Fachhochschule Graubünden
University of Applied Sciences

Project: AI-Supported Lightweight Code Editor

AISE501 – AI in Software Engineering

Spring Semester 2026

Framework: `Streamlit` Language: `Python 3.10+` Duration: multi-week project

1 Overview and Goals

The goal of this project is to create a **lightweight code editor** built with Streamlit that integrates AI assistance throughout the development workflow. Your editor will:

1. Display code files in a navigable file structure
2. Include a chat interface to interact with an AI assistant for code suggestions, debugging, and explanations
3. Support reading and editing files
4. Enable internet searches to fetch additional documentation or examples
5. Execute Python code by invoking the Python interpreter and capture runtime errors or debugging messages for further AI feedback
6. Use **Streamlit** as the framework for the user interface
7. Provide a clear architectural plan with defined functions, classes, and their interactions
8. Integrate system prompts and file contexts into AI prompts for complex tasks

Why build this?

This project combines the prompting techniques from the lab exercises with real software engineering. You will design a multi-module application, integrate an LLM API, and handle real-world concerns such as file I/O, error handling, and user interface design. The result is a tool you can actually use for your own development work.

2 Functional Requirements

2.1 File Display & Management

Features

- Browse a project directory and list code files (e.g., `.py`, `.js`, `.html`).
- Open, edit, and save files within the editor.

Classes/Functions

- `FileManager`: Handles directory traversal, file reading, and saving.
- `FileViewer`: UI component in Streamlit to display file contents.

2.2 Chat Interface for AI Interaction

Features

- A chat widget where users can prompt the AI.
- Display conversation history between the user and the AI.
- Allow the AI to suggest code modifications, answer questions, or help with debugging.

Classes/Functions

- `ChatManager`: Manages chat sessions, history, and interfaces with the AI API.
- `SystemPrompter`: Prepares system-level prompts (including file context) to guide AI responses for more complex tasks.

2.3 File I/O Capabilities

Features

- Read and load file contents into the editor.
- Optionally support file uploads/downloads.

Classes/Functions

- Integrated within the `FileManager` and `FileViewer` components.

2.4 Internet Search Integration

Features

- Mechanism to perform internet searches from within the editor.
- Display search results or summaries relevant to the query.

Classes/Functions

- `SearchManager`: Handles search requests, integrates with external APIs, and processes results.

2.5 Code Execution and Debugging

Features

- Invoke the Python interpreter to run code.
- Check for code correctness using the `ast` module.
- Capture output, errors, and exceptions.
- Provide feedback (debugging messages, error logs) to the AI chat.

Classes/Functions

- `ExecutionEngine`: Uses Python's `subprocess` module to execute code.
- `DebugLogger`: Captures and formats output and error messages.

2.6 User Interface (Streamlit)

Features

- A sidebar for file navigation.
- A main pane for code display and editing.
- A dedicated chat window for AI interaction.
- An area for displaying code execution results and debugging messages.

Classes/Functions

- UI functions built with Streamlit widgets (`st.sidebar`, `st.text_area`, `st.button`, etc.).
- Functions to update UI components based on interactions with backend modules.

3 Architectural Plan and Component Interactions

Before diving into implementation, you should draft a small architectural diagram or plan. This plan should include the following modules and their interactions.

Architecture Overview

The application follows a **frontend–backend** split. The Streamlit UI handles user interactions and delegates logic to backend modules. Each backend module is responsible for a single concern.

```
+----------------------------+
|   Streamlit Frontend       |
|  +-----+ +------+ +----+ |
|  |Files | | Code | |Chat| |
|  |Nav   | |Editor| |UI  | |
|  +--+---+ +--+---+ +-+--+ |
+-----|--------|--------|---+
      |        |        |
+-----|--------|--------|---+
|  Backend Modules           |
|  FileManager               |
|  ChatManager               |
|  SystemPrompter            |
|  SearchManager             |
|  ExecutionEngine           |
|  DebugLogger               |
+----------------------------+
```

3.1 Frontend (Streamlit UI)

Components and Interactions

- **Components:** File navigation sidebar, code editor pane, chat interface, output display.
- **Interactions:**
  - User selects a file → UI calls `FileManager` to load file contents.
  - User enters a chat message → UI sends it to `ChatManager`, which adds system prompts (including file context if needed) via `SystemPrompter`.
  - User triggers code execution → UI invokes `ExecutionEngine` to run the code and displays output using `DebugLogger`.

