AI Bug Fix Advisor

An Intelligent Python Debugging Assistant Powered by Groq LLMs

Project Overview

The **AI Bug Fix Advisor** is a state-of-the-art Python web application designed to fundamentally change the way developers approach and resolve code errors. Leveraging the unparalleled speed of **Groq's Language Processing Units (LPUs)**, this tool provides instant, comprehensive, and context-aware debugging support. The primary goal of the project is to eliminate the tedious, time-consuming process of manually searching for solutions online by providing a definitive set of three distinct, high-quality fixes for any Python bug.

The system is engineered to accept raw Python source code and the corresponding error traceback—the two pieces of information an LLM needs to diagnose a problem effectively. This input is meticulously processed by the error_parser.py and utils/chunk_processor.py components to extract the exact error type, message, and crucial code context. This surgical analysis then feeds into the **Groq API**, managed by the groq_handler.py module, which is carefully prompted to generate a response that adheres to a strict four-part structure: a root cause explanation, followed by three unique solution approaches.

This approach provides value across the entire spectrum of developers, from students learning fundamental error handling to senior engineers requiring rapid, robust, and best-practice solutions for production environments. The intuitive user interface, built with **Gradio**, ensures that this powerful AI is accessible and easy to use, making the debugging process efficient and even educational.

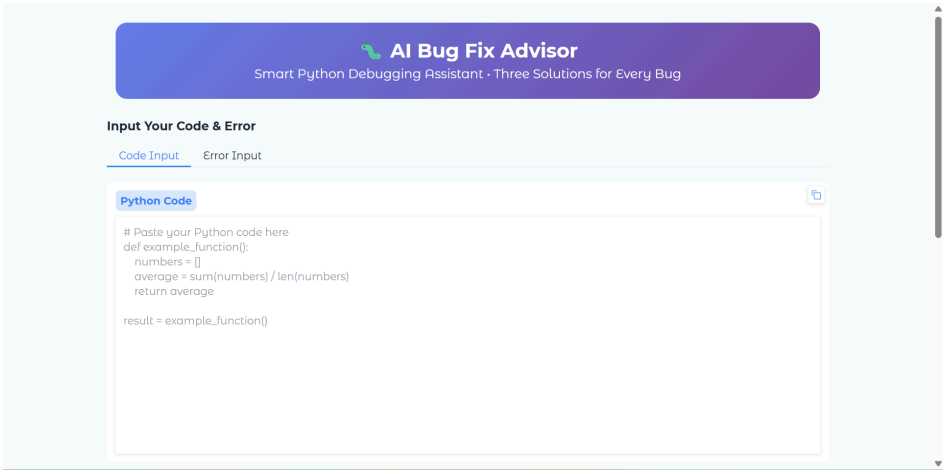


Figure 1: The opening page of the AI Bug Fix Advisor

Scenario 1: Quick Bug Resolution for Professionals

Corporate developers, DevOps engineers, and back-end specialists often face complex, intermittent errors in large-scale production codebases. In these high-stakes scenarios, the need is not just for a fix, but for the most appropriate fix under pressure.

When a professional encounters an error, such as a subtle requests.exceptions.JSONDecodeError from an unstable external API or an unexpected AttributeError deep within object serialization logic, they can use the AI Bug Fix Advisor to save critical development time. The system's output is tailored to offer choices that reflect real-world engineering decisions:

- **Solution 1 (Simple Fix):** This provides the quickest path to resolution, often a one or two-line correction like a missing argument, a simple conditional check, or a direct variable reassignment. This is the fix for when immediate deployment is necessary.
- **Solution 2 (Try-Except Handling):** This focuses on **defensive programming**. The system provides a robust code block that specifically catches the precise exception (e.g., catching requests.exceptions.JSONDecodeError rather than a generic Exception), handles the failure gracefully, and often includes proper logging or fallback logic, ensuring production stability.
- **Solution 3 (Alternative Approach):** This offers a method to refactor or restructure the problematic code using **best practices**. It might suggest using Python's built-in csv module instead of manual string splitting, using context managers, or implementing data validation (like Pydantic) to preemptively catch issues before execution, thus improving long-term code health.

The professional receives three viable paths forward, allowing them to choose a fix based on project standards, code review requirements, and time constraints.

Scenario 2: Learning and Best Practices for Beginners

The debugging experience for a beginner or a student is fundamentally an educational one. Common errors like ZeroDivisionError, IndexError, or TypeError are frustrating, but they present key learning opportunities. The AI Bug Fix Advisor transforms these roadblocks into structured lessons.

When a beginner pastes code that generates a TypeError (e.g., trying to concatenate a string and an integer), the Advisor first provides a clear, concise **ERROR EXPLANATION** explaining the underlying principle—that Python is strictly typed in that context. The three solutions then act as a graduated curriculum:

- **Solution 1 (Simple Fix):** The system provides the direct fix, typically converting the integer to a string using str(). This satisfies the immediate need to make the code run and demonstrates the simplest solution.
- **Solution 2 (Try-Except Handling):** This introduces the student to **proper error management**. It shows how to wrap the problematic section in a try...except block, specifically catching the TypeError, and providing a clean message to the user, thereby teaching the concept of fault tolerance.
- **Solution 3 (Alternative Approach):** This often demonstrates **best coding practices**. For instance, it might show how to use f-strings for concatenation, or how to use isinstance() to check for the correct type before attempting the operation, instructing the student on how to write code that avoids the error entirely.

By providing these three perspectives, the Advisor moves beyond being just a fix-it tool and serves as a personalized programming mentor, helping beginners understand the why behind the fix and guiding them toward cleaner, more robust code development.

Architecture

The AI Bug Fix Advisor is built upon a modular and highly efficient architecture designed to maximize processing speed while maintaining output quality and flexibility. This separation of concerns ensures that components can be easily maintained and upgraded.

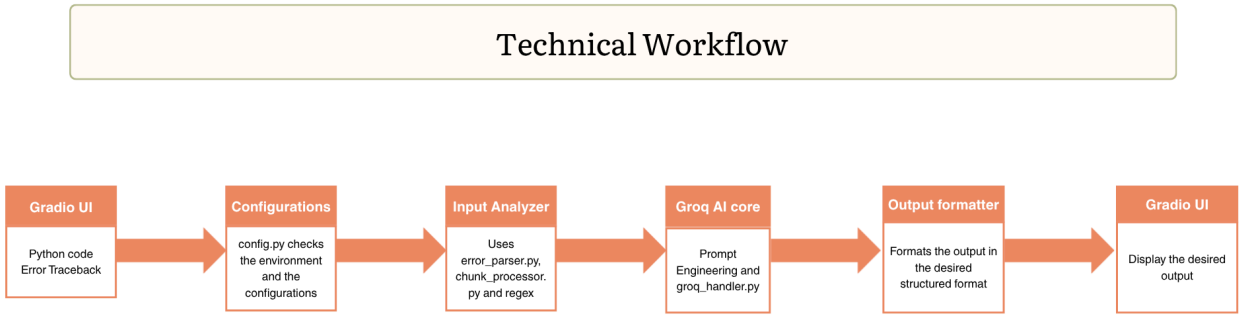


Figure 2: Technical Workflow of the project

Core Components

The system is fundamentally divided into four operational areas: Configuration, Input Analysis, AI Core, and Output Handling.

a. Configuration Manager (config.py and .env)

The **Configuration Manager** serves as the initial layer, responsible for establishing all environment-specific and operational parameters. The config.py file uses the python-dotenv library to securely import variables from the .env file, including the critical **GROQ_API_KEY**, the desired MODEL_NAME (e.g., llama-3.1-70b-versatile), and various operational settings like **CHUNK_SIZE** and TEMPERATURE. This separation allows developers to manage API keys and adjust processing logic without altering the core codebase. Crucially, it includes validation logic to check for missing API keys or unsupported models, defaulting to stable settings to guarantee a successful application launch.

b. Input Analysis and Pre-processing (error_parser.py and utils/chunk_processor.py)

This layer is responsible for intelligently dissecting the user's raw input. The **ErrorParser** initiates the process by analyzing the traceback string. It relies on the **ChunkProcessor** to perform two key functions:

- **Error Detail Extraction:** The ChunkProcessor uses regular expressions to isolate the specific **error type** (e.g., TypeError), the **error message**, and the precise **line number** where the execution failed.
- **Contextual Sourcing:** Using the identified line number, the get_error_context method extracts a focused window of code, pulling several lines before and after the failure point (as defined by CONTEXT_LINES in config.py). This is a critical step, as it provides the LLM with the minimal, most relevant information required for a non-speculative fix.

By performing this pre-processing, the system avoids sending unnecessarily long code snippets to the LLM and directs the AI's attention to the exact area of concern.

c. AI Core and Prompt Engineering (groq_handler.py)

The **Groq Bug Fixer** is the intelligence hub. It uses the groq SDK to establish a connection with Groq's low-latency LPU platform. Its primary function is the meticulous construction of the LLM prompt via the _create_enhanced_prompt method. This prompt is deliberately verbose and highly structured, containing the following directives:

1. The full, raw code and traceback.
2. Explicit, non-negotiable section headers: ERROR EXPLANATION:, SOLUTION 1 (SIMPLE FIX):, SOLUTION 2 (TRY-EXCEPT HANDLING):, and SOLUTION 3 (ALTERNATIVE APPROACH):.
3. A list of **CRITICAL REQUIREMENTS** that mandate the differentiation between the three solutions, ensuring they are not mere variations but fundamentally distinct approaches (simple, defensive, refactored).

The _call_groq_api function then executes the request, utilizing a slightly higher temperature to encourage the creative diversity needed for three unique fixes.

d. Output Handling and Formatting (utils/formatters.py)

This final layer converts the raw LLM text into the polished, structured output displayed to the user. The **OutputFormatter** implements a **multi-strategy parsing approach** (regex and line-based fallback) to robustly extract the content for the four mandatory sections, regardless of minor stylistic deviations by the LLM. It then uses the format_final_output method to wrap this content in easy-to-read Markdown, ensuring a clean, professional presentation in the Gradio interface.

Project Flow

The application flow is a linear, sequential process designed for minimal user waiting time, leveraging the speed of the Groq API.

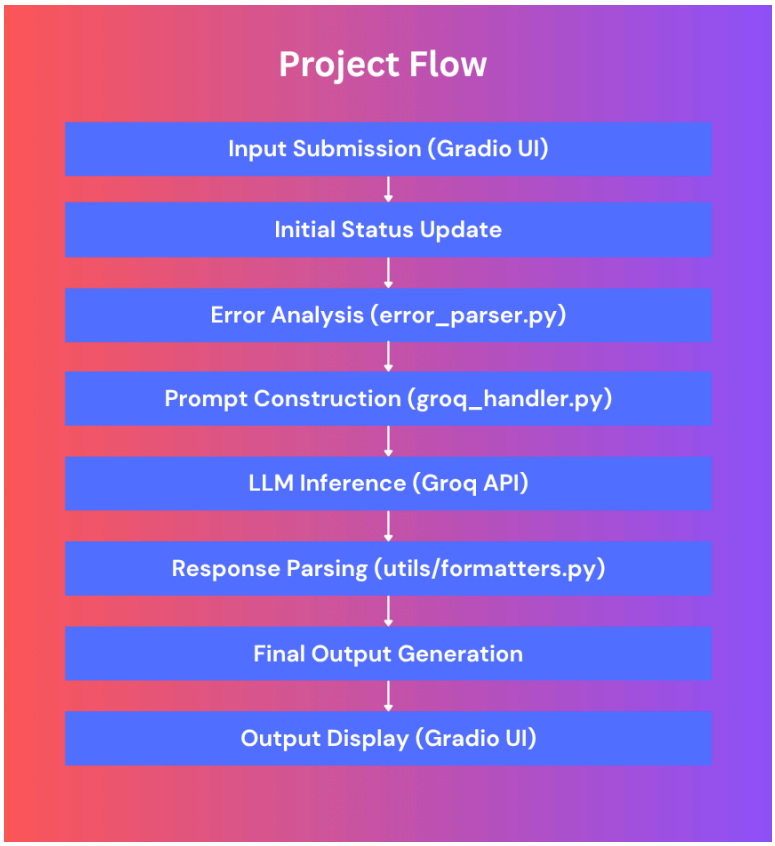


Figure 3: Project flow for AI Bug Fix Advisor

1. **Input Submission (Gradio UI):** The user enters the Python code and the error traceback into the respective tabs in the Gradio interface (app.py) and clicks the **"Analyze & Generate Fixes"** button.
2. **Initial Status Update:** The main analyze_code function in app.py immediately yields a status message (e.g., "Analyzing your code...") to the UI, providing instantaneous feedback.
3. **Error Analysis (error_parser.py):** The code and traceback are passed to the ErrorParser. This module determines the error line and uses the ChunkProcessor to retrieve the contextual code snippet. All extracted details are packaged into an analysis dictionary.
4. **Prompt Construction (groq_handler.py):** The analysis dictionary is passed to the GroqBugFixer. It uses the code, traceback, and extracted context to generate the highly specific, multi-solution prompt template, ensuring adherence to the strict formatting requirements.

