Temple HCI Lab CLOVER

Table of contents:

Introduction

Features

- Code Suggestions
  - [x] Inline Code Block Suggestions
  - [x] Inline Suggestions can be Wrong
  - [x] Inline Line by Line Suggestions
- Authentication
  - [x] Sign Up in the Website
  - [x] Sign Up with the Extension
  - [x] Sign In with the Website
  - [x] Sign In with the Extension
  - [x] Sign In with Providers (GitHub)
  - [x] Sign Up with Providers (GitHub)
- Logging
  - [x] Code Suggestions Logged with Metadata About Them
  - [x] How Long the User Takes to Accept/Reject a Suggestion is Logged
  - [x] How many Correct/Incorrect Suggestions the User Accepts is Logged
  - [ ] How much the User to Actually Writing Code is Logged
- User Analytics Page
  - [x] Charts Display Data Based on Student Performance
  - [x] Instructors can View Students Performance from Their Dashboard
- User Experience
  - [x] Users Can Edit Their Profile
  - [x] Users Can Delete Their Account

- [x] Extension Hosted on VS Marketplace
- [ ] Mobile Version of the Website
- [ ] View Past Quizzes

- Education

  - [x] Students can Join their Instructor's Class
  - [x] Students are Quizzed Based on Past Suggestions
  - [ ] Learning Badges/Achievements for Students

- Performance

  - [ ] Performance Similar to GitHub Copilot
    - While close, its not as fast yet
  - [ ] Model Distillation to Use Smaller Models

- User Intervention

  - [x] User Can be Suspended
  - [x] User Can be Locked out
  - [x] User Can Review Incorrect Code

- Maintainability

  - [x] Containerized Deployment
  - [x] Automated Testing
  - [ ] Updated Documentation

Know Issues

- Some languages will not have suggestions generated for them.

*Last updated by **Nicholas Rucinski***

Requirements Specification

Document Overview goes here.

System Overview

Project Abstract

System Block Diagram

block-diagram drawio (1)

General Requirements

Users will need:

Features and Requirements

Functional Requirements

Use-case descriptions

usecase-diagram drawio

System Overview

Project Abstract

The goal of this project is to develop an AI-powered coding assistant similar to GitHub Copilot, but with a strong emphasis on education. Unlike traditional educational tools that rely on structured coding modules, the assistant uses detailed logging and mistake recognition to analyze how students code, fostering a deeper understanding of their work while preventing over-reliance on AI-generated suggestions. By tracking user interactions and offering real-time feedback, the system serves as both a coding assistant and a learning aid for novice programmers. Additionally, users can access this data through a portal to review coding habits and track progress over time. The aim is to integrate seamlessly into the IDE, ensuring minimal disruption to the coding workflow while maintaining fast response times.

Conceptual Design

This project will integrate an AI model, such as Meta's Ollama or OpenAI's ChatGPT, to deliver inline code suggestions within a Visual Studio Code extension. The extension will not only assist users in real-time but also track various statistics on how these suggestions are utilized. This data will be sent to a backend server, where it will be analyzed to identify patterns in user behavior. When necessary, the assistant will intervene with contextual feedback, helping users recognize mistakes, improve coding habits, and develop a deeper understanding of their work. Additionally, a dashboard will present these insights in an intuitive interface, allowing users to monitor their progress, review past interactions, and refine their skills over time. By combining real-time assistance with structured analytics, this system aims to create a more interactive and educational coding experience.

Background

AI-powered code assistants, such as OpenAI's ChatGPT and GitHub Copilot, have significantly transformed software development by helping programmers write code more efficiently and with fewer errors. However, these tools have also posed challenges for novice programmers. Many

beginners rely on the suggestions provided by the AI without fully grasping the underlying concepts, sometimes even accepting solutions without reading or understanding them. This over-reliance can encourage poor programming habits and hinder skill development. This project seeks to address this issue by implementing a system that still provides intelligent code suggestions but requires users to demonstrate a clear understanding of the suggested code before it is integrated into their projects. This approach aims to strike a balance between convenience and education, ensuring that users not only receive assistance but also gain a deeper comprehension of their code.

While there are code analyzers that aim to provide feeback, such as Sourcery or PyLint, they are typically not real-time and operate outside of an IDE. These tools tend to focus on writing "clean" and effecient code, but fail to explain why such practices are imporant or how users can improve their understanding. This project's approach is to integrate directly with VSCode and provides realtime feedback in order to disrupt the programmer as little as possible. With this approach we hope that AI coding assistance can still be used by newer programmers while still promoting learning and good programming habits.

Last updated by ***Nicholas Rucinski***

System Block Diagram

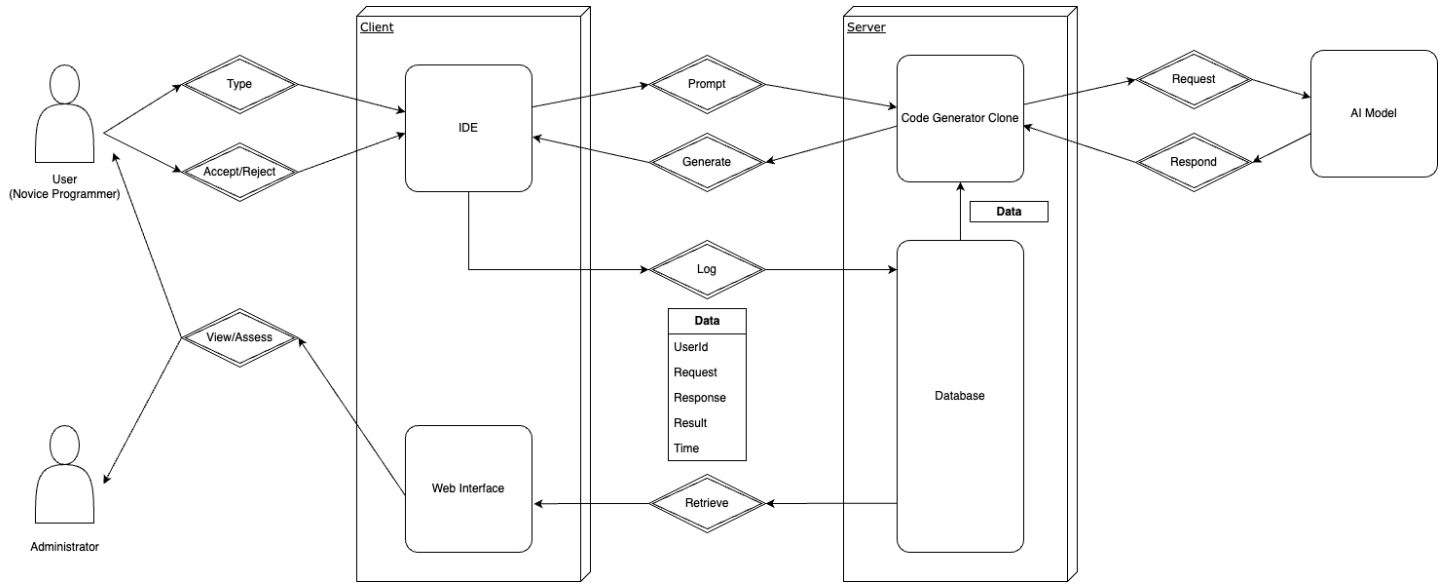


Figure 1. Architecture design of the *Github Copilot Clone* application.

Description

The system architecture is designed to function as a GitHub Copilot-style AI coding assistant, providing real-time inline code suggestions while monitoring user engagement and learning patterns. It consists of three main components: the Client, Server, and AI Model, working together to analyze user input and deliver intelligent code completions.

The Client integrates directly into the user's IDE, continuously analyzing their typing patterns to determine when to send prompts to the AI Model for inline code suggestions. Users can accept, reject, or modify these suggestions, and these interactions are logged to track their progress. A web interface is also available for administrators (typically instructors) to review user engagement, helping assess how learners interact with the AI-generated recommendations.

The Server manages request processing, and activity logging. It forwards user input to the AI Model and returns the generated suggestions. The system is designed to intentionally introduce minor errors in some suggestions, encouraging users to critically evaluate and refine their code. If a user consistently accepts incorrect suggestions, the system dynamically adjusts the response speed, slowing down code completions to encourage more thoughtful engagement. This behavior is driven

by analyzing historical interaction logs stored in the Database, which tracks user actions, accepted or rejected suggestions, and overall accuracy trends.

Last updated by ***Nicholas Rucinski***

General Requirements

Users will need:

- An internet connection
- Visual Studio IDE

Code Maintenance and Documentation:

- Git and Github for version control
- Docusaurus for documentation

*Last updated by **Nicholas Rucinski***

Features and Requirements

Functional Requirements

Logging

The system must log user interactions to gather insights on how suggestions are used. Logged data includes:

- Whether a code suggestion was accepted or rejected
- How often code suggestions are given to a user
- How long it takes for a user to accept or reject a suggestion from its generation

Code Suggestions

- The system must provide **context-aware** code suggestions based on user code
- The system must provide suggestions **inline** in the editor
- The system must be able to **generate correct and incorrect** suggestions
- The system must notify users when they accept an incorrect suggestion.

User Interaction

- The system allows users to write code as usual within the IDE.
- The system must allow users to manually **mark** a suggestion as correct or incorrect.

Limitations & Safeguards

The system should encourage **thoughtful code acceptance** by implementing safeguards:

- Users may be temporarily locked out from suggestions after **three incorrect acceptances**.
- A warning message should appear before locking a user out.
- A cooldown period should be implemented before resuming suggestions.

Nonfunctional Requirements

Performance

- The system must generate code suggestions **within 5 seconds**, comparable to GitHub Copilot.

Statistics & Insights

- A **portal/dashboard** should allow users and administrators to access logged statistics, including:
  - User acceptance/rejection rates.
  - Average response time to suggestions.

Education & Learning

- The system should promote critical reflection in programmers who are still learning

User Experience

- The system should integrate **seamlessly** into the coding workflow without unnecessary interruptions.

Maintainability

- The codebase should be **modular and well-documented** to allow easy feature additions and maintenance.

*Last updated by **Nicholas Rucinski***

Use-case descriptions

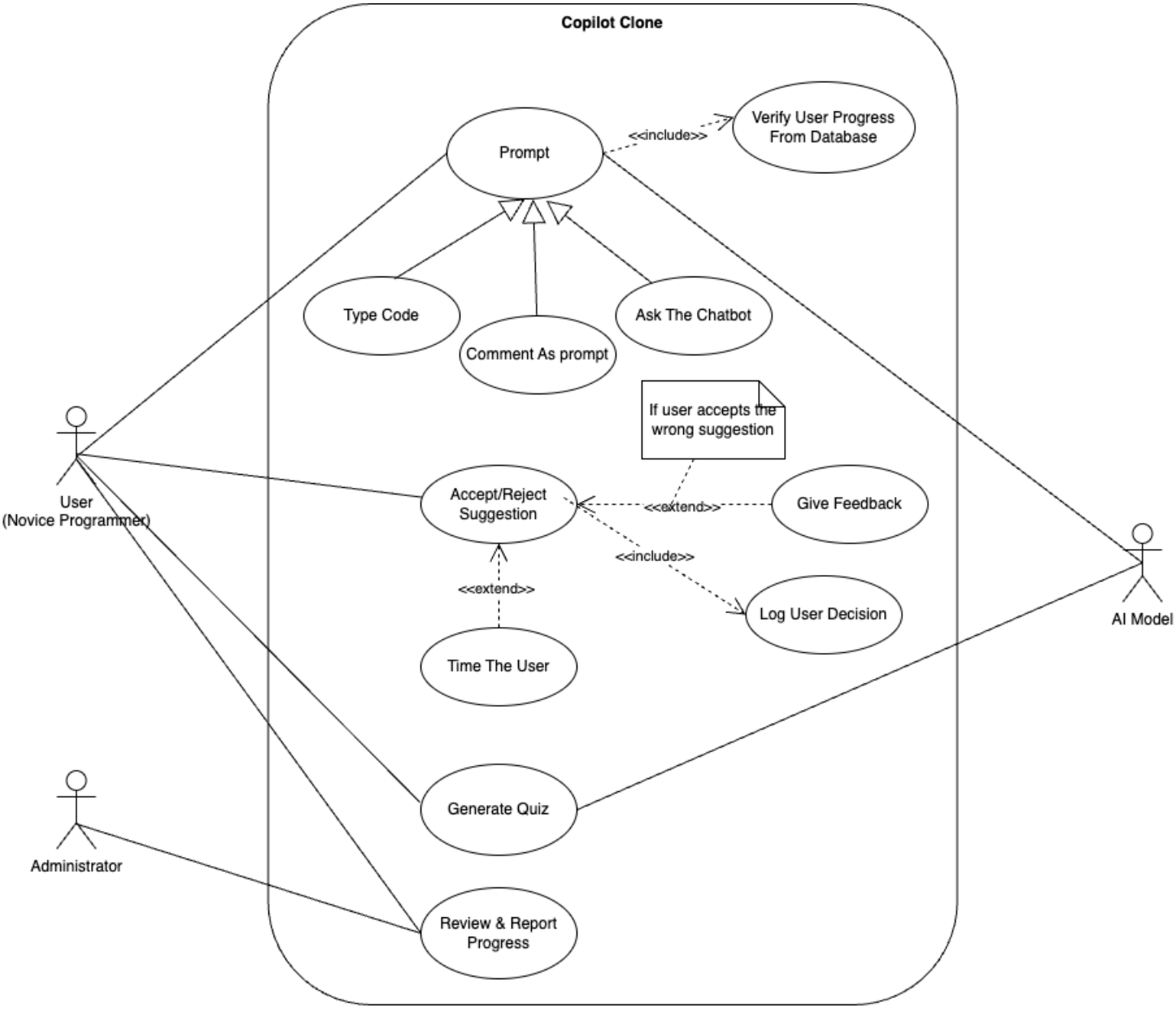


Figure 1. UML Use Case Diagram of the *Github Copilot Clone* application.