3.2 Backend Modules

`FileManager`

- `list_files()`: Returns a list of code files in the project directory.
- `read_file(file_path)`: Reads and returns the content of a file.
- `save_file(file_path, content)`: Writes content to a file.

ChatManager

- `send_message(message)`: Sends a user message to the AI API.
- `receive_response()`: Returns the AI's response.
- **Interaction:** Integrates with AI services, appending system prompts and file context to user queries.

SystemPrompter

- `generate_prompt(user_message, file_context)`: Builds a complete prompt embedding file contents or contextual information to enhance response accuracy.

SearchManager

- `perform_search(query)`: Executes an internet search.
- `parse_results(raw_results)`: Extracts and formats relevant results.

ExecutionEngine

- `run_code(code)`: Runs the given Python code via `subprocess`.
- `capture_output()`: Returns stdout and stderr from the execution.

DebugLogger

- `log_error(error_message)`: Records an error message.
- `format_debug_output(output)`: Formats output and error messages for display.

3.3 Interaction Flow Examples

End-to-End Flows

1. **File Loading:** User selects a file → `FileManager` retrieves content → UI updates file editor.
2. **Complex Task Prompt:** User requests a code enhancement → `ChatManager` calls `SystemPrompter` to create a prompt including recent file content → AI returns suggestions.
3. **Code Execution:** User executes code → `ExecutionEngine` runs code → `DebugLogger` captures output → UI displays results and sends errors back to `ChatManager` for AI analysis.

4 System Prompts & File Context Integration

4.1 System Prompts

Requirements

- Define default instructions for the AI (e.g., *"You are a code assistant helping to debug Python code."*).
- Allow dynamic system prompts based on the user's current task.
- Include safety or sandboxing instructions when executing code.

4.2 File Context in AI Prompts

Requirements

- When users request complex tasks (e.g., code optimisation or debugging), automatically include relevant file content or snippets.
- Develop a mechanism to summarise or selectively extract file context to avoid overwhelming the AI prompt.
- The `SystemPrompter` module should manage this context inclusion by taking parameters such as file name, recent changes, or highlighted code sections.

Hints & Tips

- Use the prompting techniques from the lab exercises: XML-structured prompts, persona/task/data separation, and structured output where appropriate.
- Keep system prompts concise – long system prompts consume tokens and can dilute the instruction.
- Consider token limits when including file context. For large files, extract only the relevant function or class.
- Use `<code>` tags to clearly separate file content from instructions in the prompt.

Common Mistakes

- Never execute user-uploaded code without sandboxing or at minimum a syntax check via `ast.parse()`.
- Do not pass entire large files as context – this wastes tokens and degrades AI response quality.
- Avoid hardcoding API keys. Use environment variables or a `.env` file (excluded from version control).

5 Development Milestones

Milestone 1 – Project Setup

- Initialise the project and environment.
- Set up Streamlit and install required dependencies.
- Create the project structure with separate modules for each component.

Milestone 2 – UI Development

- Build the sidebar for file navigation.
- Build the main code editor pane.
- Build the chat interface.
- Build the output display area.
- Ensure dynamic updating based on user interactions.

Milestone 3 – Backend Module Development

- Develop the `FileManager` and `FileViewer` for file handling.
- Implement `ChatManager` and `SystemPrompter` for AI interaction.
- Create the `SearchManager` to handle internet searches.
- Build the `ExecutionEngine` and `DebugLogger` for code execution and debugging.

Milestone 4 – Integration and Interaction

- Connect UI actions with backend functions.
- Ensure that file contexts and system prompts are correctly injected into AI queries.
- Validate the complete flow: file loading → chat interaction → code execution → debugging feedback.

title

- Write tests for file operations, AI interactions, search integration, and code execution.
- Gather user feedback, document the architecture and usage, and iterate on improvements.
- Plan for secure code execution environments and sandboxing.

Hints & Tips

- Start with Milestone 1 and 2 to get a working UI skeleton before adding backend logic.
- Use `st.session_state` to persist data across Streamlit reruns.
- Test each module independently before integrating.
- Use Git for version control and commit after completing each milestone.

Fachhochschule Graubünden · Pulvermühlestrasse 57 · 7000 Chur
https://fhgr.ch/cds