5. **LLM Inference (Groq API):** The groq_handler.py calls the Groq API. The LLM processes the prompt at high speed, generating a raw text response containing the explanation and the three required solutions, each segmented by its respective heading.
6. **Response Parsing (utils/formatters.py):** The raw text is immediately passed to the OutputFormatter. This module uses its robust parsing logic to extract the four distinct solution components into a structured Python dictionary.
7. **Final Output Generation:** The OutputFormatter then takes the structured data and formats it into the final Markdown string, applying the necessary aesthetic elements (emojis, bolding, headings).
8. **Output Display (Gradio UI):** The analyze_code function yields the final formatted Markdown string, which is then rendered dynamically in the Gradio output box for the user to view.

To Accomplish This Project, You Must Complete the Following Activities:

The development of the AI Bug Fix Advisor requires the completion of several core activities, focusing on environment preparation, robust AI integration, complex logic implementation, and frontend delivery.

- **Install and configure all required libraries and APIs.**
    - Set up the virtual environment and install dependencies listed in requirements.txt: **gradio**, **groq**, and **python-dotenv**.
    - Create and populate the **.env** file with the **GROQ_API_KEY** and all processing constants (CHUNK_SIZE, MODEL_NAME).
    - Develop **config.py** to load and validate these settings, ensuring a stable application foundation.
- **Initialize the Groq LLM Client and Core API Logic.**
    - Implement the **GroqBugFixer** class in **groq_handler.py** to initialize the groq.Client using the validated API key.
    - Define the **_call_groq_api** method, configuring the high-speed LLM model and setting a high temperature to promote diverse solutions.
- **Develop the Multi-Solution Generation and Processing Functions.**
    - Create the **ErrorParser** and **ChunkProcessor** to analyze the raw traceback, extract the error line number, and generate the crucial contextual code snippet.
    - Engineer the **_create_enhanced_prompt** function to meticulously enforce the **three-solution output structure** (Simple Fix, Try-Except, Alternative Approach) using mandatory section headers and critical constraints.
    - Develop the **OutputFormatter** to implement robust multi-strategy parsing (regex and line-based fallback) of the raw LLM response, ensuring all four required sections are successfully extracted.
- **Build the Gradio User Interface.**
    - Design and implement the frontend in **app.py** using **gr.Blocks**, including custom CSS for the professional, code-friendly theme.
    - Create the two-column layout with the tabbed input (Code and Traceback textboxes) and the output display (Markdown component).
    - Define the **analyze_code** generator function that orchestrates the entire pipeline, from error parsing to final output formatting.
- **Orchestrate Application Flow and Launch the Interface.**
    - Bind the "Analyze & Generate Fixes" button to the main orchestration function.
    - Implement and connect the "Quick Examples" buttons to pre-populate the inputs for testing.
    - Configure the main execution block (if __name__ == "__main__":) to launch the complete application using **demo.launch()**, making the Bug Fix Advisor accessible via a web browser.

Prior Knowledge

To effectively understand, maintain, or extend the **AI Bug Fix Advisor** project, you should have a solid working knowledge of the following technologies and concepts, which form the core of its architecture:

You should have an understanding of:

1. **Groq API and High-Speed LLM Inference** Groq provides an advanced platform for lightning-fast LLM inference using proprietary LPUs. Knowledge of interacting with the Groq Python SDK (groq), configuring models like Llama, and understanding the benefits of low-latency API calls is essential for optimizing the **groq_handler.py** module.

   Official Documentation: https://console.groq.com/docs/overview

2. **Gradio Framework** Gradio is the Python-based framework used to build the interactive web frontend for the application. Understanding how to manage components (gr.Textbox, gr.Button, gr.Markdown), handle layout (gr.Blocks, gr.Row), manage state, and bind component interactions to backend Python functions is crucial for working with **app.py**.

   Official Tutorials: https://www.gradio.app/docs

3. **Prompt Engineering (Structured Output)** Prompt engineering is the process of designing and structuring input prompts for the LLM to elicit a specific, desired output format. Since **groq_handler.py** relies on a strict template demanding three distinct solutions and four mandatory headings, knowledge of crafting clear constraints, templates, and mandatory headers is vital for ensuring the LLM's response can be reliably parsed.

   Learning Resources:
   https://developers.google.com/machine-learning/resources/prompt-eng

4. **Regular Expressions (Regex)** This project heavily relies on regular expressions for two critical components: parsing and formatting. The **utils/chunk_processor.py** uses regex to surgically extract the error type and line number from the raw traceback, while **utils/formatters.py** uses multiple regex patterns to robustly parse the structured output from the LLM, making reliable extraction possible.

   Official Documentation: https://docs.python.org/3/library/re.html

5. **Python Programming and Environment Management** Core Python skills are essential for integrating all components, handling data flow, and implementing the application logic. Furthermore, familiarity with the **python-dotenv** library is required for the secure management of environment variables (like the GROQ_API_KEY) within the **config.py** module, ensuring sensitive credentials are not exposed.

   Official Documentation: https://docs.python.org/3/library/

Project Structure

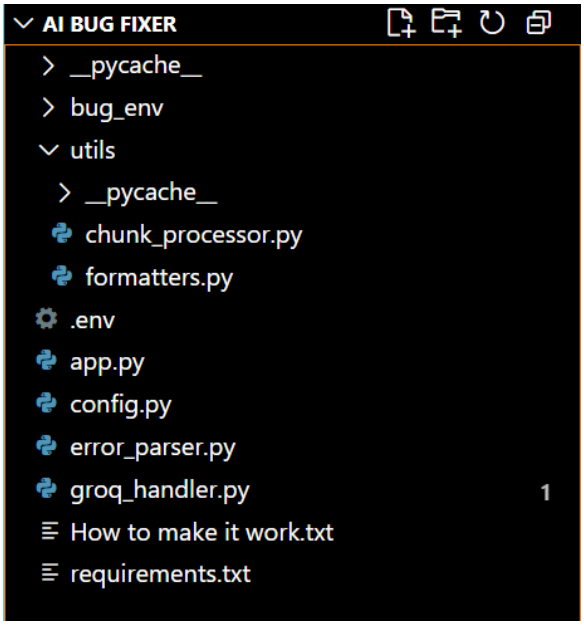


Figure 4: Folder structure of AI Bug Fixer application in VSCode interface

Milestone 1: Project Setup and Configuration

Milestone: Library Installation and Secure Configuration

This foundational milestone ensures all necessary Python libraries are installed and the application environment is securely configured. It prepares the project for high-speed LLM interaction via Groq and frontend delivery via Gradio.

Option 1: Using requirements.txt for Library Installation

This approach ensures reproducibility and easy setup for all users by installing all required libraries at once from a requirements.txt file.

Steps

1. **Ensure requirements.txt Exists** The project folder should contain a requirements.txt file with the following content:
    - gradio>=4.0.0
    - groq>=0.3.0
    - python-dotenv>=1.0.0

Figure 5: requirements.txt in VSCode interface

2. **Install All Libraries** Run the following command in your project directory (after creating and activating a virtual environment) to install all dependencies at once:

Figure 6: Installing requirements.txt

Explanation of Each Library

- **gradio**: Provides a simple, fast way to build the interactive web user interface (app.py), enabling developers to easily input code and tracebacks and display the structured AI output.
- **groq**: The official SDK used to connect to Groq's LPU platform. This is critical for generating instant, low-latency, and high-quality solutions from the LLM.
- **python-dotenv**: Used by config.py to securely load sensitive environment variables, such as the GROQ_API_KEY, from the hidden .env file, ensuring credentials are never hardcoded.

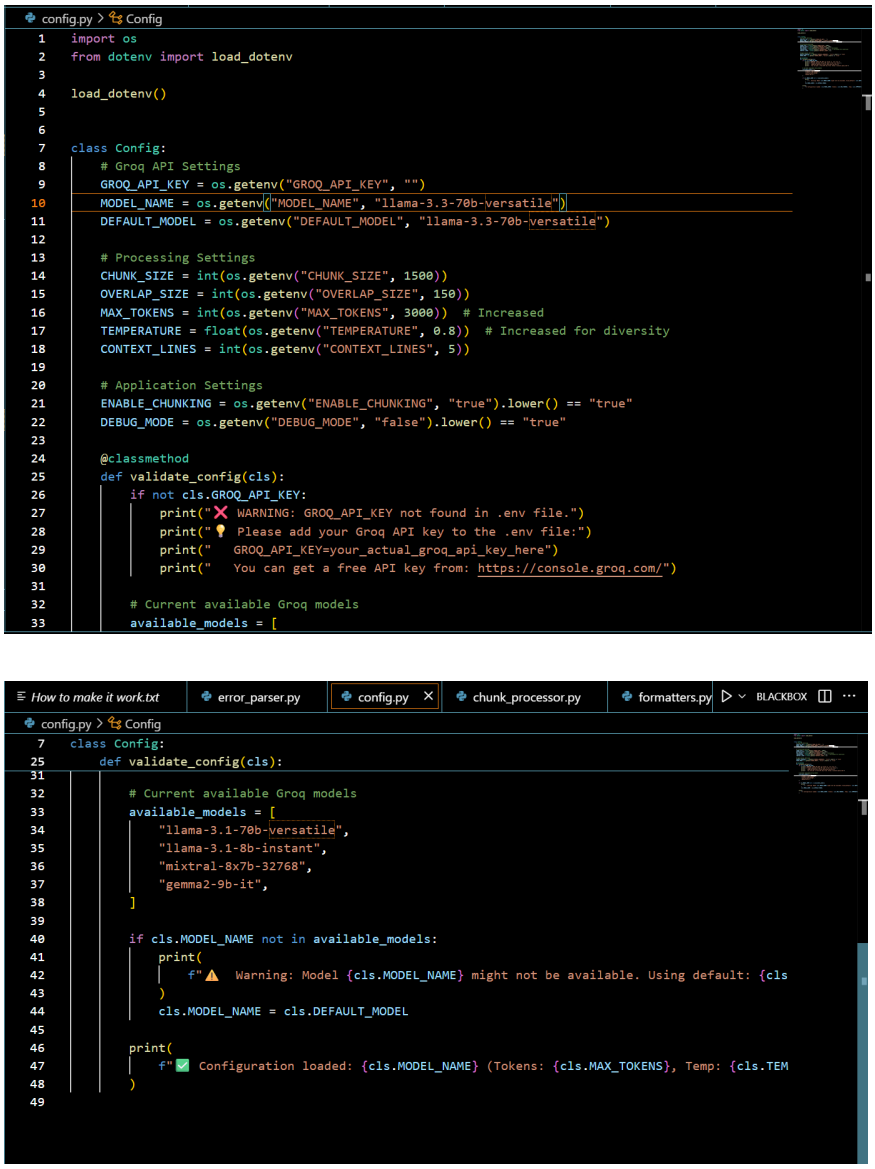
Activity 1: Loading Environment Variables and Validating Configuration (config.py)

This activity establishes the application's configuration hub, ensuring all settings and API keys are loaded securely and validated before the core logic executes.