Actors

Our project has multiple actors, meaning that different use cases are needed to describe how each actor interacts with the system.

- **User (beginner programmers)**: Student learning to program, interacting with our Copilot VS Code Extension for code suggestions and asking questions about concepts.
- **AI Model**: The AI model running on our system that will create code suggestions, answer concept questions, and generate quizzes.
- **Administrator**: An instructor that wants to monitor students' progress and review logged data from the system to identify concepts that should be reviewed.

Use Cases

Use Case 1: Receiving Context-Aware Code Suggestions

Actors: User, AI Model

The user wants to get inline code suggestions to assist with a coding assignment and determine if it is correct or incorrect.

1. **User** begins typing code.
2. **AI model** analyzes the context of the code and creates a code suggestion (incorrect or correct).
3. System displays the **AI model** suggestion.
4. **User** reviews the suggestion and determines if it is correct or not.
5. **User** selects “Accept” if they think the code is right or “Reject” if they believe it is wrong.
6. System logs the user’s choice, time it took for the user to make a decision, and whether or not it was correct.
7. *If the user accepts an incorrect suggestion, the **AI model** gives an explanation of the mistake.*

Use Case 2: Asking Inline Questions about Code

Actors: User

The user wants to ask a question about a specific piece of code inline while coding.

1. **User** highlights a portion of their code.
2. **User** clicks on "ask Copilot" to type a question.
3. **AI model** provides an explanation for the question.

4. **User** reviews the response.
5. System logs the user's question and whether the user requested further clarification.

Use Case 3: Asking Questions in the Copilot chat

Actors: User, AI Model

The user wants to ask a general programming question about a coding concept.

1. **User** opens ai model chat.
2. **User** types a question (e.g. "How do for loops work in Java?")
3. **AI model** processes the question and returns an answer.
4. **User** reviews the response.
5. System logs the user's question and whether the user requested further clarification.

Use Case 4: Logging Decision Time for Code Suggestions

Actors: User, AI Model

The system has to track how long a user takes to accept or reject a suggestion to ensure the user is actually engaging with the code.

1. System begins a timer when it suggests code to the user.
2. **User** reviews the suggestion (as quickly or slowly as they want).
3. **User** accepts or rejects the suggestion.
4. System stops the timer and logs the response time.
5. *If the user consistently makes quick and random wrong selections, the system flags them for potential disengagement or lack of knowledge on the subject matter.*

Use Case 5: Receiving Feedback After Selecting a Suggestion.

Actors: User

The user wants to receive feedback on whether they correctly identified a suggestion as right or wrong.

1. **User** accepts or rejects a suggestion.

2. System determines if the user's choice was correct.
3. *If the **user** chooses correctly, system provides confirmation.*
4. *If the **user** chooses incorrectly, system provides an explanation of the mistake.*
5. System logs the mistake for admin review.

Use Case 6: Tracking and Logging User Decisions

The system has to track user engagement, correctness, and common errors.

1. System logs each accepted or rejected code suggestion.
2. System tracks if the choice was correct or incorrect.
3. System records how long the user took to decide on accepting or rejecting.
4. *If user frequently accepts incorrect suggestions, system logs recurring mistakes for admin review.*
5. *If multiple users make the same mistakes, system flags the concept as a struggle area.*

Use Case 7: Identifying Common Student Mistakes

Actors: AI Model

System identifies patterns in incorrect suggestions and concepts the user struggles with.

1. **AI Model** keeps logs of its own suggestions.
2. **AI Model** tracks key points in the suggested code that the user continuously mistakes as correct.
3. **AI Model** suggests similar code to the user that builds knowledge for the concepts that are struggled with.
4. If the incorrect suggestion is chosen again, the pattern will be updated based on the user's response.
5. New suggestions will be made that will allow the user to learn from their mistakes.

Use Case 8: Generating Learning Reports for Administrators

Actors: AI Model, Administrator

System compiles the user data into a detailed report for instructors.

1. **AI Model** will keep track of the users selections on different code prompts.
2. The user's performance data will be placed into a formatted report for the instructor.
3. Report will show percentages for the users answers to the code suggestions that were either correct or incorrect.
4. The administrator can see which areas of learning need more focus for the user (student).

Use Case 9: Monitoring User’s (Student) Progress

Actors: Administrator, AI Model

Administrator wants to track students’ learning and figure out concepts that might need extra review.

1. **AI Model** will show the users weekly or monthly progress from the detailed report.
2. **AI Model** will show a breakdown of computer science topics that the user has been struggling with the most.
3. The **Administrator** can view different students' progress on a weekly basis and see if areas that need improvement are getting better, plateauing, or declining.
4. If multiple students are making similar errors in the same areas of their code, extra review on these topics can be implemented.

Use Case 10: AI Generated Quiz Based off Previous Topics

Actors: User, AI Model

The AI Model will supply the user with a weekly quiz on topics that were in the code suggestions the week prior.

1. **AI Model** will log topics that are discussed throughout each week.
2. **AI Model** will determine which areas students need most improvement in based off of their learning reports.
3. **AI Model** generates the quiz and makes it available to the users.
4. The User takes a new quiz every week.
5. **AI Model** tracks correct and incorrect quiz responses and makes results available to both the user and the administrator.

6. The **User** can view their results as well as a review on why they answered incorrectly on some questions if necessary.

*Last updated by **Nicholas Rucinski***

Getting Started

How to get started running the program locally. The first step is to clone the main branch in the GitHub repository.

Running the Extension

How to run the extension in a local environment.

Running the Server

How to run the server in a local environment.

Running the Website

How to run the website in a local environment.

Using the Program

Website Usage

Running the Extension

How to run the extension in a local environment.

Install the dependencies

Node.js is required and can be installed from here. From the "extension" folder in the root directory run:

```
npm install
```

The extension requires that the server to be running. If the server is deployed then setting the testing variable in "extension/src/api/types/endpoints.ts" to false will use that version. If set to true the extension will look for a localhost server running on port 8001.

Running in debug mode

Open the extension folder in VSCode Once it is loaded pressing `f5` will opne a debug version of VSCode with the extension installed. Alternatively, you can open the command palette with either `CMD+Shift+P` on Mac or `CTRL+Shift+P` on Windows/Linux and search for "Debug: Start Debugging". In the original VSCode instance there should now be a terminal open with the tab "Debug Console" open. This will display that clover has been activated and any console.log statements will be shown here.

Packaging the extension

For a more detailed explanation visit the Official VSCode Docs First you need to install the Visual Studio Code Extension tool

```
npm install -g @vscode/vsce
```

To package the extension run:

```
vsce package
```

This will look at the values in the extensions package.json and generate a .vsix file that can then be installed in VSCode.

Installing from a .vsix file

Inside of the extension menu in VSCode:

- Press the three dots in the top right of the menu
- Press the "Install from VSIX..." option
- Locate the file that was either downloaded or just created when packaging the extension.

Publishing the extension

The extension is currently hosted on the Visual Studio Marketplace under Nick Rucinski's account(tur45021@temple.edu). The publisher id is "capstone-team-2.temple-capstone-clover" and the name of the extension is "Temple Capstone Clover".

The extension can either be updated from the command line by running

```
vsce publish
```

in the extension directory or by going to the extension management area in the Visual Studio Marketplace and manually updating the extension with a .vsix file.

Building the docs

Run

```
npx typedoc
```

in the extension directory.

Docs automatically get put in "/webserver/docs/tsdoc" so they can be served from the api.

To update the docs shown on docusaurus move the tsdoc folder to /documentation/static/. This goes for the extension and webserver docs. To update the api doc run the server with the new changes and navigate to localhost:8001/apispec_1.json and download the file. Replace the old apispec in "/documentation/static" with the new file

Running tests

To run the test `fetchSuggestion`:

```
npx jest suggestion.test.ts
```

*Last updated by **Nicholas Rucinski***

Running the Server

How to run the server in a local environment.

Create the .env

A .env with AI keys and database credentials can be made available from any of the team members.

AI Keys Setup

The current version is using Google's Gemini AI. An API key can be generated here. In the .env file in the server directory copy the key following this format not including the brackets.

`GEMINI_API_KEY=<YOUR KEY HERE>`

Database Setup

The database that is in use is Supabase. In the dashboard for a created project under "Project Settings/Data API". From this dashboard copy the project url, anon/public key, and the service_role/secret key and place them in the .env file using this format without the brackets.

`SUPABASE_URL=<YOUR KEY HERE>`

`SUPABASE_KEY=<YOUR KEY HERE>`

`SUPABASE_SERVICE_KEY=<YOUR KEY HERE>`

Install the Dependencies

The server uses Python3 and the version we used was 3.12.0. Download Python Here. For dependency management our team uses a requirements.txt file and a virtual environment. Inside of the server folder

Create the virtual environment:

This must be activated to run the server.

Windows

```
py -m venv .venv
```

Mac/Linux

```
python3 -m venv .venv
```

Activate/Deactivate the virtual environment

Windows

```
.venv\Scripts\activate
```

```
deactivate
```

Mac/Linux

```
source .venv/bin/activate
```

```
deactivate
```

Install the dependencies

With the virtual environment active run

```
pip install -r requirements.txt
```

Running the program

```
flask --app app run --debug --port 8001 --host "0.0.0.0"
```

or without setting up the virtual environment the run.py file can be used to setup a virtual environment, install dependencies, run tests, create a coverage report, generate the documentation, and finally run the app all in one go.

Running Tests

From the webserver directory start the virtual environment and run

```
pytest tests/ -v
```

This will run all the tests in the tests directory. Adding more v's to the arguments adds additional verbosity to the output.

*Last updated by **Nicholas Rucinski***

Running the Website

How to run the website in a local environment.

Setup the .env

Like with the server an already configured .env can be made available from any of the team members.

From the Supabase. dashboard for a created project under "Project Settings/Data API" you will need these keys, project url, anon/public key, and the service_role/secret key and place them in the .env file using this format without the brackets.

`VITE_SUPABASE_URL=<YOUR KEY HERE>`

`VITE_SUPABASE_ANON_KEY=<YOUR KEY HERE>`

Install the dependencies

Node.js is required and can be installed from here. From the "website" folder in the root directory run:

```
npm install
```

The extension requires that the server to be running. If the server is deployed set the url to that deployment in the .env file otherwise it will try to connect to a localhost server running on port 8001.

Running the Website

The website can be run using

```
```

```
npm run dev
```

A few links to the local website should be generated in the terminal for you to click. This method uses Vite to allow hot reloading to changes update instantly in the browser.

Building the Website

To build the extension to be deployed run

```
npm run build
```

Doing this will generate a dist folder within the website directory. This can than be served using a tool like Serve or Nginx.

*Last updated by **Nicholas Rucinski***

Using the Program

Website Usage

Current hosted at Clover.nickrucinski.com

Without signing in all users can access

- Home
- Download
  - Getting Started
- About
- Log In
- Sign Up

After signing in the user will have access to their dashboard. Their are currently four different dashboards:

- Student
  - The student dashboard can be filtered based on all classes, a specific class, or coding activities that were done outside of class using the drop down in the top right
  - The rest of the dashboard gives various charts and graphs that show their usage statistics
- Instructor
  - The instructor dashboard can be filtered using the drop down in the top right to only show details for a specific class
  - In the top left there is a button to create a new class that has a form to fill in data about the new class and a color
  - The rest of the dashboard gives various charts and graphs that show information about their classes
- Admin
  - Gives a list view of users signed up that can be filtered by role, name, or status
  - When a user is clicked on it shows their information including logs and any classes they belong to or instruct

    - At the bottom of the user data is a section for settings that can be changed to effect how the extension will operate
        - If these settings are changed the user will need to restart the extension or re-login for them to take effect
- Developer
    - Gives the most details but is only for usage within the team for debugging

Each level of access also has access to the roles below them so an admin can still access their student dashboard and use the application normally.

Also when the user is signed in their profile image will take the place of the sign in and out buttons and this is where they can:

- Sign out
- View/Edit their profile

Extension Usage

The extension is only for the Visual Studio Code editor.

The extension can be downloaded by going to the extension tab within VSCode and searching for Temple Capstone Clover or by visiting this link. It should automatically open VSCode for you if its installed.

When first starting the extension a new UI element will show up along the bottom bar on the right telling you to sign in. Clicking this will take you to your VSCode command palette where you are directed through a menu to either sign in or sign up. After authenticating, you are able to open a file and start getting code suggestions. In the bottom right there is a pair of curly braces that turn into spinning arrows when the editor is waiting for a response and sometimes it can take a little bit of time. All suggestions at this point will go your Non-Class Activities section. If you have signed up for a class, it can be picked using the "SELECT CLASS" button on the bottom bar of the editor on the left. This will show a view of all classes you are registered for and associated all code suggestions with that class.

Extension Commands

VSCode commands can be accessed with the command palette. This is either accessed by clicking the search bar at the top of the screen and and adding a `>` before searching for the command or by using the keybind `CTRL+SHIFT+P` (Windows/Linux) and `CMD+SHIFT+P` (Mac).

