# [readme.md](http://readme.md) **Code-Mitra v3.0 - Your Personal AI Coding Assistant**

<!-- You can add a screenshot of your app here -->

Code-Mitra is a desktop application built entirely in Python that serves as a real-time assistant for developers. It functions as an AI-powered coding companion that understands your code, identifies errors, suggests solutions, and answers your questions, all within a single, intuitive interface.

## **What Problem Does It Solve?**

This application addresses common challenges faced by both new and experienced developers, such as:

* Finding and understanding subtle errors in the code.
* Grasping the logic of complex or unfamiliar codebases.
* Exploring alternative and more efficient ways to write code.
* Breaking down high-level project ideas into actionable steps.
* The need to constantly switch between the code editor and a web browser for searching, which disrupts focus and reduces productivity.

**Solution (Code-Mitra):** Code-Mitra provides a centralized solution to these problems. It acts as a personal coding assistant that helps developers in real-time, allowing them to stay focused, save time, and write better code. It makes the entire coding experience smoother and more productive.

## **Key Features**

* **Live Code Editor:** A built-in editor where you can write code, execute it to see the output, and save it to a file.
* **Real-time Analysis:** As soon as you stop typing in the live editor, the application automatically begins to analyze your code.
* **File Monitoring:** Select a project folder, and the app will automatically analyze any file that is saved within that folder.
* **Enhanced AI Explanation:** The AI provides a detailed explanation of your code's logic, suggests alternative approaches, and offers potential optimizations.
* **AI Error Solver:** Identifies errors using pylint, and then uses the Gemini API to explain them and provide corrected code snippets.
* **Dual-Mode "Ask AI":**
  + **Code Generation:** If the editor is empty, ask the AI to generate new code (e.g., "write a function for factorial").
  + **Contextual Q&A:** If the editor contains code, ask any question related to that specific code.
* **Markdown Task Planner:** Write your project idea in a .md file, and the AI will break it down into manageable steps.

## **Technology Stack**

This entire application is built using Python and its standard libraries.

* **Language:** Python 3
* **GUI (User Interface):** Tkinter (with ttk themed widgets)
* **File Monitoring:** watchdog library
* **Code Analysis:** pylint library
* **AI Integration:** Google Gemini API
* **API Communication:** requests library
* **Concurrency:** threading and subprocess modules

## **Setup and Usage**

Follow the steps below to run this application on your local machine.

### **1. Prerequisites**

* **Python 3.8** or newer must be installed.
* You need a **Google Gemini API Key**.

### **2. Installation**

1. Download or git clone this project's code.
2. Open your Terminal or Command Prompt and navigate into the project directory.
3. Run the following command to install the necessary libraries:  
   pip install -r requirements.txt

### **3. API Key Configuration**

1. Open the gemini\_client.py file in the project.
2. In the following line, replace the placeholder with your actual Gemini API key:  
   API\_KEY = "YOUR\_GEMINI\_API\_KEY\_HERE"
3. Save the file.

### **4. Running the Application**

In your Terminal, from the project's root directory, run the following command:

python main\_app.py

This will launch the Code-Mitra GUI window.

### **5. How to Use**

* **Live Coding:** Start typing Python code directly into the "Live Code Editor" tab. Use the "Run Code" button to see the output below and "Save As..." to save your work.
* **File Analysis:** Click "Load & Analyze File" to choose any .py or .md file. Its content will automatically appear in the Live Editor and get analyzed.
* **Code Generation:** Keep the Live Editor empty, type a request like "code for fibonacci series" in the bottom input box, and click "Ask AI".
* **Ask Questions:** With code in the Live Editor, type a related question in the bottom input box and click "Ask AI". The answer will appear in the "Ask AI Results" tab.

requirements.txt

watchdog pylint requests

Documetation of code-mitra  
**Complete Documentation for the Code-Mitra Project**

## Project Purpose (What Problem Does It Solve?)

**The Problem:** Both new and experienced developers face several challenges during coding, such as:

* Finding and debugging subtle errors in their code.
* Understanding the logic of complex or unfamiliar code written by others.
* Thinking of alternative or more efficient ways to accomplish a task.
* Breaking down high-level project ideas into concrete steps for planning.
* Constantly switching context between the code editor and a web browser to search for solutions, which leads to a loss of time and focus.

**The Solution (Code-Mitra):** "Code-Mitra" is an AI-powered desktop application that provides a centralized solution to all these problems. It acts as a personal coding assistant that helps developers in real-time. This allows developers to maintain their focus, save time, and ultimately write better code, making the entire development experience smoother and more productive.

## File-wise Documentation

The application is architected into four main Python files, each with a specific responsibility:

### 1. main\_app.py (The Controller / The Brain)

This file is the core of the application. Its primary role is to act as a bridge between the User Interface (GUI) and the backend logic (analysis, AI communication).

