of the AI. With the facility to upload files and define a language context, the Gemini model is provided with a richer, more complete prompt. This results in outputs that are not only more precise but also customized according to the user's particular project requirements and programming style.

6.3.3 Prompt Engineering for Expert Outputs

One significant technical contribution is our systematic method of prompt engineering. Through the creation of exact instruction templates, like the ---EXPLANATION--- delimiter, we can consistently produce structured outputs from the Gemini API. This guarantees that the user obtains the code along with a step-by-step explanation in a consistent and easy-to-understand manner, increasing the utility of the tool as an instructional aid.

6.4 Limitations and Challenges

Although the project is a success, it is crucial that its limitations and challenges in development are recognized.

6.4.1 API Dependencies and Constraints

The platform's functionality relies heavily on the existence and performance of the Gemini and Judge0 APIs. This presents potential issues with API latency, rate limits, and cost control at scale. Downtime or alterations to these external services would translate directly to the user experience.

6.4.2 Limitations ofthe ExecutionEnvironment

Although Judge0 offers a safe sandboxed environment, it's not a complete development environment. The platform is still not able to execute projects with complicated dependencies, many file imports, or network connections. This restricts its usability for testing and debugging larger, realistic applications.

6.4.3 Streamlit State Management

The management of the application state, more specifically, the chat history and output of different code execution blocks, was a challenge in Streamlit's script-rerun execution model. Although we have a sound session state management system implemented, its scalability and avoiding race conditions demand careful consideration.

6.5 Future Work and Research Directions

The current platform serves as a strong foundation for numerous future enhancements and research opportunities that could further revolutionize AI-assisted development.

Future iterations could integrate more features traditionally found in Integrated Development Environments (IDEs). This includes real-time syntax highlighting and linting within the editor, intelligent code completion suggestions powered by the AI, and more advanced debugging tools that allow for setting breakpoints and inspecting variables.

A major leap forward would be to expand the tool's context window from single files to entire project directories or even full Git repositories. This would enable the AI to perform complex, project-wide refactoring, generate code consistent with the existing codebase, and provide more holistic architectural suggestions.

We see a future in which the tool is used for real-time collaboration. Many users might work in one session, with the AI serving as an active participant—a genuine pair-programming partner that can provide recommendations, solve merge conflicts, and maintain coding standards.

There is tremendous potential in fine-tuning the underlying language model on specific, proprietary codebases. This would allow organizations to build a highly specialized AI assistant that knows their particular architectural styles, coding conventions, and internal libraries, leading to hyper-relevant and optimal code generation.

The next logical step after this project is to merge it with standard version control software like GitHub and GitLab. The AI can be programmed to generate new branches automatically, commit code, and open pull requests with detailed descriptions, making the development process more efficient.

6.6 Potential Industry and Academic Impact

This project, and the class of software that it represents, will have a profound impact on industry and academia. To the software industry, this platform offers a powerful means to accelerate development cycles, enhance code quality, and lower the threshold of entry for new developers. It can potentially be an indispensable tool for rapid prototyping, learning a new language, and automating tedious coding tasks. At a research level, this project provides a treasure trove for pedagogy in software engineering, human-computer interaction, and artificial intelligence application. It can be used to investigate how the creativity and problem-solving capacity of humans can be bettered using AI, and to develop novel methods for teaching computer programming in a more intuitive and interactive manner.

In short, this project is a triumphant example of the enormous potential of integrating advanced AI capabilities with the development of software. With the inclusion of an exhaustive, intelligent, and interactive tool, not only have we created a powerful assistant, but we have also made the foundation for a future in which human developers collaborate with artificial intelligence to reach historic levels of productivity and innovation. The road ahead is filled with fascinating potential, and we expect that the ideals and methods outlined in this book will play a key role in shaping the future of software development.