- `Clover: Sign In to Clover`
  - Handles the sign in and sign up menu
- `Clover: Sign Out of Clover`
  - Signs you out of the extension if you are signed in
- `Clover: Fetch Settings`
  - Used to get new settings that were changed on the website while the extension was being used
- `Debug: Test fetchSuggestions`
  - Debug command to test if the connection to the AI is working

Extension Keybinds

- Accept Code Suggestion
  - `Tab` (Window/Mac/Linux)
- Reject Code Suggestion
  - `CMD+R` (Mac)
  - `CTRL+Space` (Windows/Linux)

Extension Settings

Extension settings can be modified in VSCode settings using the shortcut `CTRL+,`

Alternatively you can navigate to File>Preferences>Settings>User>Extensions in the top menu bar.

They will be found under the Clover section. These were originally to change the model being used on the backend but they have been deprecated for now.

*Last updated by **Nicholas Rucinski***

System Architecture

Document Overview goes here.

Component Overview

The AI-assisted coding assistant project consists of several key components and technologies, including a ...

Class Diagram

Overview

Sequence Diagrams

Use Case 1: Receiving Context-Aware Code Suggestions

Entity Relation Diagram

Copilot ERD

Ollama - AI Model

CodeLlama is an advanced code generation model developed by Meta AI. It is a large language model (LLM)...

Development Environment

Required Hardware

Version Control

Overview

Logging

Logging is critical for understanding user behavior and system performance for this project. Logs are used t...

Component Overview

The AI-assisted coding assistant project consists of several key components and technologies, including a **VS Code extension** for code suggestions, a **Flask backend** for processing and logging user interactions, **Ollama** as the AI model, **Supabase** for authentication and database storage, and a **Next.js dashboard** for users to track their progress.

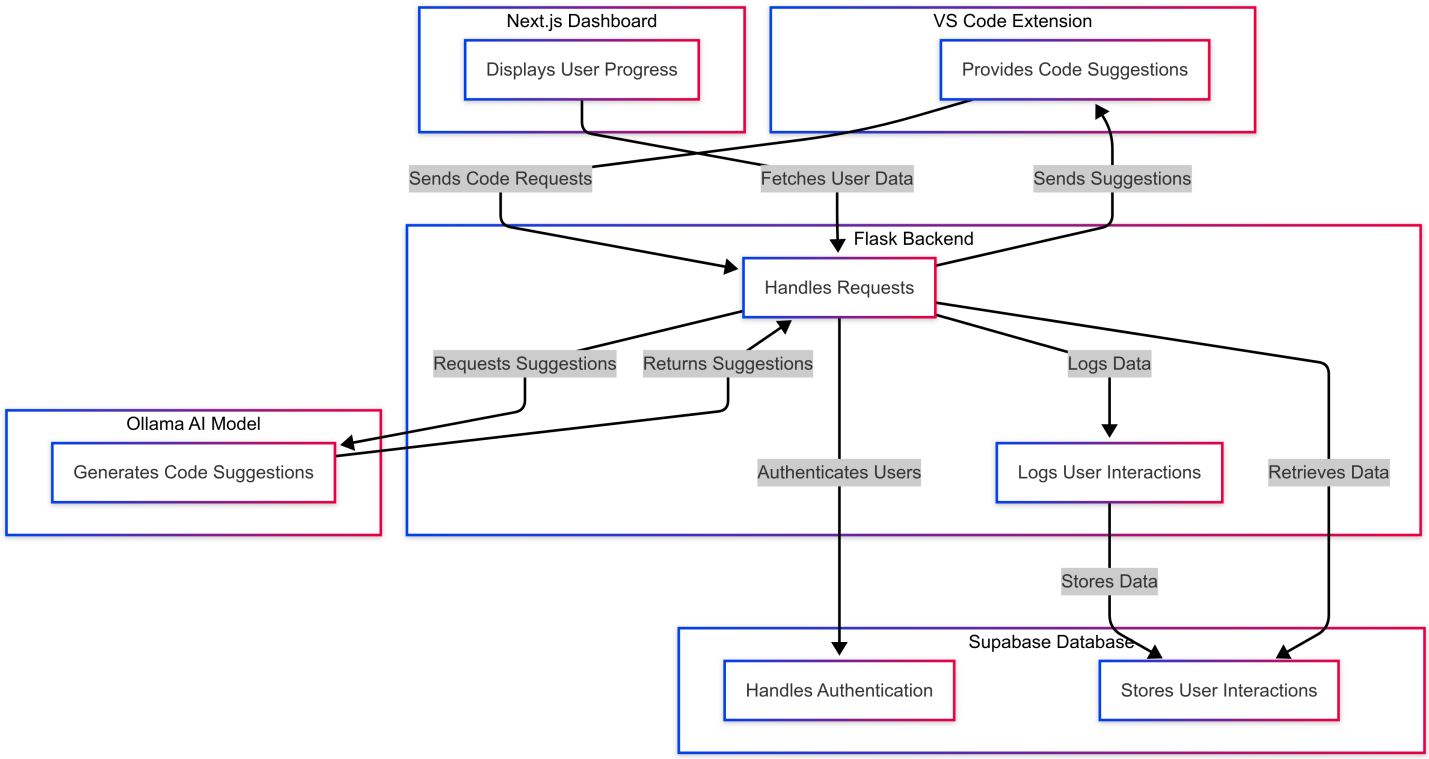


Figure 1. Component Diagram of the Github Copilot Clone application.

Main Application

🛠️ VS Code Extension (TypeScript, Node.js)

The core of the system is a VS Code extension that provides code suggestions while occasionally introducing small logic errors to test user attentiveness.

🔷 **Key Features:**

- **AI-Powered Code Suggestions** – Uses Ollama to generate helpful code recommendations.

- **Bug Injection** – Sometimes modifies code (e.g., changing `add(a, b)` to `a - b`) to check if users notice mistakes.
- **User Response Logging** – Tracks how users interact with suggestions (accept, modify, or reject).
- **Adjustable Difficulty** – Adapts suggestions based on past responses.

Backend Services

🌐 Flask (Python)

The backend handles all API requests between the VS Code extension, AI model, and database.

🔷 **Key Features:**

- **Processes Code Requests** – Sends user code to the AI model and returns suggestions.
- **Tracks User Behavior** – Logs whether users accept, modify, or reject suggestions.
- **Controls Suggestion Flow** – Can slow down suggestions or require manual edits based on user performance.

AI Model

🤖 Ollama (AI Code Generator)

The AI model generates code suggestions and sometimes introduces small mistakes to test users.

🔷 **Key Features:**

- **Context-Aware Suggestions** – Provides relevant recommendations based on the user's code.
- **Intelligent Mistakes** – Occasionally tweaks suggestions with logical errors to test user focus.
- **Adaptive Learning** – Adjusts suggestions based on user behavior.

Database & Authentication

🔐 Supabase (PostgreSQL, Auth)

Supabase handles user login and stores all interaction data.

🔷 **Key Features:**

- **User Authentication** – Manages logins and keeps track of individual progress.
- **Logs User Activity** – Records which suggestions were accepted, modified, or rejected.
- **Real-Time Sync** – Updates and injects user progress instantly to the AI model and on the dashboard.

User Dashboard

📊 Next.js (React, Tailwind CSS)

The dashboard allows users to view their progress and track their learning journey.

🔷 **Key Features:**

- **Progress Tracking** – Displays stats on correct vs. incorrect responses.
- **Insightful Feedback** – Helps users understand where they need improvement.
- **Encourages Learning** – Provides interactive insights to refine coding skills.

*Last updated by **Nicholas Rucinski***

Class Diagram

Overview

This class diagram represents the architecture of a **Code Suggestion System** integrated with a VSCode extension, a Flask backend, and AI-driven code suggestions.

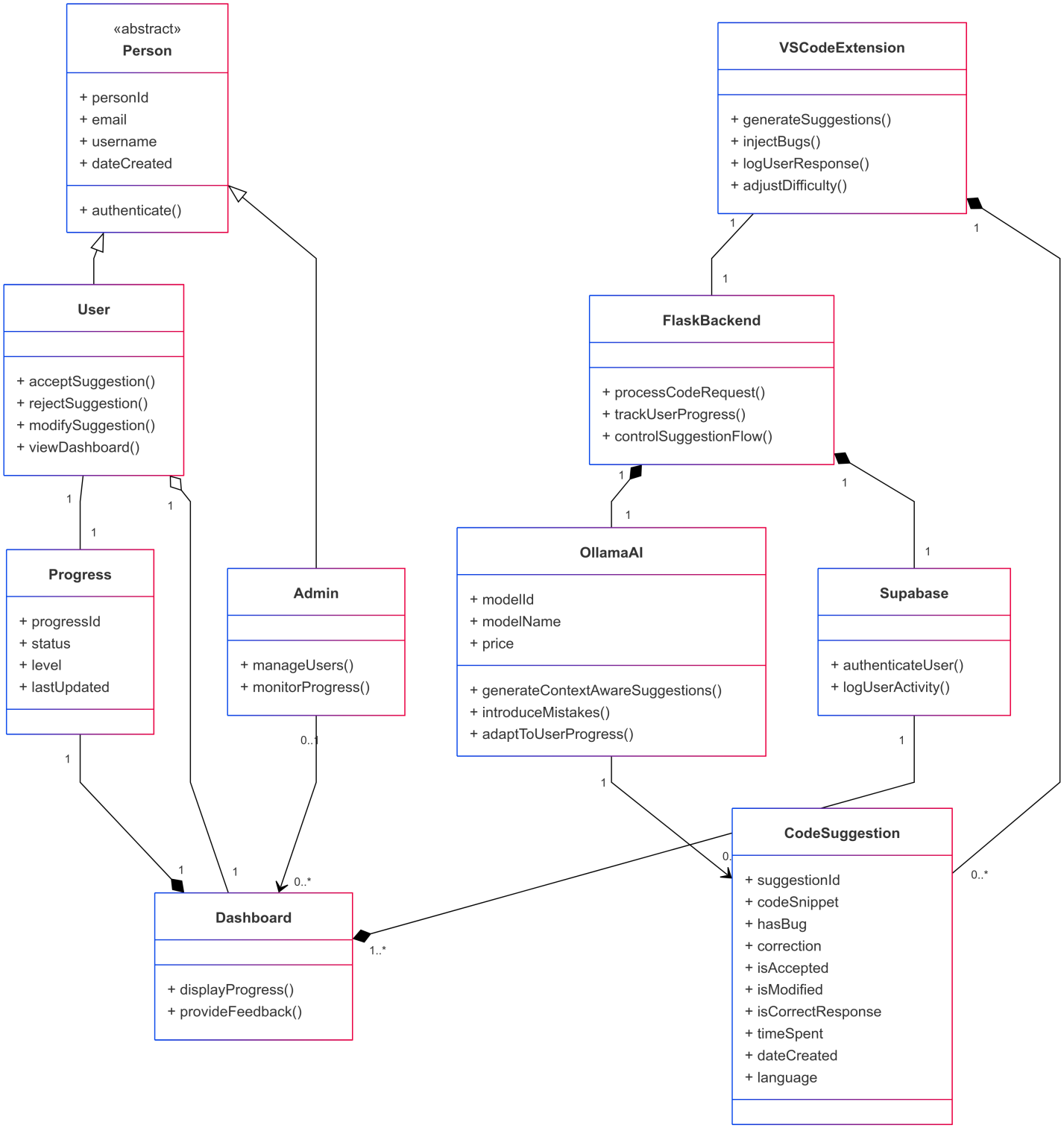


Figure 1. Class Diagram of the Github Copilot Clone application.

Classes and Relationships

👤 Abstract Class: Person

- Defines common attributes (`personId`, `email`, `username`, `dateCreated`) for **User** and **Admin**.
- Includes an `authenticate()` method.

👨‍💻 User & Admin

- **User** inherits from **Person** and can:
  - Accept, reject, or modify code suggestions.
  - View the dashboard.
- **Admin** inherits from **Person** and can:
  - Manage users.
  - Monitor progress.

🖥️ VSCodeExtension

- Generates suggestions and injects bugs to challenge the user.
- Logs user responses and adjusts difficulty.

🔧 FlaskBackend

- Handles code request processing and user progress tracking.
- Controls the suggestion flow.

🤖 OllamaAI

- AI-powered component that generates context-aware code suggestions.
- Can introduce mistakes and adapt to user progress.

☁️ Supabase

- Manages user authentication and logs user activity.

📊 Dashboard

- Displays user progress and provides feedback.

💡 CodeSuggestion

- Represents individual code suggestions with attributes like:
  - `codeSnippet`, `hasBug`, `correction`, `isAccepted`, etc.
- Linked to the AI system **(OllamaAI)**.

🚀 Progress

- Tracks user progress with `status`, `level`, and `lastUpdated`.