Explanation

- **Secure Loading**: The config.py module uses load_dotenv() to read API keys and variables (like GROQ_API_KEY, MODEL_NAME, CHUNK_SIZE) from the .env file, preventing exposure of sensitive credentials.
- **Validation**: The Config.validate_config class method performs crucial integrity checks. It verifies that the **GROQ_API_KEY** is present and prints a warning if it's missing. It also checks if the specified MODEL_NAME is an available Groq model, defaulting to a supported model if necessary, thus guaranteeing a reliable startup.

Code snippet:



```python
import os
from dotenv import load_dotenv

load_dotenv()


class Config:
    # Groq API Settings
    GROQ_API_KEY = os.getenv("GROQ_API_KEY", "")
    MODEL_NAME = os.getenv("MODEL_NAME", "llama-3.3-70b-versatile")
    DEFAULT_MODEL = os.getenv("DEFAULT_MODEL", "llama-3.3-70b-versatile")

    # Processing Settings
    CHUNK_SIZE = int(os.getenv("CHUNK_SIZE", 1500))
    OVERLAP_SIZE = int(os.getenv("OVERLAP_SIZE", 150))
    MAX_TOKENS = int(os.getenv("MAX_TOKENS", 3000))  # Increased
    TEMPERATURE = float(os.getenv("TEMPERATURE", 0.8))  # Increased for diversity
    CONTEXT_LINES = int(os.getenv("CONTEXT_LINES", 5))

    # Application Settings
    ENABLE_CHUNKING = os.getenv("ENABLE_CHUNKING", "true").lower() == "true"
    DEBUG_MODE = os.getenv("DEBUG_MODE", "false").lower() == "true"

    @classmethod
    def validate_config(cls):
        if not cls.GROQ_API_KEY:
            print("❌ WARNING: GROQ_API_KEY not found in .env file.")
            print("💡 Please add your Groq API key to the .env file:")
            print("   GROQ_API_KEY=your_actual_groq_api_key_here")
            print("   You can get a free API key from: https://console.groq.com/")

        # Current available Groq models
        available_models = [
```



```python
class Config:
    def validate_config(cls):

        # Current available Groq models
        available_models = [
            "llama-3.1-70b-versatile",
            "llama-3.1-8b-instant",
            "mixtral-8x7b-32768",
            "gemma2-9b-it",
        ]

        if cls.MODEL_NAME not in available_models:
            print(
                f"⚠️  Warning: Model {cls.MODEL_NAME} might not be available. Using default: {cls
            )
            cls.MODEL_NAME = cls.DEFAULT_MODEL

        print(
            f"✅ Configuration loaded: {cls.MODEL_NAME} (Tokens: {cls.MAX_TOKENS}, Temp: {cls.TEM
        )
```

Figure 7: Code snippets of config.py

Milestone 2: Input Processing and AI Core Initialization

Milestone: Analyzing Input and Establishing Groq Connection

This milestone is dedicated to creating the intelligence layer of the application. It focuses on taking the raw user input (code and traceback) and surgically extracting the precise information the LLM needs, while also configuring the high-speed connection to the Groq API.

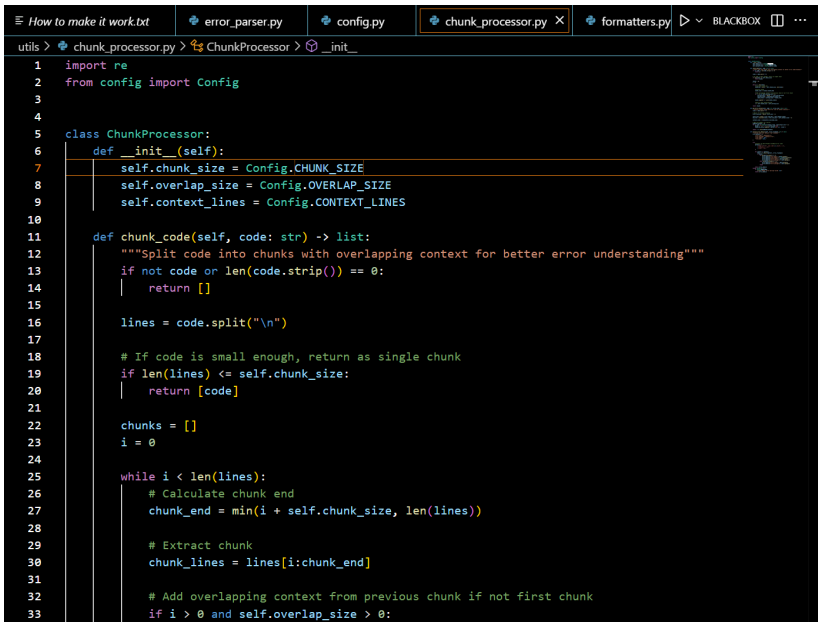
Activity 1: Developing Contextual Code Processing Utilities (utils/chunk_processor.py)

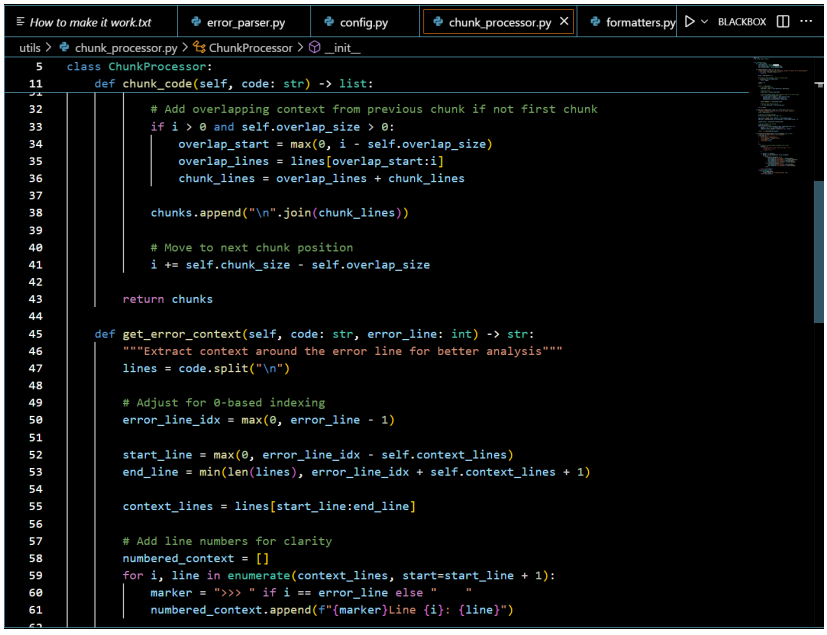
This activity ensures that the application doesn't simply dump massive amounts of code onto the LLM, but instead provides the exact context necessary for an accurate fix.

Explanation

- **Surgical Extraction (Regex)**: The extract_error_details method uses **Regular Expressions (regex)** to parse the raw traceback string. This is a critical step that isolates three key pieces of data: the precise **error type** (e.g., ValueError), the human-readable **error message**, and most importantly, the exact **line number** where the failure occurred.
- **Contextual Sourcing**: Once the failure line is identified, the **get_error_context** function takes over. It extracts a small, focused window of code (defined by CONTEXT_LINES in config.py) surrounding the error. This is then formatted with line numbers and a distinct marker (>>>) pointing to the failure line, providing the LLM with a highly relevant, localized view of the bug. This preparation drastically reduces latency and improves the accuracy of the generated fixes.
- **Chunking Logic**: The **chunk_code** method is implemented as a safeguard for very large codebases. It ensures that if the input code exceeds the configured CHUNK_SIZE, it can be broken down into manageable segments, with strategic **overlap** to maintain continuity of context across the segments.

Code snippet (Extracting Details):





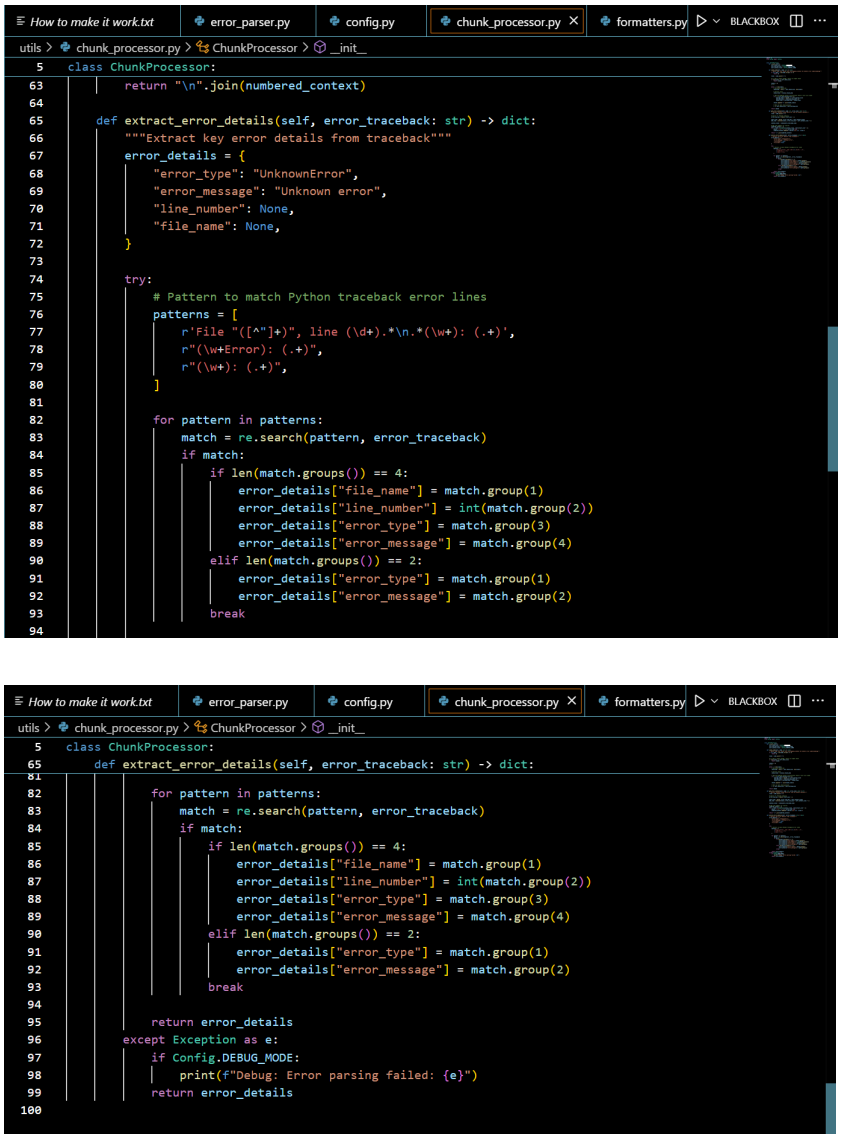


Figure 8: Developing Contextual Code Processing Utilities

Activity 2: Initializing the Groq LLM Client and Error Parser

This activity connects the analysis layer to the AI layer, setting the stage for prompt generation.

Explanation

- **Error Parser Orchestration (error_parser.py)**: The ErrorParser class is created to serve as the control center for input analysis. Its primary method, **analyze_error**, calls the necessary functions from the ChunkProcessor, collects the error details and the

contextual code snippet, and packages everything into a single dictionary ready for the Groq handler.