REFRENCES

[1] Google AI, “Gemini API documentation,” Google for Developers, 2024. [Online]. Available: https://ai.google.dev/docs/gemini_api_overview

[2] H. Horvat, “Judge0 CE API documentation,” 2024. [Online]. Available: https://api.judge0.com/

[3] Python Software Foundation, “Python 3.12.3 documentation,” 2024. [Online]. Available: https://docs.python.org/3/

[4] RapidAPI, “Judge0 CE on RapidAPI Marketplace,” 2024. [Online]. Available: https://rapidapi.com/judge0-official/api/judge0-ce

[5] K. Reitz, “Requests: HTTP for humans (Version 2.32.3) [Computer software],” Python Software Foundation, 2024. [Online]. Available: https://pypi.org/project/requests/

[6] J. Saurabh, “python-dotenv (Version 1.0.1) [Computer software],” Python Software Foundation, 2024. [Online]. Available: https://pypi.org/project/python-dotenv/

[7] Streamlit Inc., “Streamlit documentation (Version 1.35),” 2024. [Online]. Available: https://docs.streamlit.io/

[8] World Wide Web Consortium (W3C), “HTML5: A vocabulary and associated APIs for HTML and XML,” W3C Recommendation, Oct. 28, 2014. [Online]. Available: https://www.w3.org/TR/html5/

[9] Microsoft, “Visual Studio Code Documentation,” 2024. [Online]. Available: https://code.visualstudio.com/docs

[10] Visual Studio Code Team, “Visual Studio Code Documentation,” Microsoft, 2024. [Online]. Available: https://code.visualstudio.com/docs

[11] JetBrains, “IntelliJ IDEA documentation,” 2024. [Online]. Available: https://www.jetbrains.com/idea/documentation/

[12] Replit Inc., “Replit documentation,” 2024. [Online]. Available: https://docs.replit.com/

[13] CodePen, “CodePen documentation,” 2024. [Online]. Available: https://blog.codepen.io/documentation/

[14] GitHub, Inc., “GitHub Copilot documentation,” 2024. [Online]. Available: https://docs.github.com/en/copilot

[15] OpenAI, “OpenAI Codex overview,” 2024. [Online]. Available: https://platform.openai.com/docs/guides/code

[16] Mozilla Developer Network (MDN), “HTML: HyperText Markup Language,” 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML

[17] Mozilla Developer Network (MDN), “JavaScript Guide,” 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

[18] W3Schools, “HTML Tutorial,” 2024. [Online]. Available: https://www.w3schools.com/html/

[19] J. Nielsen, Usability engineering, Academic Press, 1994.

[20] SeleniumHQ, “Selenium Documentation,” 2024. [Online]. Available: https://www.selenium.dev/documentation

[21] Locust.io, “Locust load testing documentation,” 2024. [Online]. Available: https://docs.locust.io

[22] OWASP Foundation, “OWASP Top Ten Web Application Security Risks,” 2024. [Online]. Available: https://owasp.org/www-project-top-ten/

[23] Amazon Web Services, “Amazon CodeWhisperer documentation,” 2024. [Online]. Available: https://docs.aws.amazon.com/codewhisperer

[24] Y. Allal, S. Raghavan, A. Madaan, A. Liu, C. Singh, and G. Neubig, “CodeTrek: Flexible navigation of code repositories with language models,” arXiv preprint, arXiv:2305.06204, 2023. [Online]. Available: https://arxiv.org/abs/2305.06204

[25] T. B. Brown et al., “Language models are few-shot learners,” arXiv preprint, arXiv:2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[26] L. Reynolds and K. McDonell, “Prompt programming for large language models: Beyond the few-shot paradigm,” arXiv preprint, arXiv:2102.07350, 2021. [Online]. Available: https://arxiv.org/abs/2102.07350

[27] A. Sarkar and A. Saha, “AI for software development: A systematic literature review,” Journal of Systems and Software, vol. 192, p. 111365, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111365

[28] Y. Wang, E. Kamar, E. Horvitz, and H. Shah, “The future of human-AI collaboration in software development,” Microsoft Research, 2023. [Online]. Available: https://www.microsoft.com/en-us/research

[29] M. Zhou, A. Liu, B. Li, Z. Xu, and J. He, “DocPrompting: Generating code by retriever-augmented generation,” arXiv preprint, arXiv:2203.07814, 2022. [Online]. Available: https://arxiv.org/abs/2203.07814