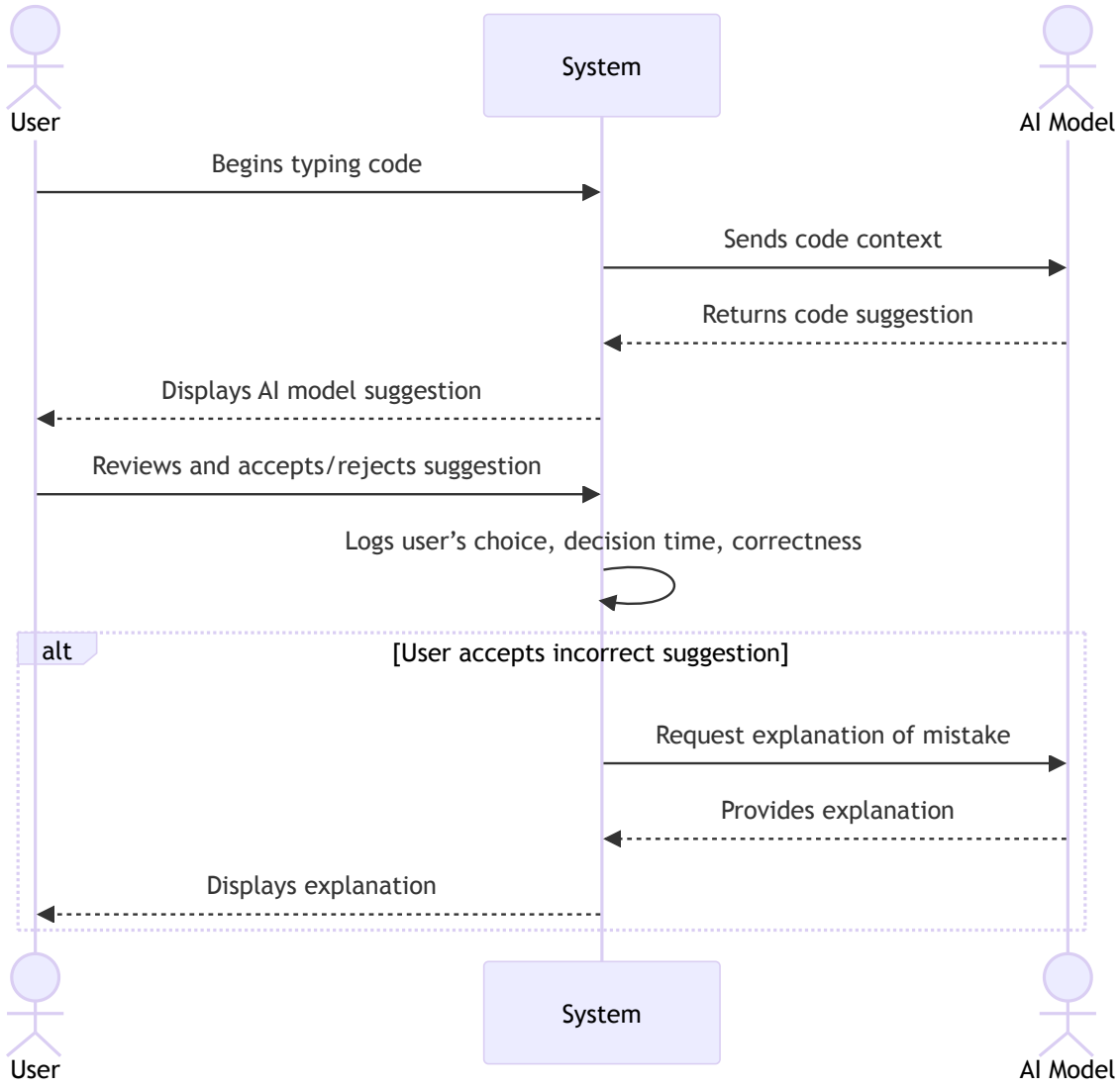
🔗 Key Relationships

1. **VSCodeExtension ↔ FlaskBackend** (1:1)
2. **FlaskBackend ↔ OllamaAI** (1:1)
3. **FlaskBackend ↔ Supabase** (1:1)
4. **OllamaAI ↔ CodeSuggestion** (1:many)
5. **User ↔ Progress** (1:1)
6. **Admin ↔ Dashboard** (0..1:many)
7. **Dashboard ↔ Progress** (1:1)
8. **Supabase ↔ Dashboard** (1:many)
9. **User ↔ Dashboard** (1:1)
10. **VSCodeExtension ↔ CodeSuggestion** (1:many)

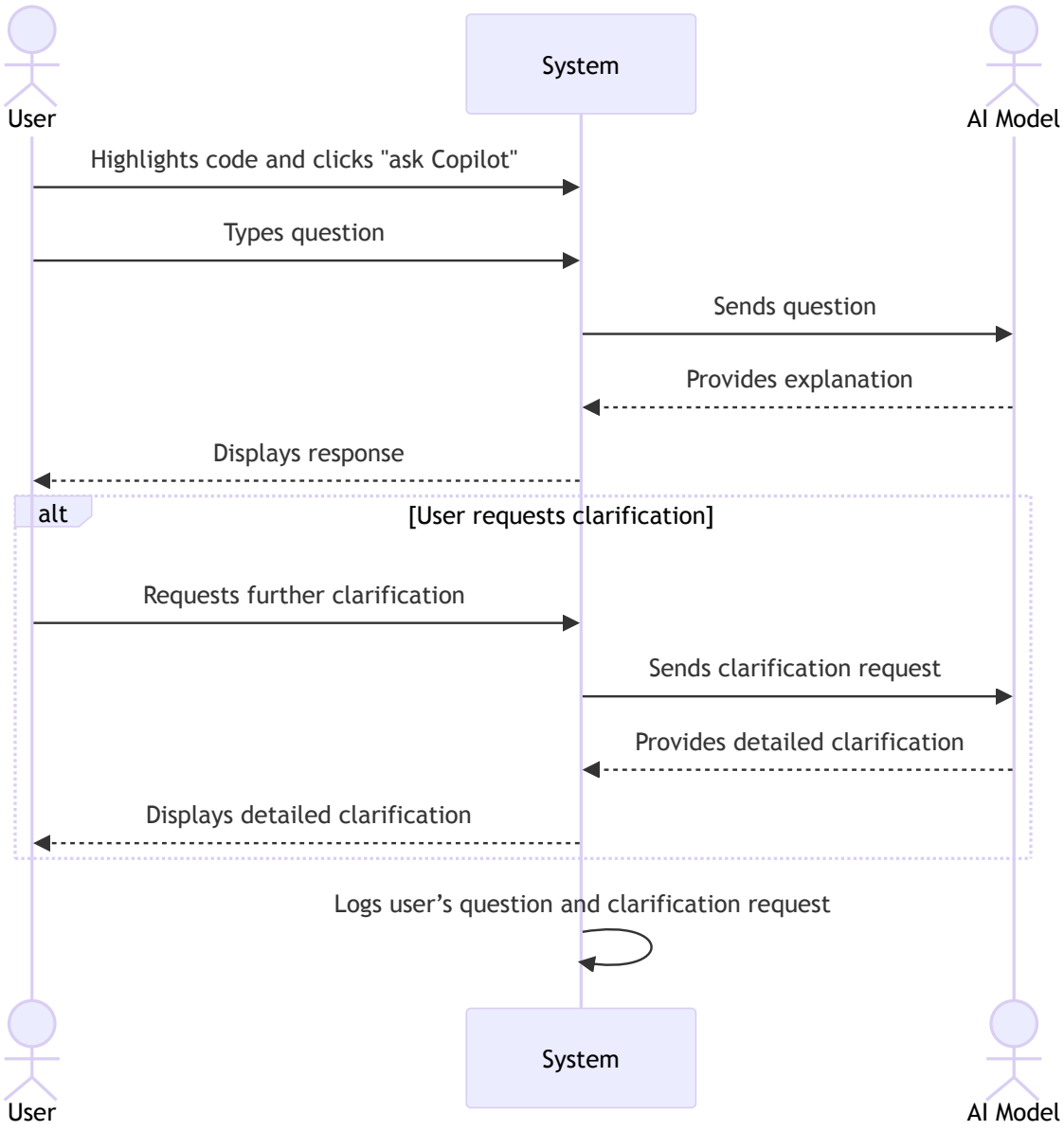
*Last updated by **Nicholas Rucinski***

Sequence Diagrams

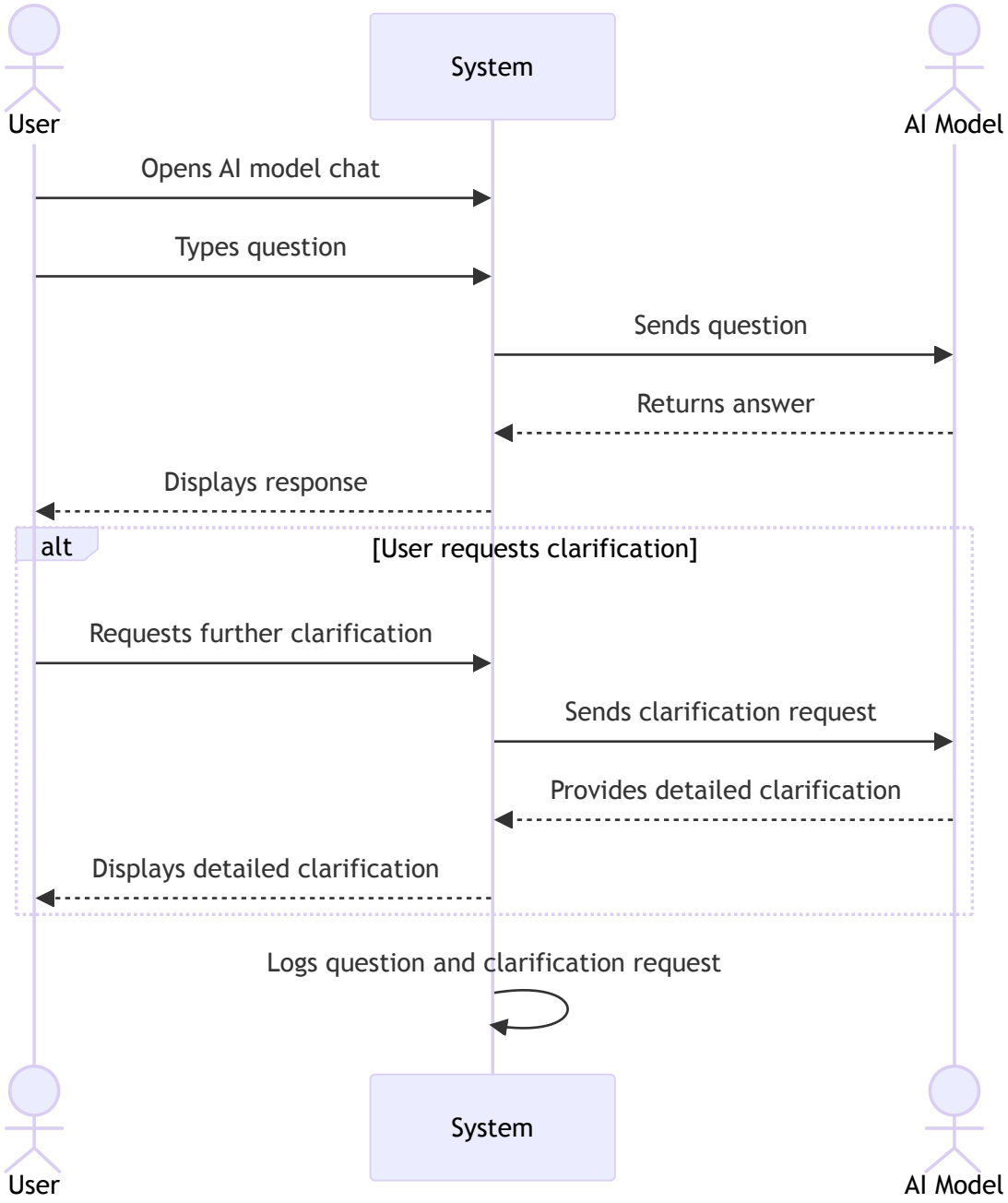
Use Case 1: Receiving Context-Aware Code Suggestions



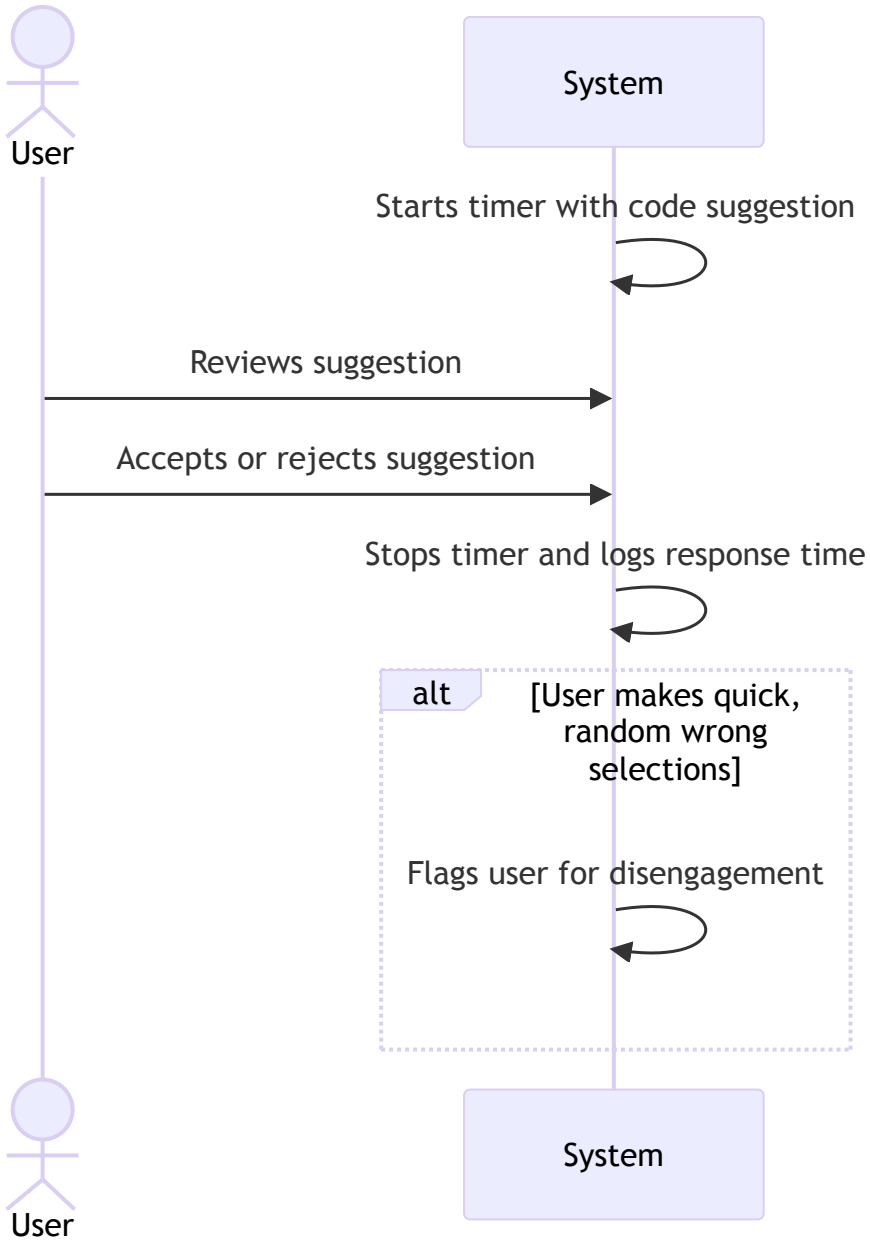
Use Case 2: Asking Inline Questions about Code