- **Groq Client Initialization (groq_handler.py)**: The **GroqBugFixer** class is initialized in the groq_handler.py module. It uses the securely loaded GROQ_API_KEY from the Config class to create the **groq.Client** instance. This step formally establishes the application's connection to the high-speed LLM service, utilizing the chosen model (e.g., llama-3.1-70b-versatile). This initialization prepares the AI engine to receive the finely tuned prompt in the next milestone.

Code snippet:

Figure 9: Groq Client Initialization

Milestone 3: Logic Development and Formatting

Milestone: Engineering the Multi-Solution Prompt and Robust Output Parsing

This critical milestone implements the core intelligence of the application: generating the structured, differentiated solutions and reliably extracting them from the LLM's raw text response. It ensures that the output is not only accurate but also consistently formatted.

Activity 1: Engineer the Multi-Solution Prompt Template (groq_handler.py)

This activity involves writing the precise instructions that guide the LLM to produce the required, differentiated output.

Explanation

- **Prompt Architecture**: The private method **_create_enhanced_prompt** is a masterclass in prompt control. It doesn't just ask the LLM for a fix; it constructs a directive containing the raw code, the traceback, the analysis, and a set of non-negotiable structural requirements.
- **Mandatory Structure**: The prompt includes explicit, capital-letter section headers: ERROR EXPLANATION:, SOLUTION 1 (SIMPLE FIX):, SOLUTION 2 (TRY-EXCEPT HANDLING):, and SOLUTION 3 (ALTERNATIVE APPROACH):. This guides the LLM to segment its response clearly.
- **Critical Constraints**: The prompt includes a list of **CRITICAL REQUIREMENTS** (e.g., "All three solutions MUST be different from each other," "Solution 2 must include proper try-except error handling"). These constraints are essential for achieving the value proposition of the app, forcing the LLM to output a simple patch, a defensive solution, and a refactored approach, ensuring true diversity.

Code snippet:

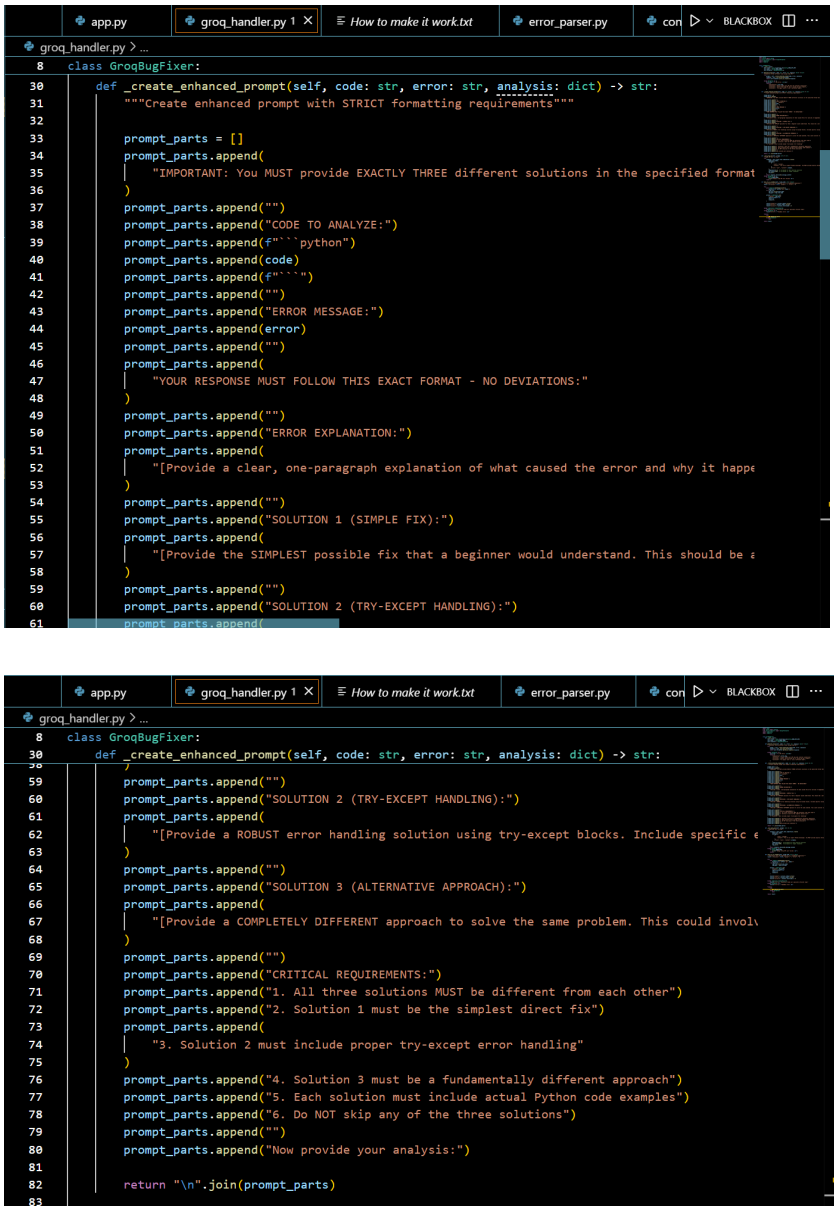


Figure 10: Engineer the Multi-Solution Prompt Template

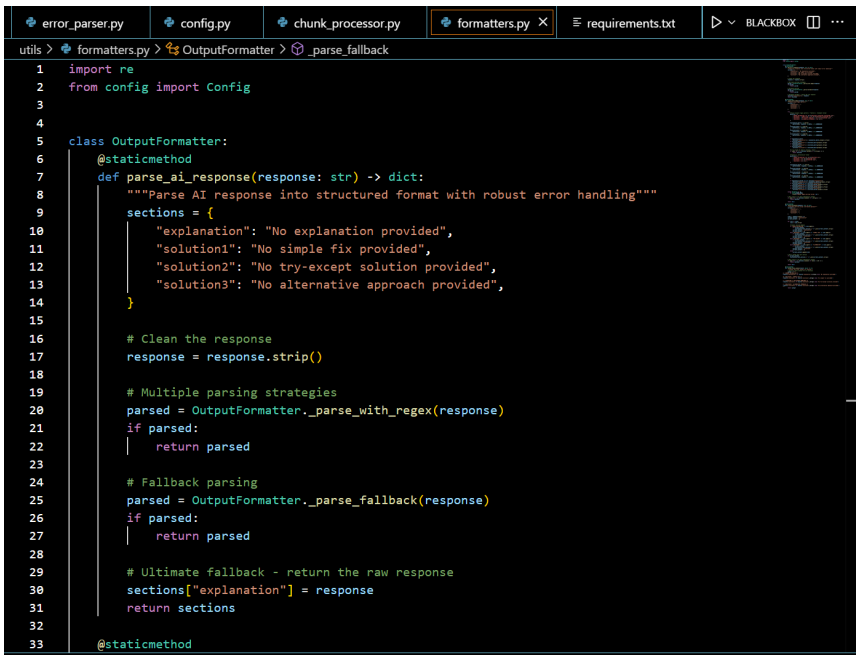
Activity 2: Develop Robust Output Formatting Utilities (utils/formatters.py)

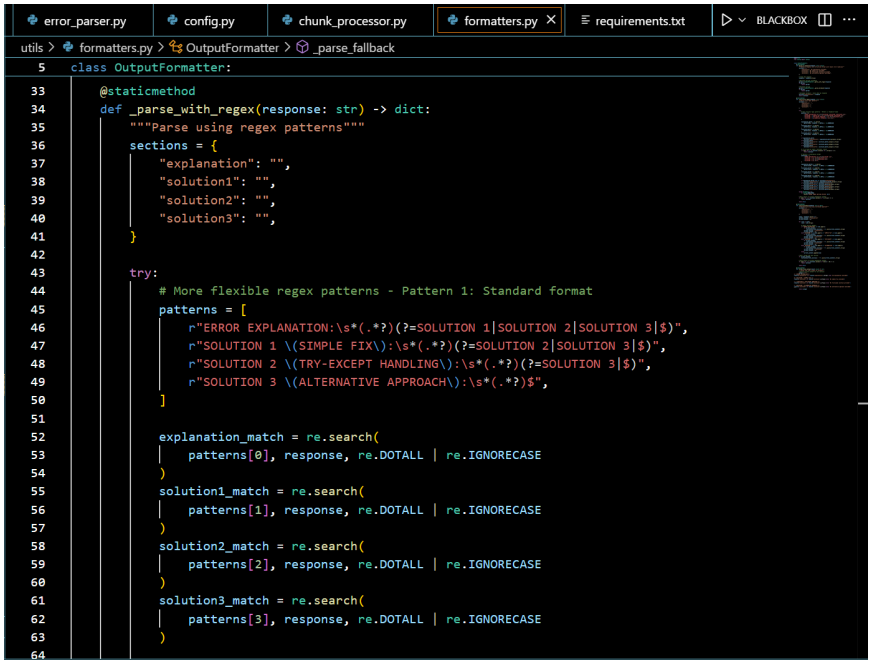
This activity addresses the reality that LLMs, even with strict prompts, sometimes deviate slightly from the required output format. A robust parser is needed to guarantee successful data extraction.

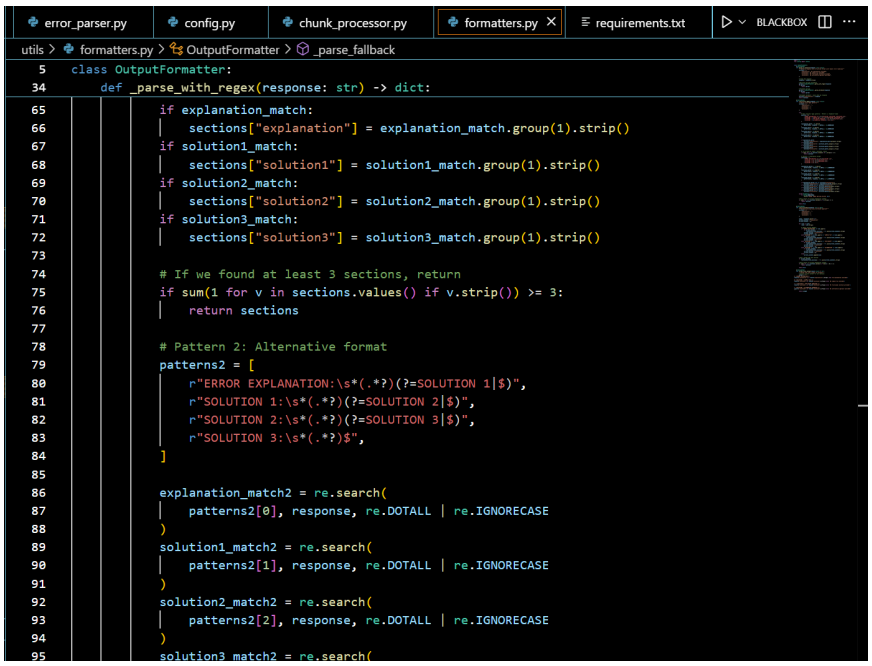
Explanation

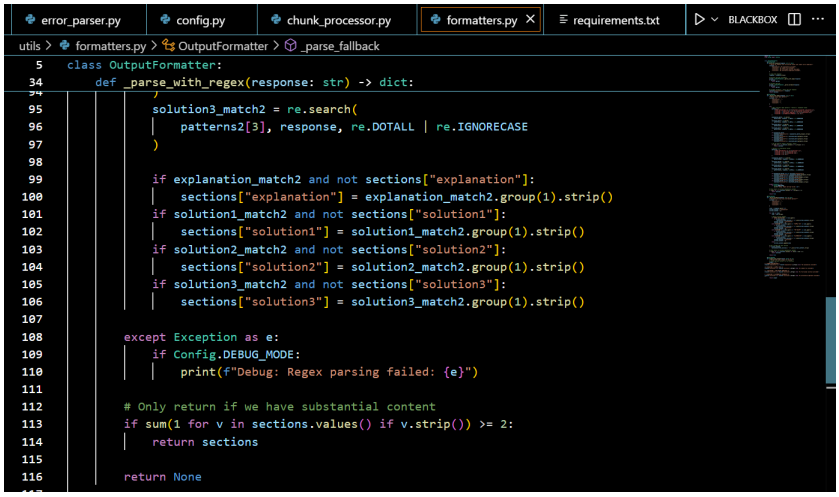
- **Multi-Strategy Parsing**: The **parse_ai_response** method implements a two-tier defense mechanism. It first attempts extraction using **flexible Regular Expressions**. If the regex patterns fail to capture all four sections due to minor LLM variations, it falls back to a **line-based parsing** approach that detects section keywords (e.g., "SOLUTION 2," "TRY-EXCEPT") to delineate the content boundaries. This guarantees reliable extraction into a structured dictionary.
- **Final Presentation**: The **format_final_output** static method takes the extracted dictionary and formats it into the clean, aesthetic Markdown output displayed in the Gradio UI. It adds visual cues such as the **emojis** (🐛, 🔧, 🛡️, 💡) and clear headings, significantly enhancing user experience by making the complex solutions immediately digestible.