* **Class MainApplication**:
  + \_\_init\_\_: This is the constructor that runs when the application starts. It initializes the GUI, sets up essential variables like the analysis lock (to prevent API spam), and links the window's close button to a safe shutdown procedure.
  + run: This method starts the main Tkinter event loop, which makes the application window visible and responsive to user actions.
  + select\_folder / load\_and\_analyze\_file: These methods provide the user with dialog boxes to select folders for monitoring or individual files for analysis.
  + handle\_live\_code\_analysis / handle\_file\_analysis: These are the trigger functions. They are called whenever a file is saved in a monitored folder or when code is typed in the live editor. They start the analysis process in a new background thread.
  + \_analyze\_python\_file / \_analyze\_markdown\_file: These methods orchestrate the analysis by calling functions from analyzer.py and gemini\_client.py and then sending the results back to the GUI.
  + handle\_ask\_question / \_process\_question: These methods power the "Ask AI" feature. They intelligently determine whether the user wants to generate new code (if the editor is empty) or ask a question about existing code, and then formulate the correct prompt for the AI.
  + run\_live\_code / save\_live\_code / overwrite\_file: These methods handle the logic for the "Run Code", "Save As...", and "Save" buttons in the Live Code Editor.
  + on\_closing: This method ensures that background processes, like the watchdog file monitor, are safely stopped when the application window is closed.
* **Class FileChangeHandler**:
  + This is a helper class for the watchdog library. Its on\_modified method is automatically called whenever a file is saved in the monitored folder, which in turn triggers the analysis logic in the MainApplication.

### 2. gui.py (The User Interface / The Face)

This file is solely responsible for creating and managing the entire graphical user interface (GUI) of the application using the Tkinter library.

* **Class AppGUI**:
  + \_\_init\_\_: Creates the main application window, sets its title and size, and calls helper methods to configure styles and create all the UI elements.
  + \_configure\_styles: This method sets up the visual theme for the application. It configures the colors, fonts, and padding for all ttk widgets (Buttons, Tabs, Labels) to give the application a modern, dark-theme look.
  + \_create\_widgets: This is where all the visual components of the application are built and placed on the screen. This includes the top buttons, the main tabbed interface, the "Live Code Editor" with its output area, and the "Ask AI" section at the bottom.
  + on\_key\_release / trigger\_live\_analysis: These methods implement the automatic analysis feature in the live editor. They use a technique called "debouncing" to wait until the user has stopped typing for 1.5 seconds before starting an analysis, which prevents excessive API calls.
  + load\_content\_to\_editor: When a file is loaded, this function clears the Live Editor and populates it with the new content, automatically switching focus to that tab.
  + update\_display / update\_status / update\_folder\_label: These are crucial **thread-safe** functions. Since Tkinter does not allow direct GUI modifications from background threads, these functions use self.after(0, ...) to safely schedule the UI updates on the main GUI thread, preventing the application from crashing.

### 3. analyzer.py (The Local Analyst)

This file handles all analysis tasks that do not require an AI. Its purpose is to perform local checks and read data from the file system.

* **run\_pylint(file\_path)**:
  + **Purpose:** To run the pylint tool on a given Python file.
  + **How it works:** It uses Python's subprocess module to execute the pylint command as if you were typing it in a terminal. It captures the complete report (errors and style issues) and returns it as a text string to main\_app.py.
* **read\_file\_content(file\_path)**:
  + **Purpose:** To safely read the entire content of any given file.
  + **How it works:** It uses standard Python file handling (open(...)) with encoding='utf-8' to ensure that special characters are read correctly. It returns the file's content as a single string.

### 4. gemini\_client.py (The AI Communicator)

This file is the application's gateway to the artificial intelligence capabilities of the Google Gemini API. Isolating this logic makes the application cleaner and easier to manage.

* **query\_gemini(prompt)**:
  + **Purpose:** To ask any question (prompt) to the AI and get a response.
  + **How it works:**
    1. **API Key Check:** It first verifies that a valid API key has been provided.
    2. **Request Creation:** It uses the requests library to send an HTTP POST request to the Gemini API endpoint. The user's prompt is sent in the body of this request.
    3. **Error Handling (Exponential Backoff):** This is the most critical feature of this function. If the API responds with an **Error 429 (Too Many Requests)**, the function doesn't give up immediately. It waits for a short period (initially 1 second) and tries again. If it fails again, it doubles the waiting time (2 seconds, then 4, and so on). This gives the API server time to become available again and prevents the application from crashing.
    4. **Response Parsing:** Once a successful response is received from the API (in JSON format), this function carefully extracts the relevant text part of the answer.
    5. Finally, it returns the clean text response to main\_app.py.