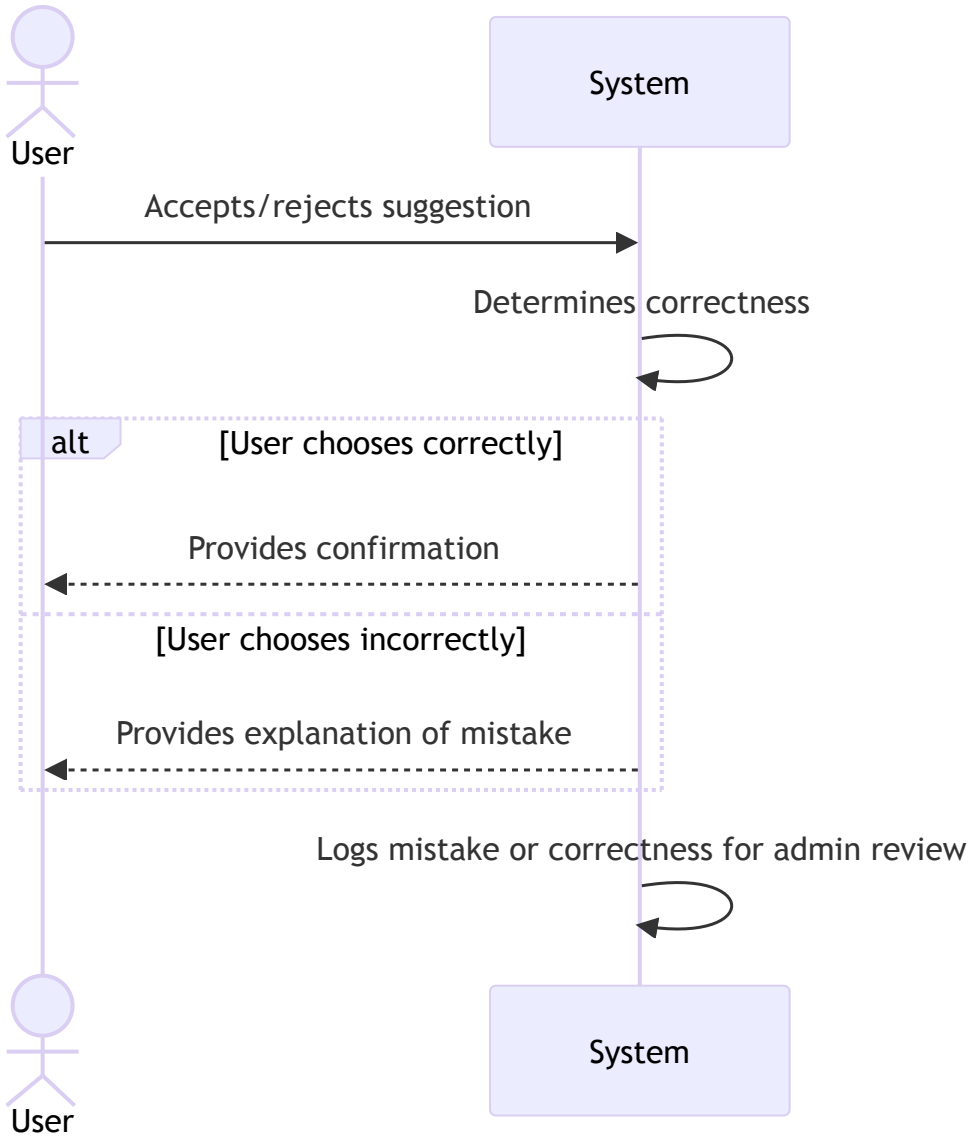
Use Case 3: Asking Questions in the Copilot Chat



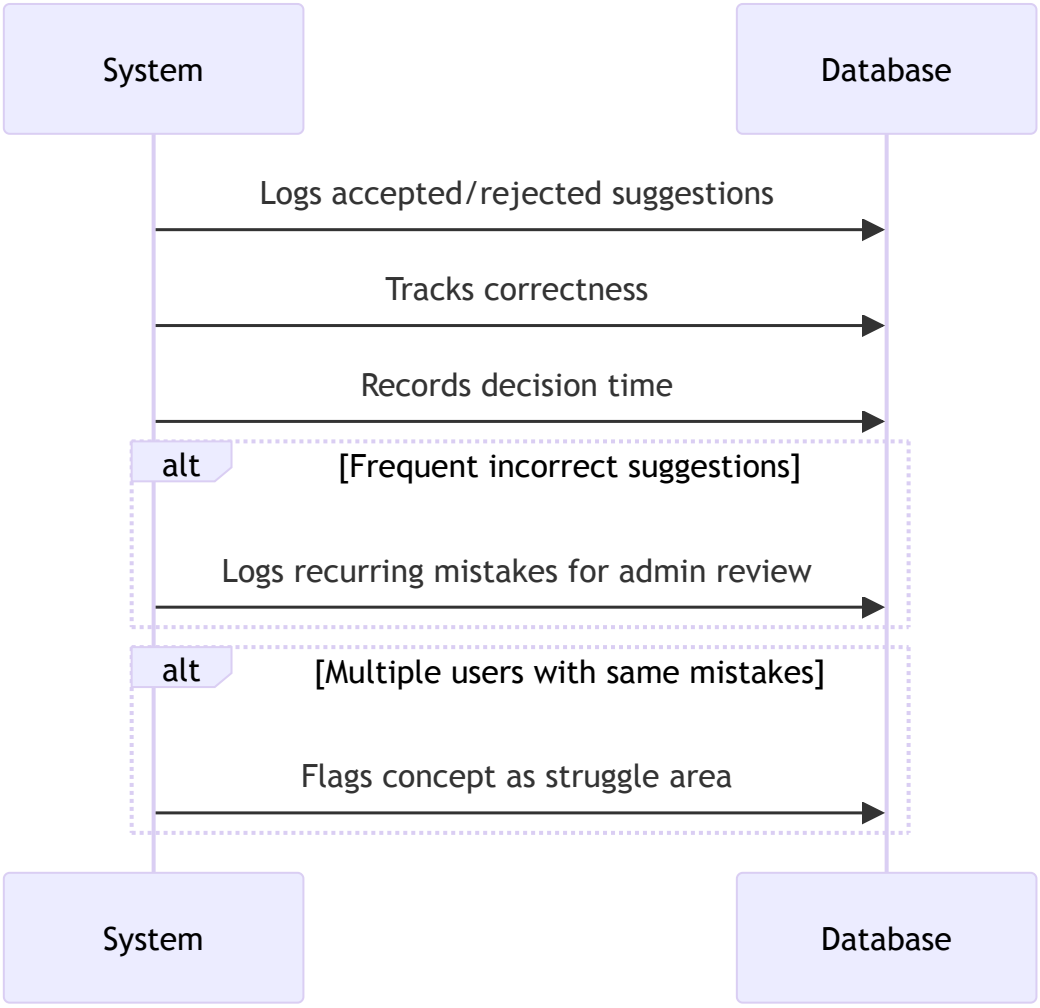
Use Case 4: Logging Decision Time for Code Suggestions



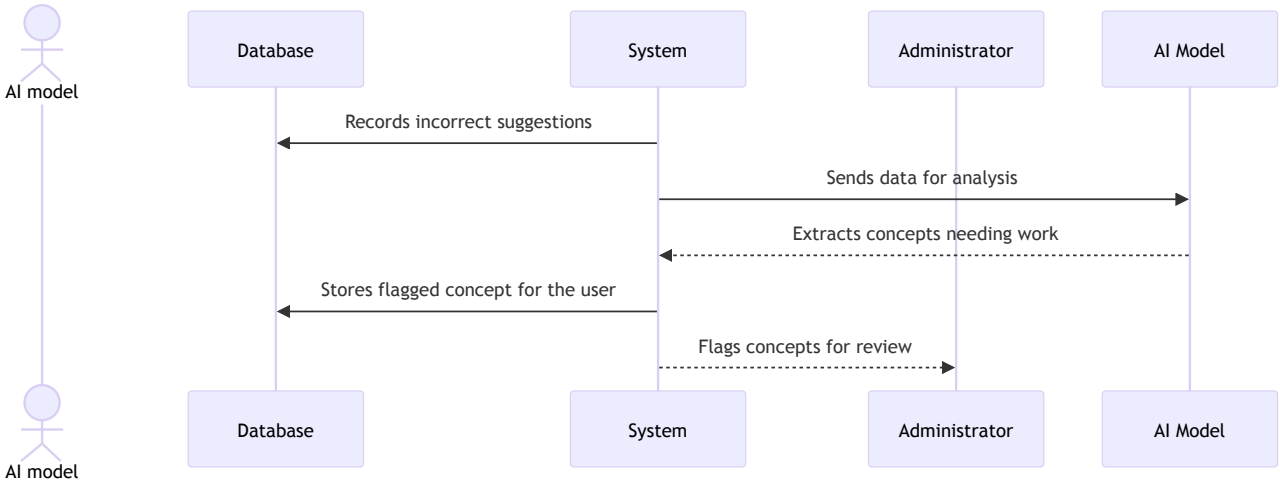
Use Case 5: Receiving Feedback After Selecting a Suggestion



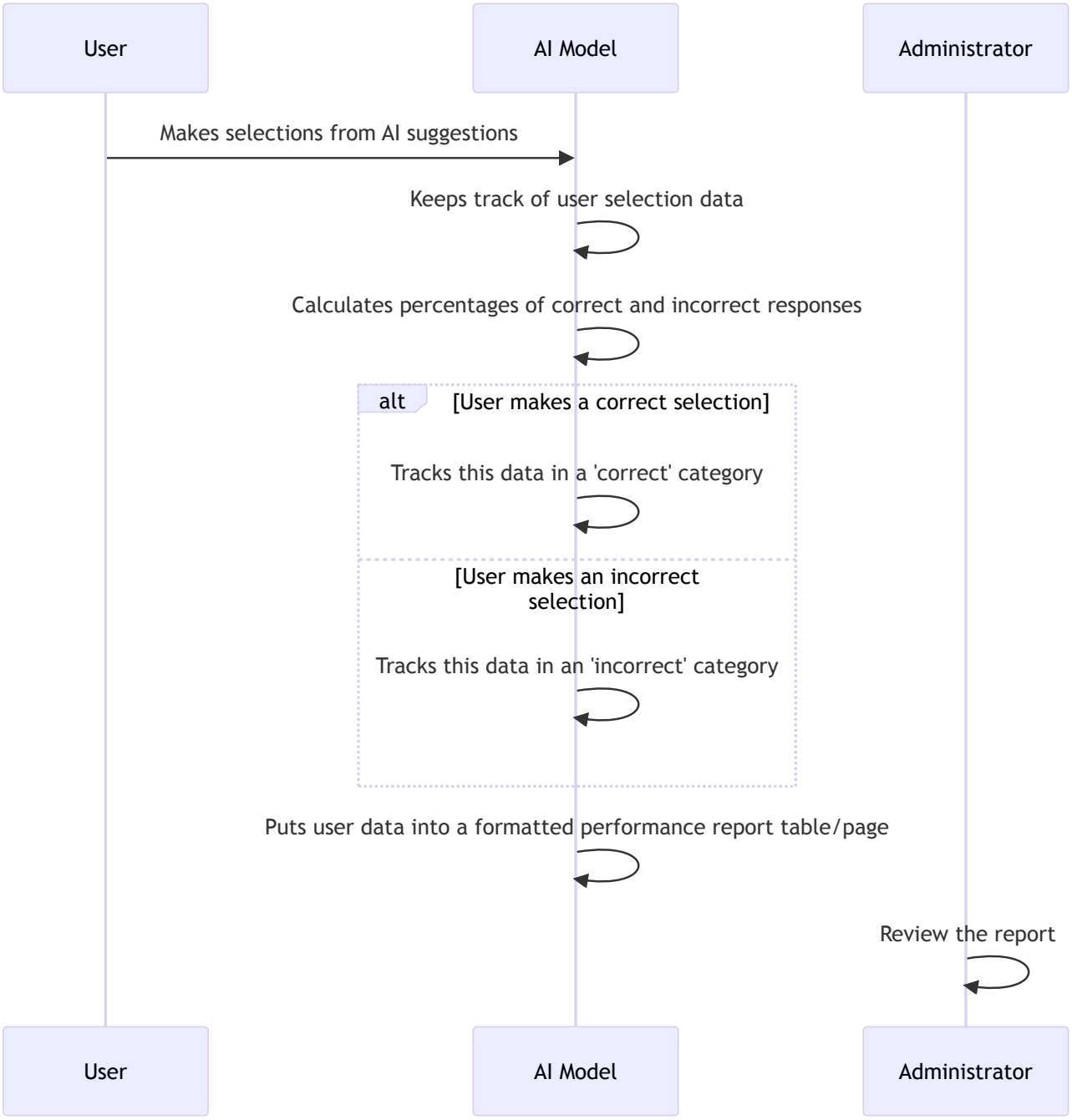
Use Case 6: Tracking and Logging User Decisions