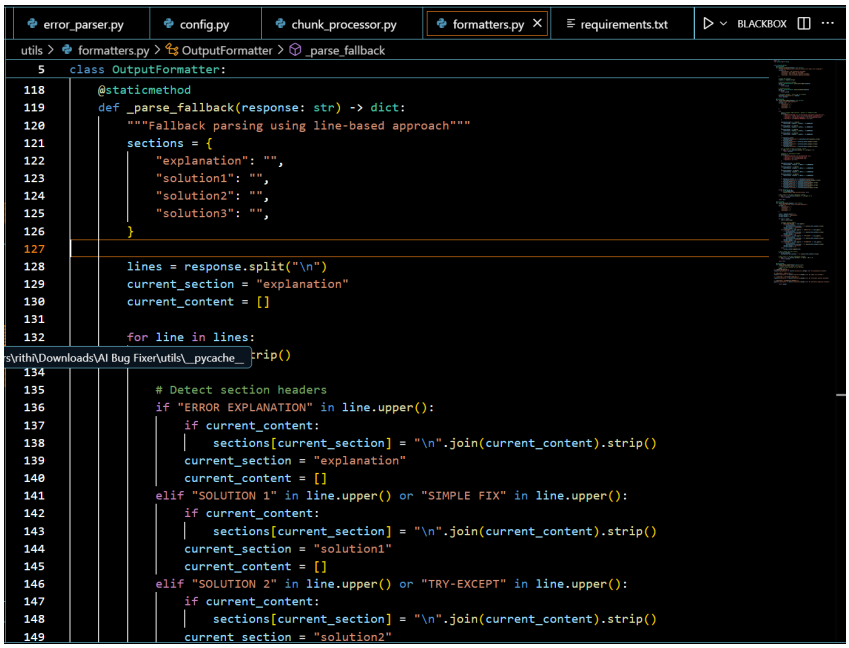
Code snippet (Output Parsing Logic):

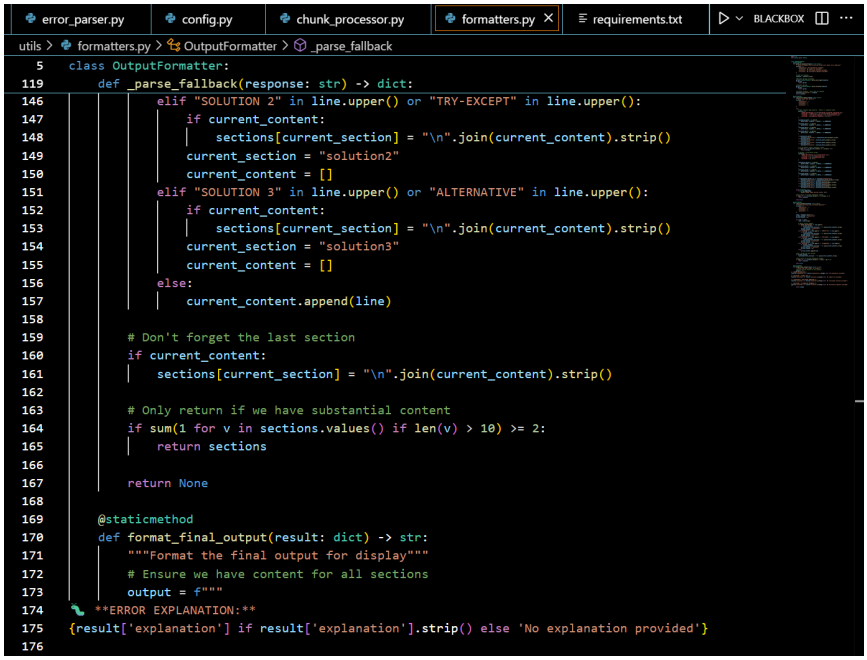



error_parser.py
config.py
chunk_processor.py
formatters.py
requirements.txt
BLACKBOX
utils > formatters.py > OutputFormatter > _parse_fallback
class OutputFormatter:
@staticmethod
def _parse_with_regex(response: str) -> dict:
"""Parse using regex patterns"""
sections = {
"explanation": "",
"solution1": "",
"solution2": "",
"solution3": "",
}
try:
# More flexible regex patterns - Pattern 1: Standard format
patterns = [
r"ERROR EXPLANATION:\s*(.*?)(?=SOLUTION 1|SOLUTION 2|SOLUTION 3|$)",
r"SOLUTION 1 \(SIMPLE FIX\):\s*(.*?)(?=SOLUTION 2|SOLUTION 3|$)",
r"SOLUTION 2 \(TRY-EXCEPT HANDLING\):\s*(.*?)(?=SOLUTION 3|$)",
r"SOLUTION 3 \(ALTERNATIVE APPROACH\):\s*(.*?)$",
]
explanation_match = re.search(
patterns[0], response, re.DOTALL | re.IGNORECASE
)
solution1_match = re.search(
patterns[1], response, re.DOTALL | re.IGNORECASE
)
solution2_match = re.search(
patterns[2], response, re.DOTALL | re.IGNORECASE
)
solution3_match = re.search(
patterns[3], response, re.DOTALL | re.IGNORECASE
)


error_parser.py
config.py
chunk_processor.py
formatters.py
requirements.txt
BLACKBOX
utils > formatters.py > OutputFormatter > _parse_fallback
class OutputFormatter:
def _parse_with_regex(response: str) -> dict:
if explanation_match:
sections["explanation"] = explanation_match.group(1).strip()
if solution1_match:
sections["solution1"] = solution1_match.group(1).strip()
if solution2_match:
sections["solution2"] = solution2_match.group(1).strip()
if solution3_match:
sections["solution3"] = solution3_match.group(1).strip()
# If we found at least 3 sections, return
if sum(1 for v in sections.values() if v.strip()) >= 3:
return sections
# Pattern 2: Alternative format
patterns2 = [
r"ERROR EXPLANATION:\s*(.*?)(?=SOLUTION 1|$)",
r"SOLUTION 1:\s*(.*?)(?=SOLUTION 2|$)",
r"SOLUTION 2:\s*(.*?)(?=SOLUTION 3|$)",
r"SOLUTION 3:\s*(.*?)$",
]
explanation_match2 = re.search(
patterns2[0], response, re.DOTALL | re.IGNORECASE
)
solution1_match2 = re.search(
patterns2[1], response, re.DOTALL | re.IGNORECASE
)
solution2_match2 = re.search(
patterns2[2], response, re.DOTALL | re.IGNORECASE
)


error_parser.py
config.py
chunk_processor.py
formatters.py
requirements.txt
BLACKBOX
utils > formatters.py > OutputFormatter > _parse_fallback
class OutputFormatter:
def _parse_with_regex(response: str) -> dict:
solution3_match2 = re.search(
patterns2[3], response, re.DOTALL | re.IGNORECASE
)
if explanation_match2 and not sections["explanation"]:
sections["explanation"] = explanation_match2.group(1).strip()
if solution1_match2 and not sections["solution1"]:
sections["solution1"] = solution1_match2.group(1).strip()
if solution2_match2 and not sections["solution2"]:
sections["solution2"] = solution2_match2.group(1).strip()
if solution3_match2 and not sections["solution3"]:
sections["solution3"] = solution3_match2.group(1).strip()
except Exception as e:
if Config.DEBUG_MODE:
print(f"Debug: Regex parsing failed: {e}")
# Only return if we have substantial content
if sum(1 for v in sections.values() if v.strip()) >= 2:
return sections
return None


error_parser.py
config.py
chunk_processor.py
formatters.py
requirements.txt
BLACKBOX
utils > formatters.py > OutputFormatter > _parse_fallback
class OutputFormatter:
@staticmethod
def _parse_fallback(response: str) -> dict:
"""Fallback parsing using line-based approach"""
sections = {
"explanation": "",
"solution1": "",
"solution2": "",
"solution3": "",
}
lines = response.split("\n")
current_section = "explanation"
current_content = []
for line in lines:
s\vith\Downloads\AI Bug Fixer\utils__pycache__
rip()
# Detect section headers
if "ERROR EXPLANATION" in line.upper():
if current_content:
sections[current_section] = "\n".join(current_content).strip()
current_section = "explanation"
current_content = []
elif "SOLUTION 1" in line.upper() or "SIMPLE FIX" in line.upper():
if current_content:
sections[current_section] = "\n".join(current_content).strip()
current_section = "solution1"
current_content = []
elif "SOLUTION 2" in line.upper() or "TRY-EXCEPT" in line.upper():
if current_content:
sections[current_section] = "\n".join(current_content).strip()
current_section = "solution2"



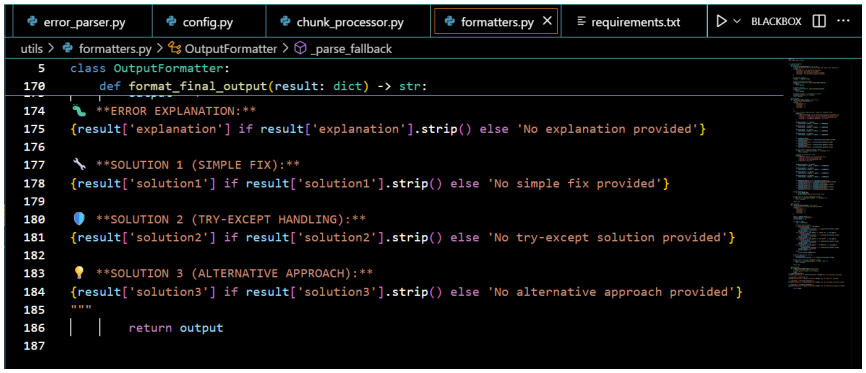


Figure 11: Develop Robust Output Formatting Utilities

Milestone 4: Frontend Delivery and Application Launch

Milestone: Building the Gradio User Interface and Orchestrating the Full Pipeline

This final milestone focuses on creating the interactive web application interface using Gradio and integrating all previously developed backend components (Configuration, Error Parser, Groq Handler, and Output Formatter) into a seamless, executable pipeline.

Activity 1: Build the Gradio User Interface (app.py)

This activity defines the visual structure and user experience of the AI Bug Fix Advisor.

Explanation

- **UI Framework**: The interface is constructed using **gr.Blocks** which provides granular control over the layout, allowing for the integration of custom styling and a clear two-column layout.
- **Input Design**: The input section features two large **gr.Textbox** components housed within **gr.Tabs**—one for the **Python Code** and one for the **Error Traceback**. This design ensures a clean workspace and prevents users from mixing the two distinct inputs.
- **Theming and Styling**: Custom CSS is applied to enhance the professional appearance, ensuring readability with code-friendly fonts and visually distinguishing the various components (e.g., using a distinct theme and color palette).
- **Output Component**: A single **gr.Markdown** component is used for the output, which is crucial as it correctly renders the highly structured, emoji-rich Markdown text generated by the OutputFormatter.