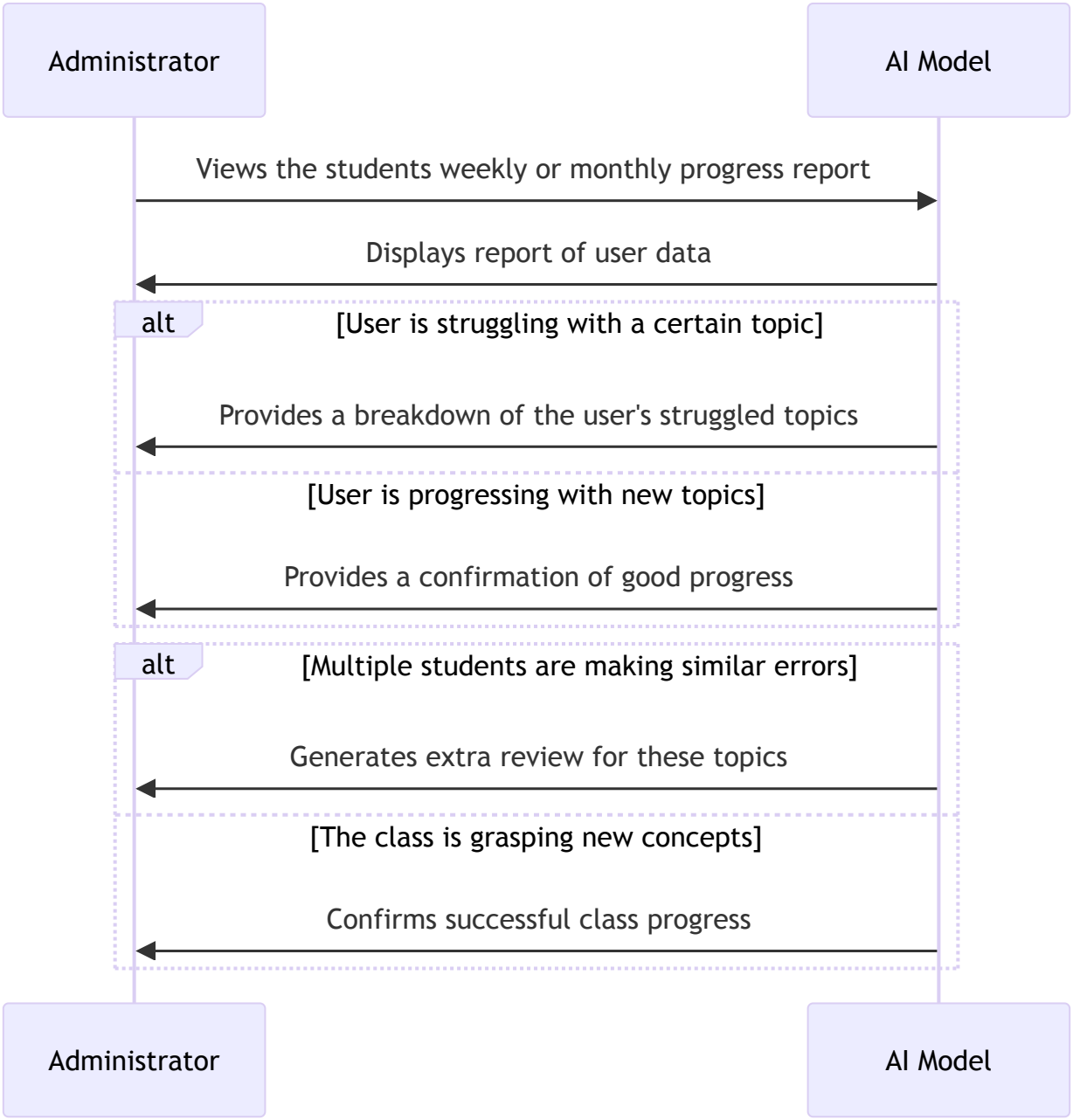
Use Case 7: Identifying Common Student Mistakes



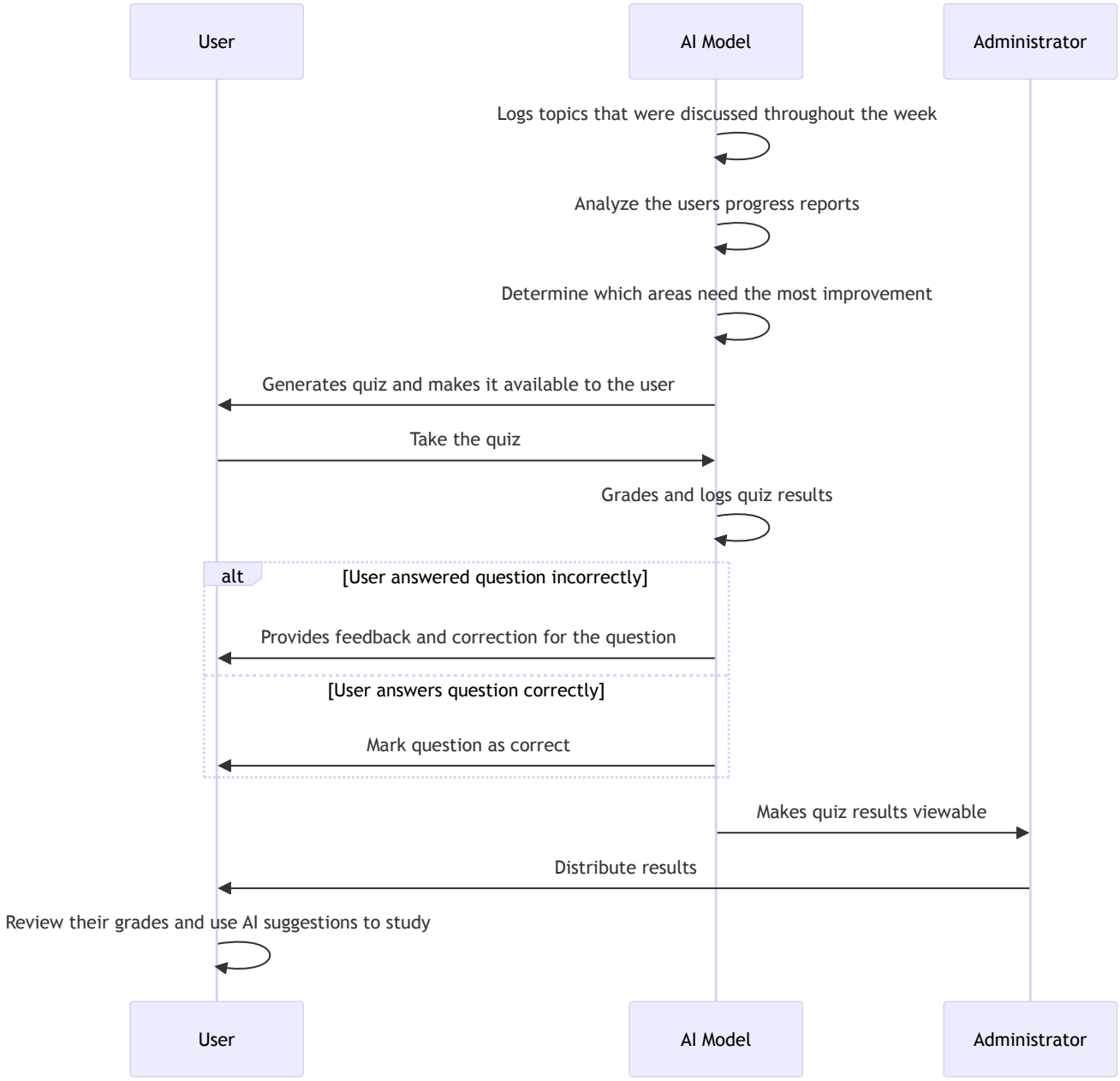
Use Case 8: Generating Learning Reports for Administrators



Use Case 9: Monitoring User's Progress



Use Case 10: AI Generated Quiz Based off of Previous Topics



*Last updated by **Nicholas Rucinski***

Entity Relation Diagram

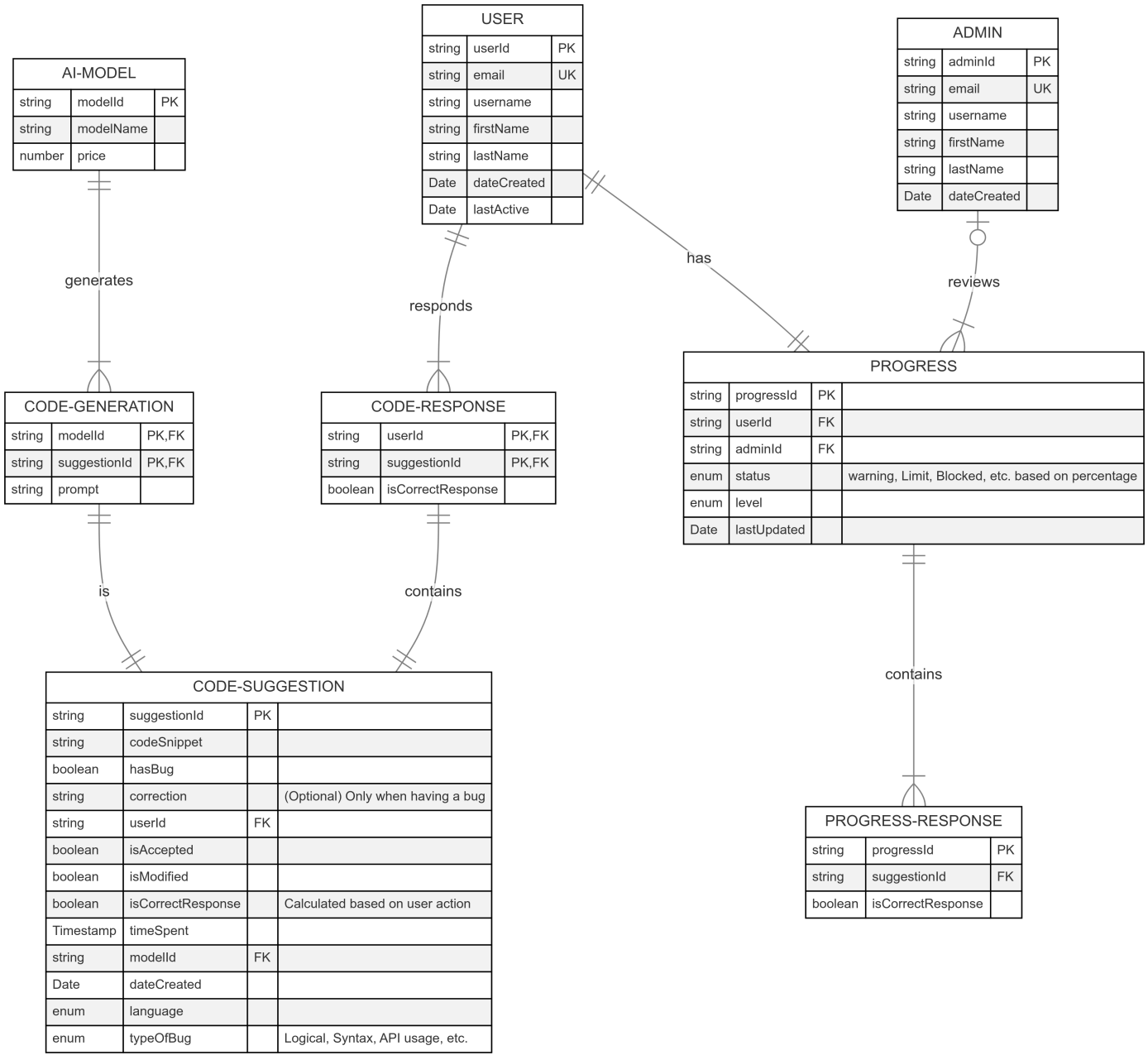


Figure 1. ER Diagram & Design Schema of the Github Copilot Clone application.

This **database schema** represents an **AI-assisted code suggestion** and **user progress tracking system**. It consists of entities for **users**, **admins**, **AI models**, **code suggestions**, and **progress tracking**.

- The **AI-MODEL** generates **CODE-GENERATION** entries that lead to **CODE-SUGGESTION** records, which users interact with.
- Users respond to suggestions, marking them as correct or modifying them, and these interactions are recorded in **CODE-RESPONSE**.
- **ADMIN** oversees user progress, tracking warnings and limitations in the **PROGRESS** table, which stores levels and statuses based on user activity.
- The **PROGRESS-RESPONSE** table links progress tracking to specific code suggestions.

This schema enables monitoring of **user engagement** with AI-generated code suggestions while maintaining administrative oversight.

*Last updated by **Nicholas Rucinski***

Ollama - AI Model

CodeLlama is an advanced code generation model developed by Meta AI. It is a large language model (LLM) designed specifically for coding tasks, leveraging the LLaMA architecture to provide efficient and high-quality code completion, generation, and understanding capabilities.

Key Features

- **Multiple Model Sizes**: Available in different sizes, including 7B, 13B, and 34B parameters, to balance performance and efficiency.
- **Context Awareness**: Handles extended code contexts effectively, making it useful for large-scale software projects.
- **Supports Multiple Languages**: Proficient in Python, C++, Java, JavaScript, Bash, and more.
- **Optimized for Efficiency**: Uses improved tokenization and inference techniques to enhance coding performance.

Important Statistics

- **Base Architecture**: LLaMA 2
- **Model Sizes**: 7B, 13B, 34B parameters
- **Training Data**: Trained on a dataset of publicly available code repositories
- **Max Context Length**: Up to 100K tokens

Use Cases

- **Code Completion**: Assists in writing and finishing code snippets efficiently.
- **Code Generation**: Generates entire functions, classes, and scripts based on prompts.
- **Code Understanding**: Helps in debugging and explaining complex code segments.
- **Refactoring and Optimization**: Suggests improvements and optimizations for better performance.

Availability

CodeLlama is open-source and available for research and commercial use under Meta's licensing terms.

For more details, visit the official repository.

*Last updated by **Nicholas Rucinski***

Development Environment

Required Hardware

- For the local AI model, a machine with
  - at least 8GB of ram
  - at least 12GB of storage.

Tools

- **IDE**
  - Visual Studio Code for extension development
    - Visual Studio Extension Test Runner
  - Any text editor for other development
- **AI**
  - Ollama
    - Tool to run different AI models
- **Package Managers**
  - Pip for Python
  - Npm for Visual Code Extension
- **Documentation Generators**
  - TypeDoc
    - HTML documentation generator for Typescript
  - Sphinx
    - HTML documentation generator for Python
  - Flasgger
    - OpenAPI spec generator for Flask API

Languages

- **Python**

    - Flask for creating the API
- **Typescript**
    - For extension development

Testing

- **Postman**
    - For API endpoint testing
- **Pytest**
    - For Python testing
- **Test-electron** and **Test-cli**
    - For extension testing

*Last updated by **Nicholas Rucinski***

Version Control

Overview

The project is managed using **Git** and **GitHub**. The Git repository serves as a **monorepo** that integrates the following key components:

- **Docusaurus Documentation** 📚 – Project documentation
- **Extension** 🛠️ – Frontend & logic handling
- **WebServer** 🌐 – Backend operations

Branching Strategy

Main Branch (`main`)

- The `main` branch holds the most **stable and up-to-date** version of the project.
- No direct commits are allowed—changes are merged via **pull requests**.

Sprint Branch (`GCCB-SP#-main`)

- At the start of each **Sprint**, a new **working branch** from `main` is created in the format: `GCCB-SP#-main` (e.g., `GCCB-SP3-main` for Sprint 3)
- This branch **accumulates all changes and features** during the sprint.
- It ensures `main` remains clean while providing a dedicated branch for sprint development.

Feature Branches (`GCCB-[#]-`)

- Feature branches are created from the current `GCCB-SP[#]-main` branch through **Jira**.
- Naming convention: `GCCB-#-` where `#` is the Jira issue number.
- These branches are task-specific and linked to Jira issues.

End of Sprint Merging

- At the end of each sprint, the final stage of `GCCB-SP#-main` will be reviewed and merged back into `main`.
- Ensures all changes are stable before reaching production.

Branch Protection Rules

Rules are strictly enforced to maintain code quality and security:

Branch	Approval Required	Direct Pushes
`main`	2 approvals	❌ Not allowed
`GCCB-SP#-main`	1 approval	❌ Not allowed

Pull Request Process

1. **Create a Feature Branch** (`BP-#`) from the sprint branch.
2. **Push changes** and open a PR for review.
3. **Review & Approvals:**

- Feature branches: **1 approval** (Sprint Branch)
- Sprint branches: **2 approvals** (`main`)

*Last updated by **Nicholas Rucinski***

Logging

Logging for GitHub Copilot Clone

Logging is critical for understanding user behavior and system performance for this project. Logs are used to track user interactions, AI model behavior, and system events. All logs are stored in a **Supabase** database for analysis.

Table of Contents

---

1. Introduction to Logging

In this project, we log the following events:

- **User Actions**: When a user accepts or rejects a suggestion.
- **AI Model Behavior**: When the AI generates a suggestion or introduces a bug.

All logs are stored in a **Supabase** database for easy querying and analysis.

---

2. Logging Events

👤 User Actions

Tracking user interactions (`accept` and `reject`) provides critical insights into how users respond to AI-generated code.

Event	Purpose	Insights Gained
`accept`	User accepts a suggestion, indicating it may meet their expectations. `Tab`	Helps determine if users are accepting buggy suggestions without noticing.
`reject`	User rejects a suggestion, signaling an issue with its quality or relevance. `Backspace`	Indicates users' ability to identify incorrect or suboptimal code.

🤖 AI Model Behavior

Logging AI behavior ensures we understand how often AI introduces bugs.

Event	Purpose	Insights Gained
`generate`	AI generates a new suggestion.	Helps track the amount of generated suggestions.
`introduce_bug`	AI intentionally introduces a bug into the suggestion.	Allows tracking whether users detect and reject the introduced bug.

3. Log Format

All logs are structured in JSON format and each log entry contains structured data to facilitate precise analysis:

Field	Purpose	Insights Gained
`event_type`	Identifies the interaction or AI event (e.g., `accept`, `reject`, `generate`).	Enables categorization of user behavior and AI performance.
`timestamp`	Records when the event occurred in ISO 8601 format.	Allows for time-based trend analysis.
`time_lapse`	Captures the time taken by the user to act on the suggestion (in ms).	Helps measure how long users review suggestions before deciding, or how fast the AI responds
`metadata`	Provides additional context such as user ID, suggestion ID, and bug status.	Supports user-specific or suggestion-specific analysis.

Why It's Important?

- **`time_lapse` Analysis**: If users spend very little time reviewing code but accept most suggestions, they might not be checking thoroughly.
- **Bug Tracking (`bug_introduced`)**: Helps compare the acceptance rates of buggy vs. correct suggestions.
- **Metadata for User Patterns**: By analyzing patterns across multiple users, we can determine who is more or less attentive.

Example Log Entry

```
{
  "event_type": "accept",
  "timestamp": "2025-02-21T20:40:52.709Z",
  "time_lapse": 1200,
  "metadata": {
    "user_id": "12345",
    "suggestion_id": "67890",
    "bug_introduced": false
```

```
  }
}
```

4. Analysis Scenarios

👨‍💻 A user frequently accepts buggy code

- If logs show that a user accepts buggy suggestions within very short `time_lapse`, they may not be reviewing code thoroughly.
- If the user later rejects suggestions more frequently after being exposed to buggy code, they may be learning from past mistakes.

👨‍💻 Users take different amounts of time to decide

- A user who spends more time (`time_lapse`) before accepting a suggestion is likely analyzing it carefully.
- If a user quickly rejects most suggestions, they might have a higher standard or a preference for writing their own code.

👨‍💻 AI generates too many rejected suggestions

- If a high percentage of generated suggestions are rejected, the AI might need improvement in suggestion quality.
- Tracking `introduce_bug` events can help determine whether AI-introduced bugs are too obvious or too subtle.

5. Supabase Integration

1. Set Up Supabase

- Create a Supabase project at Supabase.
- Create a table named `logs` with the following schema:

```
```

```
CREATE TABLE logs (
    id SERIAL PRIMARY KEY,
    event_type TEXT NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    time_lapse INTERGER,
    metadata JSONB
);
```

2. Logging to Supabase

Use the Supabase client to insert logs into the `logs` table.

Example: Logging in Python (Flask Backend)

```
from supabase import create_client, Client
import json
from datetime import datetime

# Initialize Supabase client
SUPABASE_URL: str = os.getenv('SUPABASE_URL')
SUPABASE_KEY: str = os.getenv('SUPABASE_KEY')
client: Client = create_client(SUPABASE_URL, SUPABASE_KEY)

def log_event(event_type, start_time, metadata=None):
    end_time = datetime.utcnow()
    time_lapse = int((end_time - start_time).total_seconds() * 1000)

    log_entry = {
        "event_type": event_type,
        "timestamp": end_time.isoformat(),
        "time_lapse": time_lapse,
        "metadata": metadata
    }

    response = supabase.table("logs").insert(log_entry).execute()
    print("Logged event:", response)
```

---

6. Analyzing Logs

Querying Logs in Supabase

Use Supabase's SQL interface or client libraries to query logs.

💡 Query All Logs

```
def get_all_logs():
    return supabase.table("logs")
```

💡 Query Users Frequently Accepting Buggy Code

```
def get_frequent_bug_acceptors(user_id):
    response = supabase.table("logs")
        .select("metadata->>user_id, time_lapse")
        .eq("event_type", "accept")
        .eq("metadata->>bug_introduced", "true")
        .eq("metadata->>user_id", user_id)
        .execute()
return response.data
```

💡 Query Average Amount of Time to Decide

```
def get_avg_decision_time(user_id):
    response = supabase.table("logs")
        .select("metadata->>user_id, time_lapse, event_type")
        .eq("metadata->>user_id", user_id)
        .in_("event_type", ["accept", "reject"])
        .execute()

    total_time = sum(log["time_lapse"] for log in response.data)
    total_actions = len(response.data)

    avg_time = total_time / total_actions if total_actions > 0 else 0
    return avg_time
```

💡 Query AI Rejection Percentage Due to Bugs

```
def get_ai_rejection_percentage():
    response = supabase.table("logs")
```

```
        .select("event_type, metadata->>bug_introduced")
        .in_("event_type", ["accept", "reject"])
        .execute()

    total_actions = len(response.data)
    total_rejections = sum(1 for log in response.data if log["event_type"]
== "reject")
    rejection_percentage = (total_rejections / total_actions) * 100 if
total_actions > 0 else 0

    return rejection_percentage
```

---

7. Best Practices

- **Structured Logging**: Always log in JSON format for easy parsing.
- **Avoid Sensitive Data**: Avoid logging sensitive information like API keys or passwords.

*Last updated by **Nicholas Rucinski***

API Specification

Please remove and replace examples where necessary.

Backend API

FlaskBackend

Frontend API

VSCodeExtension

API Spec

API Specification from apispec_1.json

Backend API

FlaskBackend

Component used to handle code suggestions, user requests and the flow of suggestions.

Methods:

`processCodeRequests()`

- Takes incoming user made coding requests generated from the AI tool.
- **Returns:** `CodeSuggestion`
  - The AI-generated code response.

`trackUserProgress()`

- Handles the user's suggestion choices.
- Correct and incorrect modifications and choices are stored and used to track progression or decline while using the copilot tool.
- Progress data fields are updated with corresponding tracking data.
- **Returns:** `boolean`
  - `true` if tracking is successful.

`controlSuggestionFlow()`

- Control the frequency of coding suggestions presented to the user.
- Control where within the user's code the suggestions will be made.
- **Returns:** `void`

OllamaAI

AI Component that will be used to monitor progress and generate coding suggestions and intentional errors to help the user learn new coding concepts.

Data fields

- `modelId: integer`: Classifies which OllamaAI model is being used.
- `modelName: string`: The name of the AI model that is being used.
- `price: float`: The price of the AI service. Monthly or yearly subscription for access to the service and its features.

Methods

`generateContextAwareSuggestions()`

- Creates an AI generated, correct or incorrect coding suggestion to the user.
- User's action taken on the suggestions is logged.
- **Returns:** `CodeSuggestion`
  - AI-generated code snippet.

`introduceMistakes()`

- Presents the user intentionally incorrct code.
- Code is generated with what the user is working on in their codespace, requiring careful analysis to determine that it is an error.
- **Returns:** `CodeSuggestion`
  - Buggy AI-generated code snippet.

`adaptToUserProgress()`

- Monitor correct and incorrect suggestion choices and use the AI model to update level and difficulty.
- Calls the trackUserProgress function.
- Returns coding suggestions based off of the user's progression.
- **Returns:** `void`

SupaBase

Component that will provide backend storage and authentification.

Methods

`authenticateUser()`

- Calls a person's credential information.
- Confirms information with that in the database to allow for access to an account.
- **Returns:** `boolean`
  - `true` if authentication is successful.

`logUserActivity()`

- Track a user's suggestion choices.
- Create logs in order to update difficulty of prompts and adapt to progress.
- Update suggestion settings for different users based off their progress.
- **Returns:** `boolean`
  - `true` if logging is successful.

*Last updated by **Nicholas Rucinski***

Frontend API

==========

VSCodeExtension

Component that will work with the copilot education tool to allow for specific feature integrations and interactions.

Methods

`generateSuggestions()`

- Generate code based on what the user has written in VSCode.
- Returns suggestions to the user from the backend database that are created using API service.
- **Returns:** `CodeSuggestion`
  - An AI-generated code snippet.

`injectBugs()`

- Create bugs in the users VSCode work space.
- Method takes in bug suggestions from AI extension that are stored in backend database.
- **Returns:** `CodeSuggestion`
  - A buggy code snippet.

`logUserResponse()`

- Actions taken within the extension are logged to the database.
- Verify the user's action as accept, reject, or modify.
- User responses are passed to backend database.
- **Returns:** `boolean`
  - `true` if the response is logged successfully.

`adjustDifficulty()`

- Method to intensify or dial back the difficulty of the AI generated suggestions.
- VSCode extension will process the users progresss through backend data to properly adjust the difficulty of their suggestions.
- **Returns:** `void`

Dashboard

Component that allows the user to view their progress while using the Copilot tool, as well as for the administrator to provide feedback.