Activity 2: Orchestrate Application Flow and Launch

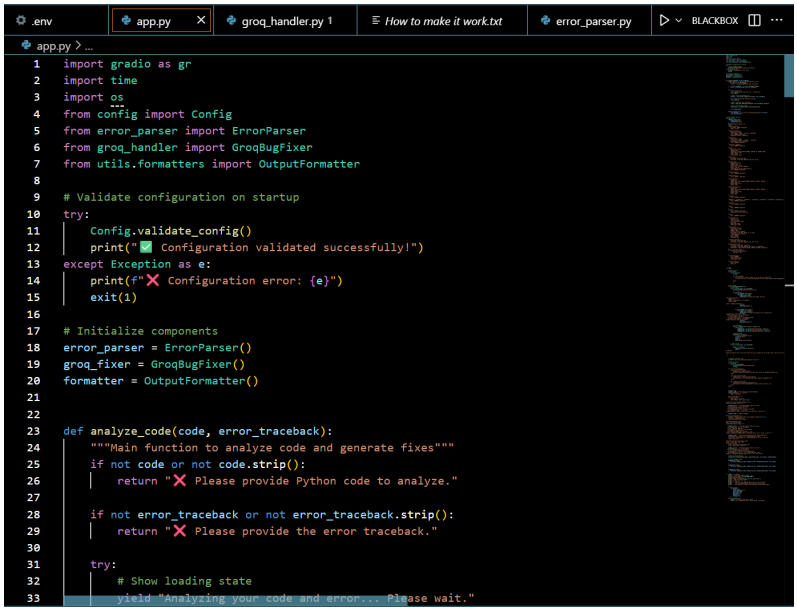
This final activity binds the UI elements to the backend logic, defining the end-to-end user journey and launching the application.

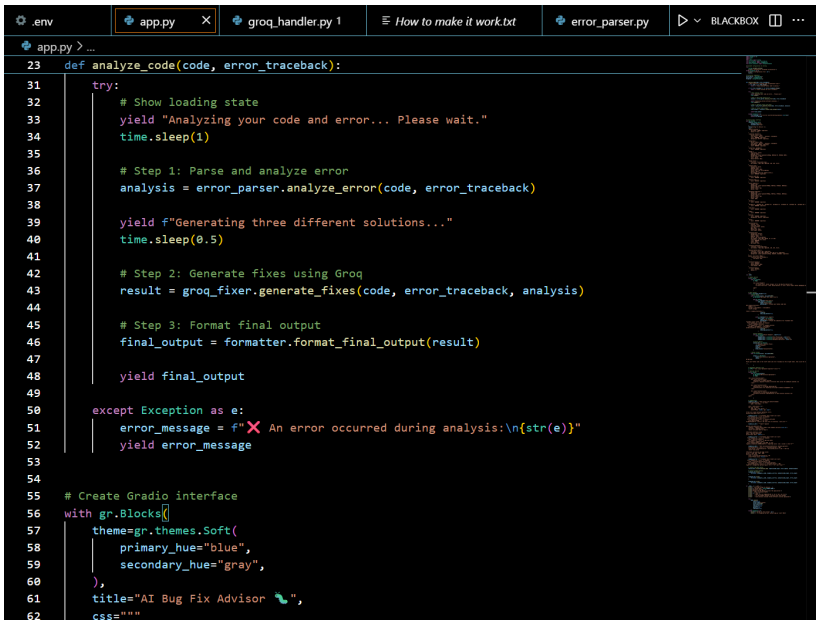
Explanation

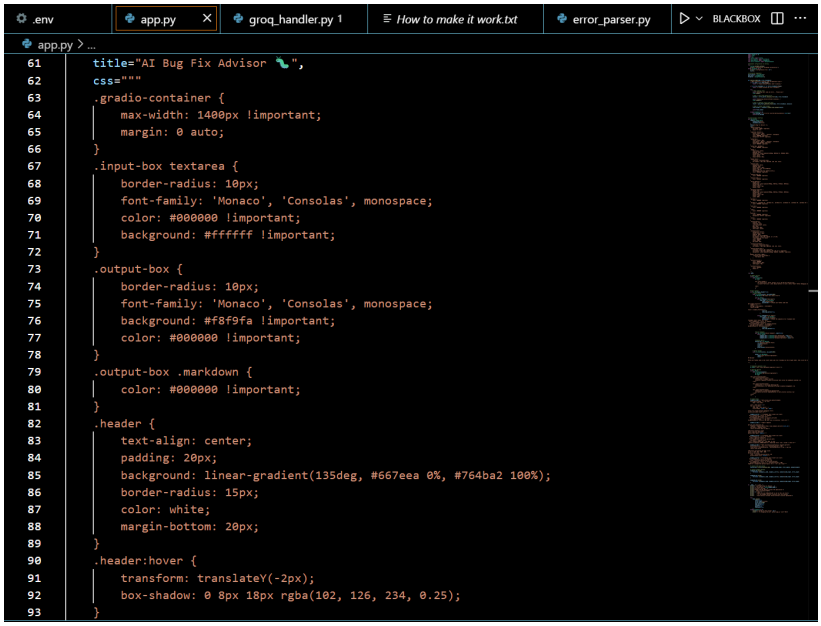
- **Orchestration Function**: The main **analyze_code** function is implemented to define the sequential execution of the entire pipeline. It takes the code and traceback as input, calls the ErrorParser, passes the resulting context to the GroqBugFixer, and finally feeds the raw LLM response to the OutputFormatter. The use of a **generator function** allows for **streaming status updates** (e.g., "Analyzing your code...", "Contacting Groq...") to the UI, improving perceived speed.

- **Binding Interactivity**: The **analyze_btn** is bound to the analyze_code function, ensuring that a user click triggers the full process and the final, formatted output updates the output_markdown component.
- **Quick Examples**: Logic is implemented to bind "Quick Example" buttons (not shown in the snippet) to functions that pre-populate the input textboxes, enabling new users and testers to instantly demonstrate the system's capabilities with pre-loaded scenarios.
- **Application Launch**: The final block of code uses the standard Python entry point to launch the Gradio interface, making the AI Bug Fix Advisor accessible via a local server URL.

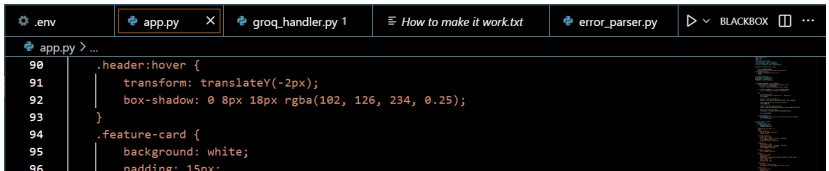
Code snippets of app.py:



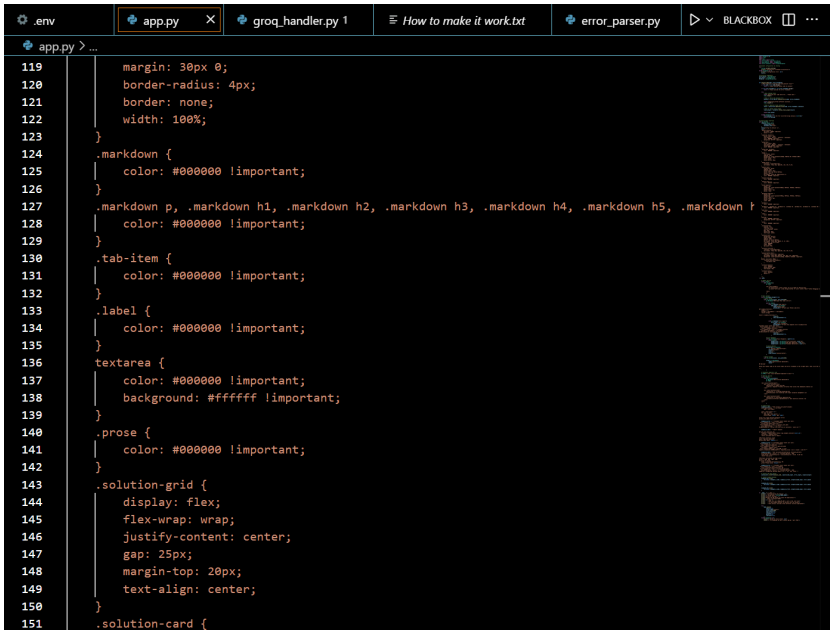




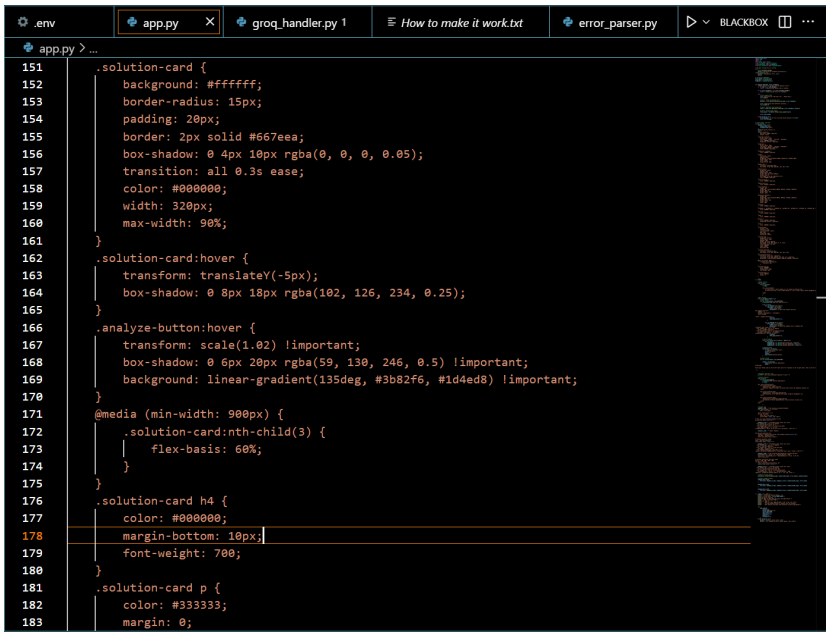

```
61      title="AI Bug Fix Advisor 🔧",
62      css="""
63      .gradio-container {
64          max-width: 1400px !important;
65          margin: 0 auto;
66      }
67      .input-box textarea {
68          border-radius: 10px;
69          font-family: 'Monaco', 'Consolas', monospace;
70          color: #000000 !important;
71          background: #ffffff !important;
72      }
73      .output-box {
74          border-radius: 10px;
75          font-family: 'Monaco', 'Consolas', monospace;
76          background: #f8f9fa !important;
77          color: #000000 !important;
78      }
79      .output-box .markdown {
80          color: #000000 !important;
81      }
82      .header {
83          text-align: center;
84          padding: 20px;
85          background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
86          border-radius: 15px;
87          color: white;
88          margin-bottom: 20px;
89      }
90      .header:hover {
91          transform: translateY(-2px);
92          box-shadow: 0 8px 18px rgba(102, 126, 234, 0.25);
93      }
```



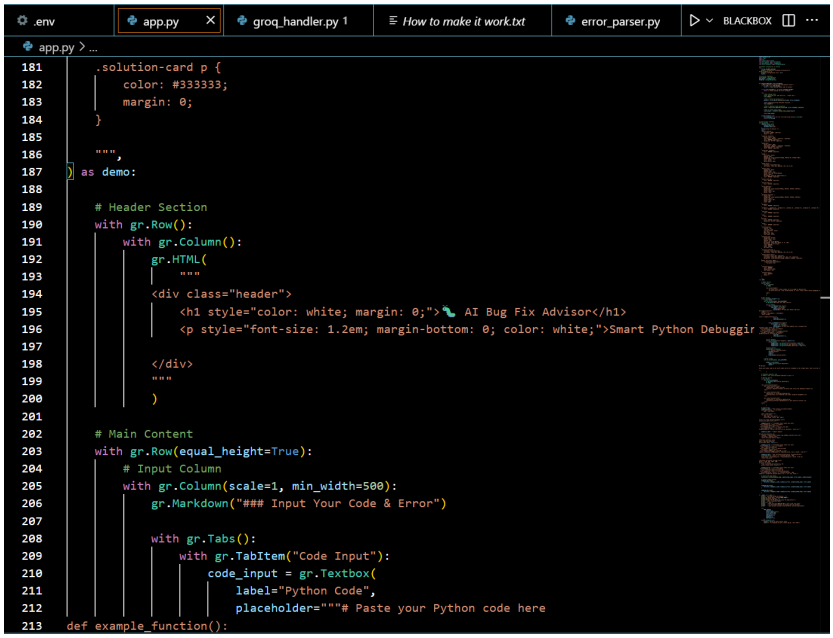

```
90      .header:hover {
91          transform: translateY(-2px);
92          box-shadow: 0 8px 18px rgba(102, 126, 234, 0.25);
93      }
94      .feature-card {
95          background: white;
96          padding: 15px;
97          border-radius: 10px;
98          border-left: 4px solid #667eea;
99          margin: 10px 0;
100         box-shadow: 0 2px 4px rgba(0,0,0,0.1);
101         color: #000000 !important;
102     }
103     .feature-card h4 {
104         color: #000000 !important;
105     }
106     .feature-card p {
107         color: #333333 !important;
108     }
109     .thick-separator {
110         height: 6px;
111         background: linear-gradient(90deg, #667eea, #764ba2, #667eea);
112         margin: 40px 0;
113         border-radius: 3px;
114         border: none;
115     }
116     .permanent-separator {
117         height: 8px;
118         background: linear-gradient(90deg, #667eea, #764ba2, #667eea);
119         margin: 30px 0;
120         border-radius: 4px;
121         border: none;
122         width: 100%;
```




```
119         margin: 30px 0;
120         border-radius: 4px;
121         border: none;
122         width: 100%;
123     }
124     .markdown {
125         color: #000000 !important;
126     }
127     .markdown p, .markdown h1, .markdown h2, .markdown h3, .markdown h4, .markdown h5, .markdown h
128         color: #000000 !important;
129     }
130     .tab-item {
131         color: #000000 !important;
132     }
133     .label {
134         color: #000000 !important;
135     }
136     textarea {
137         color: #000000 !important;
138         background: #ffffff !important;
139     }
140     .prose {
141         color: #000000 !important;
142     }
143     .solution-grid {
144         display: flex;
145         flex-wrap: wrap;
146         justify-content: center;
147         gap: 25px;
148         margin-top: 20px;
149         text-align: center;
150     }
151     .solution-card {
```

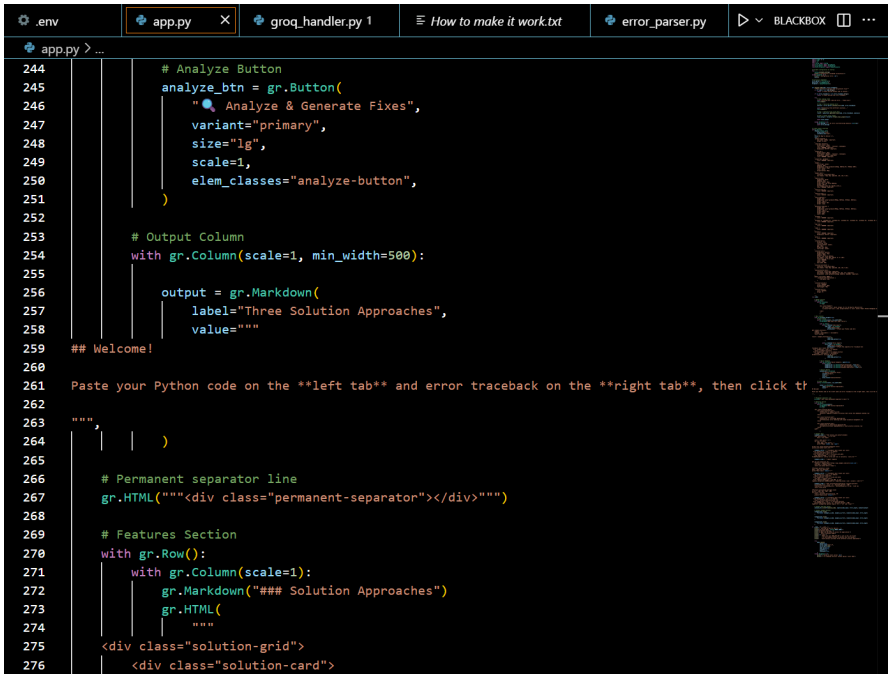



```
151     .solution-card {
152         background: #ffffff;
153         border-radius: 15px;
154         padding: 20px;
155         border: 2px solid #667eea;
156         box-shadow: 0 4px 10px rgba(0, 0, 0, 0.05);
157         transition: all 0.3s ease;
158         color: #000000;
159         width: 320px;
160         max-width: 90%;
161     }
162     .solution-card:hover {
163         transform: translateY(-5px);
164         box-shadow: 0 8px 18px rgba(102, 126, 234, 0.25);
165     }
166     .analyze-button:hover {
167         transform: scale(1.02) !important;
168         box-shadow: 0 6px 20px rgba(59, 130, 246, 0.5) !important;
169         background: linear-gradient(135deg, #3b82f6, #1d4ed8) !important;
170     }
171     @media (min-width: 900px) {
172         .solution-card:nth-child(3) {
173             flex-basis: 60%;
174         }
175     }
176     .solution-card h4 {
177         color: #000000;
178         margin-bottom: 10px;
179         font-weight: 700;
180     }
181     .solution-card p {
182         color: #333333;
183         margin: 0;
```

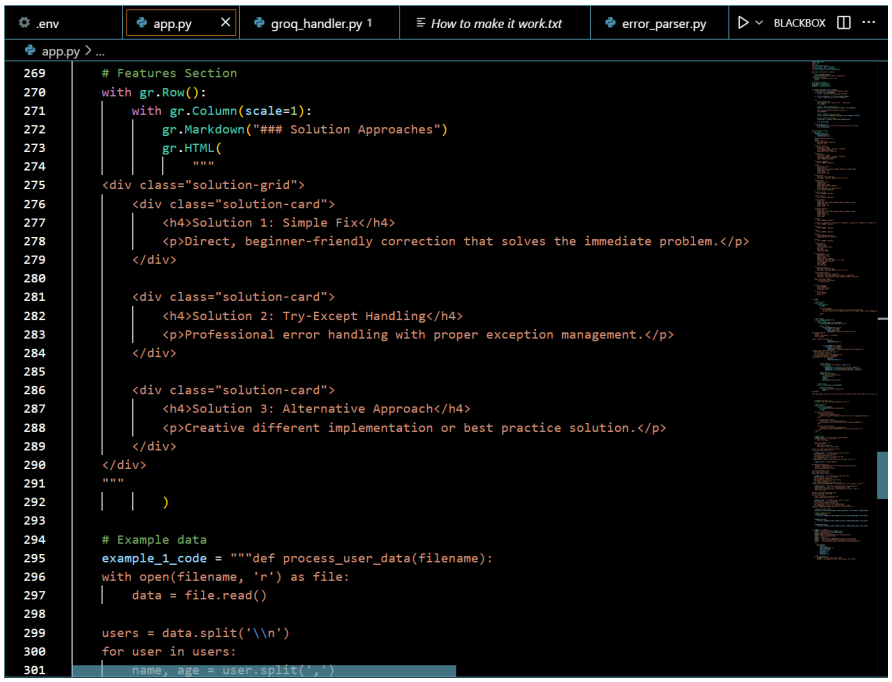


```
.env | app.py | groq_handler.py 1 | How to make it work.txt | error_parser.py | BLACKBOX
app.py > ...
181     .solution-card p {
182         color: #333333;
183         margin: 0;
184     }
185
186     """,
187 ) as demo:
188
189     # Header Section
190     with gr.Row():
191         with gr.Column():
192             gr.HTML(
193                 """
194             <div class="header">
195                 <h1 style="color: white; margin: 0;">🐞 AI Bug Fix Advisor</h1>
196                 <p style="font-size: 1.2em; margin-bottom: 0; color: white;">Smart Python Debuggir
197
198             </div>
199             """
200             )
201
202     # Main Content
203     with gr.Row(equal_height=True):
204         # Input Column
205         with gr.Column(scale=1, min_width=500):
206             gr.Markdown("### Input Your Code & Error")
207
208             with gr.Tabs():
209                 with gr.TabItem("Code Input"):
210                     code_input = gr.Textbox(
211                         label="Python Code",
212                         placeholder="""# Paste your Python code here
213 def example_function():
```

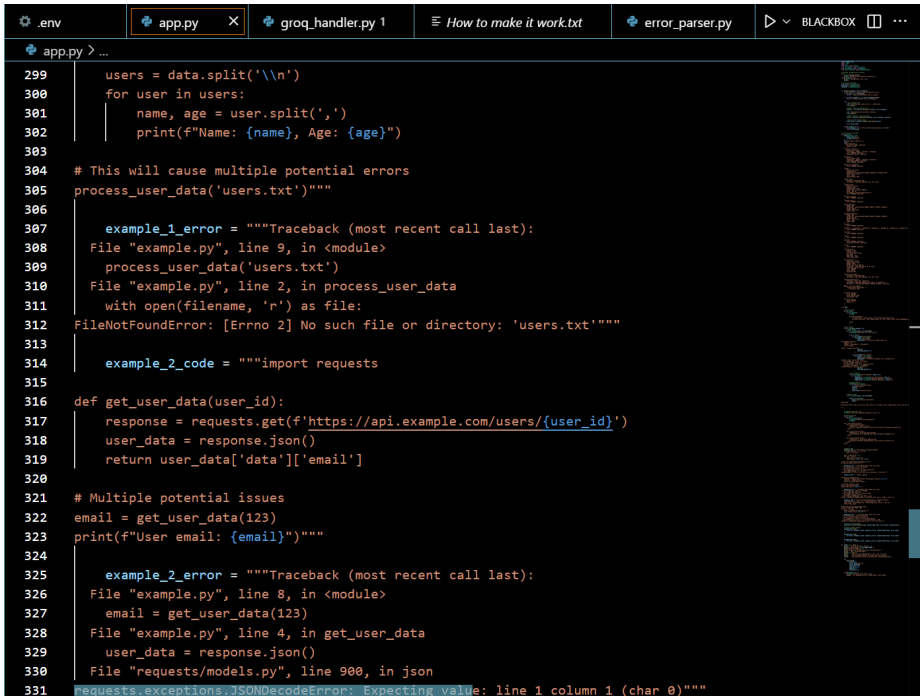
```
.env | app.py | groq_handler.py 1 | How to make it work.txt | error_parser.py | BLACKBOX
app.py > ...
213 def example_function():
214     numbers = []
215     average = sum(numbers) / len(numbers)
216     return average
217
218 result = example_function()""",
219                         lines=15,
220                         show_copy_button=True,
221                     )
222
223                 with gr.TabItem("Error Input"):
224                     error_input = gr.Textbox(
225                         label="Error Traceback",
226                         placeholder="""# Paste the complete error traceback here
227 Traceback (most recent call last):
228   File "example.py", line 6, in <module>
229     result = example_function()
230   File "example.py", line 3, in example_function
231     average = sum(numbers) / len(numbers)
232 ZeroDivisionError: division by zero""",
233                         lines=10,
234                         show_copy_button=True,
235                     )
236
237             # Quick Examples
238             with gr.Accordion("Quick Examples", open=False):
239                 with gr.Row():
240                     example1_btn = gr.Button("File Processing", size="sm")
241                     example2_btn = gr.Button("API Data Handling", size="sm")
242                     example3_btn = gr.Button("Database Operations", size="sm")
243
244             # Analyze Button
245             analyze_btn = gr.Button(
```