Methods

`displayProgress()`

- Displays the user progress based on `userId` and `progressId` (string or int).
- Fetches `Progress` data fields.
- Accesses user suggestion details to display results and feedback.
- **Returns:** `Progress`
  - User's current progress.

`provideFeedback()`

- Allows the administrator to give users feedback that can be accessed in their dashboard.
- Updates the level data field for different tasks.
- The `lastUpdated` attribute updates when new feedback is entered.
- **Returns:** `boolean`
  - `true` if feedback is submitted successfully.

---

*Last updated by **Nicholas Rucinski***

API Spec

Search...

- Logging

Github Copilot Extension (0.0.1)

Download OpenAPI specification:Download

powered by Flasgger

Logging

Logs the event to the database. See Swagger docs for more information.

Logs the event to the database.

Request Body schema: application/json

data required	object
event required	string
timestamp required	number

- post Logs the event to the database. See Swagger docs for more information.
- get Retrieve all logs in the database See Swagger docs for more information.
- get Get all logs for a specific user See Swagger docs for more information.

Responses

200
Event logged successfully

400
Bad request or invalid input

500
Internal server error

post/log

https://civic-interactions-lab.github.io/log

- Payload

Content type
application/json

post Generate a suggestion based on the provided prompt. See Swagger docs for more information

Copy Expand all Collapse all

Logs

Retrieve all logs in the database See Swagger docs for more information.

Fetches all logged events from the database.

Responses

200
List of logs

500
Internal server error

get
/log
s

https://civic-interactions-lab.github.io/logs

Get all logs for a specific user See Swagger docs for more information.

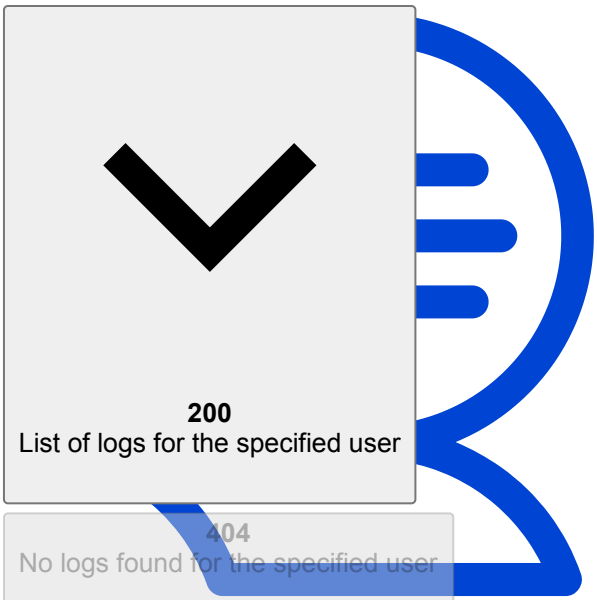
Fetches all logged events associated with a specific user ID.

path Parameters

user_id required	integer Example: 12345 The ID of the user whose logs are being retrieved.

- post Create a new user with first name, last name, email, and password.
- get TODO Get a specific user by ID

Responses



API docs by Redocly

https://civic-interactions-lab.github.io/logs/{user_id}

Suggestions

Generate a suggestion based on the provided prompt. See Swagger docs for more information.

Sends a prompt to the locally running Ollama model and returns the generated suggestion.

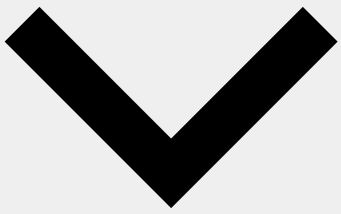
Request Body schema: application/json

prompt required	string

Responses

200
Successfully generated suggestion

400
Bad Request - No prompt provided



500
Internal Server Error - Failed to generate response

post/sugge
stion

https://civic-interactions-lab.github.io/suggestion

- Payload

Content type

application/json

Copy

- 200
- 400
- 500

Content type

application/json

Copy Expand all Collapse all

Users

Create a new user with first name, last name, email, and password.

Registers a new user with first name, last name, email, and password.

Request Body schema: application/json

email required	string
first_name required	string
last_name required	string
password required	string

Responses

201
User created successfully

400
Bad request (missing fields or email already exists)

500
Internal server error

post/users

https://civic-interactions-lab.github.io/users

- Payload

Content type

application/json

Copy

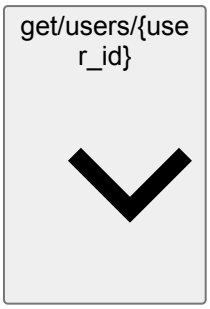
TODO Get a specific user by ID

Retrieves user details based on the provided user ID.

path Parameters

user_id	string
required	Example: 123

Responses



https://civic-interactions-lab.github.io/users/{user_id}

Last updated by ***Nicholas Rucinski***

Test Procedures

Please remove and replace examples where necessary.

Unit Tests

Testing Library

Integration tests

Overview

Acceptance test

Acceptance Test Document

Coverage

Purpose

Unit Tests

Testing Library

1. Pytest
2. Jest
3. VSCode Testing Suite

---

1. Pytest

Used for testing the Flask API **Modules:**

- Pytest-cov
  - Generates test coverage reports in HTML format.
- Pytest-mock
  - Enables the creation of mock objects for simulating dependencies.

Key Features

- Supports fixtures to set up test environments.
- Provides detailed test failure reports with traceback information.
- Enables parameterized tests for efficient testing of multiple scenarios.
- Generates test coverage reports with **pytest-cov** to track untested code.
- Supports mocking API calls and database interactions with **pytest-mock**.

Why it was chosen

- Simple syntax and easy integration with Flask applications.
- Built-in support for fixtures, assertions, and test discovery.
- Comprehensive test coverage capabilities to ensure API stability.
- Widely used in Python testing frameworks.

Running the unit tests

To run the unit tests for the Flask API:

Windows

```
cd webserver
py -m venv .venv
.venv/Scripts/activate
pip install -r requirements.txt
pytest tests -v  # -v for verbose output
coverage run -m pytest
coverage html  # Generates HTML coverage report
```

Mac/Linux

```
cd webserver
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
pytest tests -v  # -v for verbose output
coverage run -m pytest
coverage html  # Generates HTML coverage report
```

2. Jest

Used for unit testing core functionalities within the Visual Studio Code extension.

Modules:

- `jest` – The main testing framework that runs unit tests efficiently in a Node.js environment.
- `ts-jest` – Integrates Jest with TypeScript, enabling seamless testing of TypeScript code.

Key features

- Built-in mocking capabilities for API calls and dependencies.
- Code coverage reports to monitor untested portions of the extension.

- Asynchronous test support for handling Promises and async functions.

Why it was chosen

- Simple and intuitive API, making unit testing easy to write and maintain.
- Optimized for Node.js-based applications, aligning well with VS Code Extensions.
- Good mocking features for testing interactions with VS Code's API without running a full VS Code instance.
- Fast execution compared to other testing frameworks, reducing development cycle time.

Running the unit tests

To run the Jest tests:

```
cd extension
npm install
npx jest [name of test file]  # Run a specific test
npx jest --coverage  # Run all tests with coverage report
```

Test Structure

- Each feature's methods will be encapsulated in a dedicated test file. For example:
  - `suggestions.test.ts` – Contains all related tests for fetching, modifying, and handling suggestions.
- Running a Specific Test File

```
npx jest suggestions.test.ts
```

3. VSCode Testing Suite

Modules:

- @vscode/test-cli – Provides command-line tools for running VS Code extension tests.
- @vscode/test-electron – Runs tests in an Electron environment, simulating VS Code's runtime.

Key Features

- Allows running integration tests in a real VS Code instance.
- Supports launching VS Code in a headless mode for automated testing.
- Provides API hooks to interact with the VS Code extension environment.

Why it was chosen

- Preinstalled when creating VS Code extensions.
- Enables thorough testing of UI interactions and command execution.

Running the unit tests

To run the VS Code extension tests:

```
cd extension
npm install
npm test
```

*Last updated by **Nicholas Rucinski***

Integration tests

Overview

Integration testing ensures that different components of the system work together as expected. These tests are designed to validate the interactions between the user, AI model, and system logic by simulating real-world scenarios using mock objects. External input is provided via mock objects, and results are verified programmatically, eliminating the need for manual data entry.

Testing Approach

- Each test case corresponds to a use case described in the system documentation.
- Mock objects are used to simulate user actions and AI model responses.
- Automated assertions validate expected system behavior.
- Tests are designed to run without manual intervention and produce clear pass/fail results.

Integration Test Cases

1. Context-Aware Code Suggestions

Objective: Ensure that inline code suggestions are generated correctly and logged properly.

Test Steps:

- Simulate a user typing code in the editor.
- Mock the AI model's response with a correct and incorrect suggestion.
- Verify that the system displays the suggestion inline.
- Simulate user accepting/rejecting the suggestion.
- Validate that the system logs the user's choice, decision time, and correctness.

Expected Result:

The suggestion is displayed correctly. The user's decision and response time are logged. If the user accepts an incorrect suggestion, feedback is generated.

2. Inline Questioning about Code

Objective: Ensure users can ask and receive responses for inline questions.

Test Steps:

- Simulate a user highlighting a piece of code and asking a question.
- Mock the AI model's response.
- Verify that the system displays the AI-generated answer.
- Simulate user requesting further clarification.
- Validate that the system logs the question and user interactions.

Expected Result:

The AI-generated response is displayed. User requests for clarification are handled. The system logs user engagement.

3. Decision Time Logging for Code Suggestions

Objective: Ensure the system correctly tracks how long a user takes to accept or reject a suggestion.

Test Steps:

- Simulate the AI model providing a suggestion.
- Start a timer when the suggestion is displayed.
- Simulate user accepting/rejecting the suggestion.
- Verify that the system logs the response time.
- Mock repeated quick and incorrect selections.
- Validate that the system flags the user for disengagement.

Expected Result:

The system logs decision time accurately. Users who consistently make rapid incorrect selections are flagged.

4. Feedback After Selecting a Suggestion

Objective: Ensure users receive feedback on their choice after accepting/rejecting a suggestion.

Test Steps:

- Simulate a user selecting a suggestion.
- Mock the system verifying correctness.
- Verify that the system provides confirmation for correct choices.
- Verify that the system provides feedback for incorrect choices.
- Validate that the system logs the mistake.

Expected Result:

The system correctly provides feedback based on user choice. Mistakes are logged for future analysis.

5. Tracking and Logging User Decisions

Objective: Ensure all user decisions regarding code suggestions are recorded.

Test Steps:

- Simulate multiple user interactions with code suggestions.
- Verify that all accept/reject decisions are logged.
- Validate correctness tracking.
- Mock repeated incorrect choices.
- Verify that recurring mistakes are flagged.

Expected Result:

The system accurately tracks user decisions and logs errors. Recurring mistakes are flagged for instructor review.

6. Generating Learning Reports for Administrators

Objective: Ensure the system compiles user data into meaningful reports.

Test Steps:

- Simulate multiple users interacting with the system.
- Mock the AI model compiling performance reports.
- Verify that reports include percentages of correct/incorrect responses.
- Validate that the report flags topics students struggle with.

Expected Result:

The learning report is correctly generated. Struggle areas are highlighted for instructors.

7. Monitoring Student Progress

Objective: Ensure administrators can track students' learning over time.

Test Steps:

- Mock the AI model generating weekly progress reports.
- Verify that student progress is categorized into improvement, plateau, or decline.
- Validate that recurring student mistakes are flagged.

Expected Result:

The system correctly identifies student progress trends. Instructors can use reports to improve teaching strategies.

8. AI-Generated Quizzes Based on Past Topics

Objective: Ensure the AI model generates quizzes based on student activity.

Test Steps:

- Mock the AI model tracking weekly learning topics.
- Simulate quiz generation based on past struggles.
- Verify that users receive a personalized quiz.
- Validate that the AI model logs correct and incorrect quiz responses.
- Verify that users and administrators can review quiz results.

Expected Result:

The AI model generates personalized quizzes. Users receive feedback on quiz performance. Administrators can analyze student weaknesses.

Last updated by ***Nicholas Rucinski***

Acceptance test

Acceptance Test Document

This spreadsheet provides a detailed record of acceptance tests, offering a comprehensive assessment of the system's functionality and adherence to specified requirements. Each row outlines a test scenario along with its outcomes, observations, and status. This document serves to verify that all aspects of the project have been thoroughly tested and meet the expected standards.

Link to Acceptance Test Spreadsheet

Last updated by ***Nicholas Rucinski***

Coverage

Purpose

A test coverage report provides insights into how much of the codebase is exercised by the tests. This helps identify untested areas, ensuring that all critical system components and functionalities are validated.

Interpreting the Report

- Any areas with low coverage should be reviewed to determine if additional tests are needed.
- Uncovered lines may indicate untested functionality or dead code that should be refactored.
- Ideally, aim for 80-90%+ coverage to ensure thorough testing while balancing practicality.

Click me for Full Screen

404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

Click me for Full Screen

404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

Click me for Full Screen

404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

*Last updated by **Nicholas Rucinski***

Extension Documentation

Click me for Full Screen

404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

*Last updated by **Nicholas Rucinski***

WebServer Documentation

Click me for Full Screen

404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

*Last updated by **Nicholas Rucinski***

Website Documentation

Click me for Full Screen

404

There isn't a GitHub Pages site here.

If you're trying to publish one, read the full documentation to learn how to set up **GitHub Pages** for your repository, organization, or user account.

GitHub Status — @githubstatus

*Last updated by **Nicholas Rucinski***