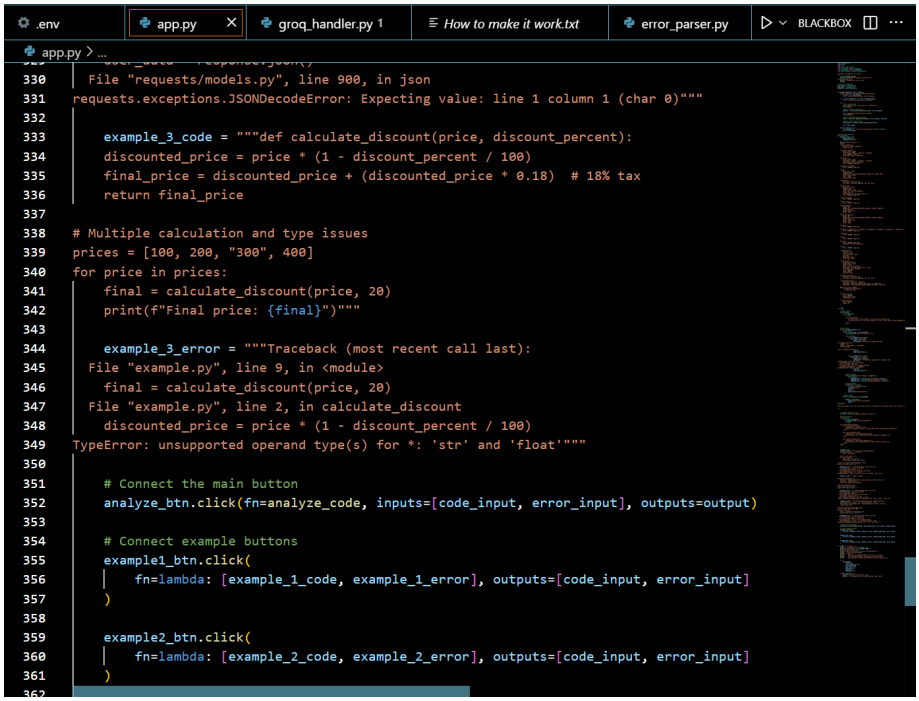
```
.env | app.py | groq_handler.py 1 | How to make it work.txt | error_parser.py | BLACKBOX
app.py > ...
244             # Analyze Button
245             analyze_btn = gr.Button(
246                 "🔍 Analyze & Generate Fixes",
247                 variant="primary",
248                 size="lg",
249                 scale=1,
250                 elem_classes="analyze-button",
251             )
252
253         # Output Column
254         with gr.Column(scale=1, min_width=500):
255
256             output = gr.Markdown(
257                 label="Three Solution Approaches",
258                 value="""
259 ## Welcome!
260
261 Paste your Python code on the **left tab** and error traceback on the **right tab**, then click th
262
263 """,
264             )
265
266     # Permanent separator line
267     gr.HTML("""<div class="permanent-separator"></div>""")
268
269     # Features Section
270     with gr.Row():
271         with gr.Column(scale=1):
272             gr.Markdown("### Solution Approaches")
273             gr.HTML(
274                 """
275     <div class="solution-grid">
276         <div class="solution-card">
```



```
.env | app.py | groq_handler.py 1 | How to make it work.txt | error_parser.py | BLACKBOX
app.py > ...
269     # Features Section
270     with gr.Row():
271         with gr.Column(scale=1):
272             gr.Markdown("### Solution Approaches")
273             gr.HTML(
274                 """
275     <div class="solution-grid">
276         <div class="solution-card">
277             <h4>Solution 1: Simple Fix</h4>
278             <p>Direct, beginner-friendly correction that solves the immediate problem.</p>
279         </div>
280
281         <div class="solution-card">
282             <h4>Solution 2: Try-Except Handling</h4>
283             <p>Professional error handling with proper exception management.</p>
284         </div>
285
286         <div class="solution-card">
287             <h4>Solution 3: Alternative Approach</h4>
288             <p>Creative different implementation or best practice solution.</p>
289         </div>
290     </div>
291     """
292             )
293
294     # Example data
295     example_1_code = """def process_user_data(filename):
296     with open(filename, 'r') as file:
297         data = file.read()
298
299     users = data.split('\\n')
300     for user in users:
301         name, age = user.split(',')
```



```
    users = data.split('\\n')
    for user in users:
        name, age = user.split(',')
        print(f"Name: {name}, Age: {age}")

# This will cause multiple potential errors
process_user_data('users.txt')"""

    example_1_error = """Traceback (most recent call last):
  File "example.py", line 9, in <module>
    process_user_data('users.txt')
  File "example.py", line 2, in process_user_data
    with open(filename, 'r') as file:
FileNotFoundError: [Errno 2] No such file or directory: 'users.txt'"""

    example_2_code = """import requests

def get_user_data(user_id):
    response = requests.get(f'https://api.example.com/users/{user_id}')
    user_data = response.json()
    return user_data['data']['email']

# Multiple potential issues
email = get_user_data(123)
print(f"User email: {email}")"""

    example_2_error = """Traceback (most recent call last):
  File "example.py", line 8, in <module>
    email = get_user_data(123)
  File "example.py", line 4, in get_user_data
    user_data = response.json()
  File "requests/models.py", line 900, in json
requests.exceptions.JSONDecodeError: Expecting value: line 1 column 1 (char 0)"""
```



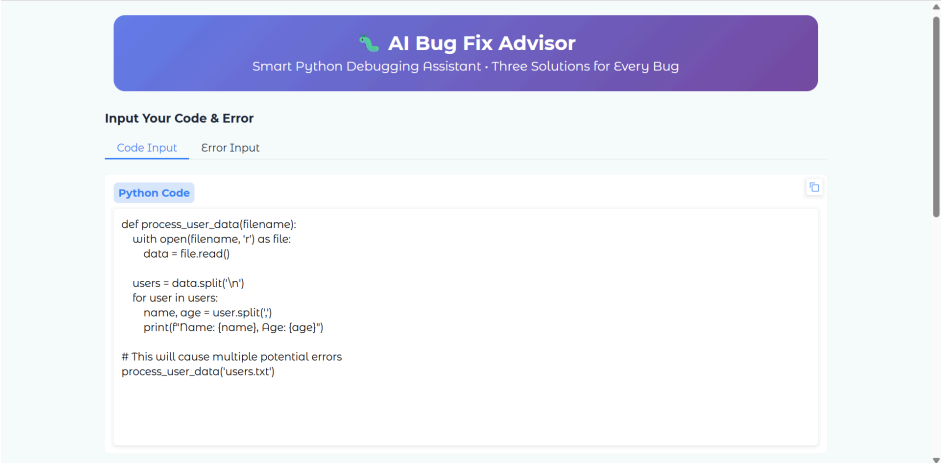
```
  File "requests/models.py", line 900, in json
requests.exceptions.JSONDecodeError: Expecting value: line 1 column 1 (char 0)"""

    example_3_code = """def calculate_discount(price, discount_percent):
    discounted_price = price * (1 - discount_percent / 100)
    final_price = discounted_price + (discounted_price * 0.18)  # 18% tax
    return final_price

# Multiple calculation and type issues
prices = [100, 200, "300", 400]
for price in prices:
    final = calculate_discount(price, 20)
    print(f"Final price: {final}")"""

    example_3_error = """Traceback (most recent call last):
  File "example.py", line 9, in <module>
    final = calculate_discount(price, 20)
  File "example.py", line 2, in calculate_discount
    discounted_price = price * (1 - discount_percent / 100)
TypeError: unsupported operand type(s) for *: 'str' and 'float'"""

    # Connect the main button
    analyze_btn.click(fn=analyze_code, inputs=[code_input, error_input], outputs=output)

    # Connect example buttons
    example1_btn.click(
        fn=lambda: [example_1_code, example_1_error], outputs=[code_input, error_input]
    )

    example2_btn.click(
        fn=lambda: [example_2_code, example_2_error], outputs=[code_input, error_input]
    )
```

Figure 12: app.py

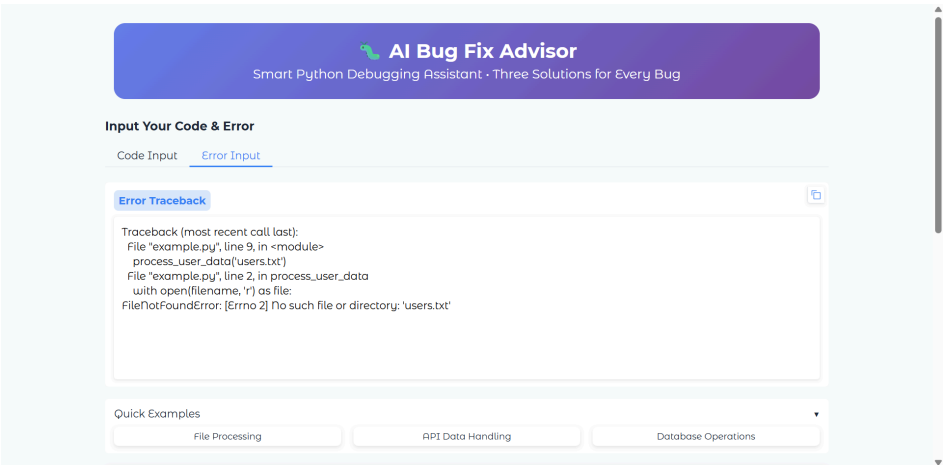
Output Preview

Example 1: File Processing

Code:



Error:



Output:

🦠 **ERROR EXPLANATION:** The error occurs because the file 'users.txt' does not exist in the current working directory. The `open()` function attempts to open the file in read mode, but since it cannot find the file, it raises a `FileNotFoundError`. This error is not handled by the code, causing the program to terminate abruptly. The code assumes that the file exists and is correctly formatted, which is not the case.

🔧 **SOLUTION 1 (SIMPLE FIX):** To fix this issue, we need to ensure that the file 'users.txt' exists before trying to open it. We can create the file manually or modify the code to create the file if it does not exist. Here is a simple fix:

```
def process_user_data(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        with open(filename, 'w') as file:
            pass  # Create an empty file if it does not exist
        print("File created. Please populate it with user data.")
        return

    users = data.split('\n')
    for user in users:
        if user:  # Check if the line is not empty
            name, age = user.split(',')
            print(f"Name: {name}, Age: {age}")

process_user_data('users.txt')
```

🛡️ **SOLUTION 2 (TRY-EXCEPT HANDLING):** We can use try-except blocks to handle potential errors when working with files. This approach allows us to provide informative error messages and handle different types of exceptions. Here's how you can modify the code to include try-except handling:

🛡️ **SOLUTION 2 (TRY-EXCEPT HANDLING):** We can use try-except blocks to handle potential errors when working with files. This approach allows us to provide informative error messages and handle different types of exceptions. Here's how you can modify the code to include try-except handling:

```
def process_user_data(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        print(f"Error: The file '{filename}' does not exist.")
        return
    except PermissionError:
        print(f"Error: You do not have permission to read the file '{filename}'.")
        return
    except Exception as e:
        print(f"An error occurred: {e}")
        return

    try:
        users = data.split('\n')
        for user in users:
            if user:  # Check if the line is not empty
                name, age = user.split(',')
                print(f"Name: {name}, Age: {age}")
    except ValueError:
        print("Error: The file is not formatted correctly.")
    except Exception as e:
        print(f"An error occurred: {e}")

process_user_data('users.txt')
```

💡 **SOLUTION 3 (ALTERNATIVE APPROACH):** Instead of reading the entire file into memory, we can use a more efficient approach by reading the file line by line. We can also use a `csv` module to handle the comma-separated values, which provides more flexibility and error handling. Here's an alternative approach:

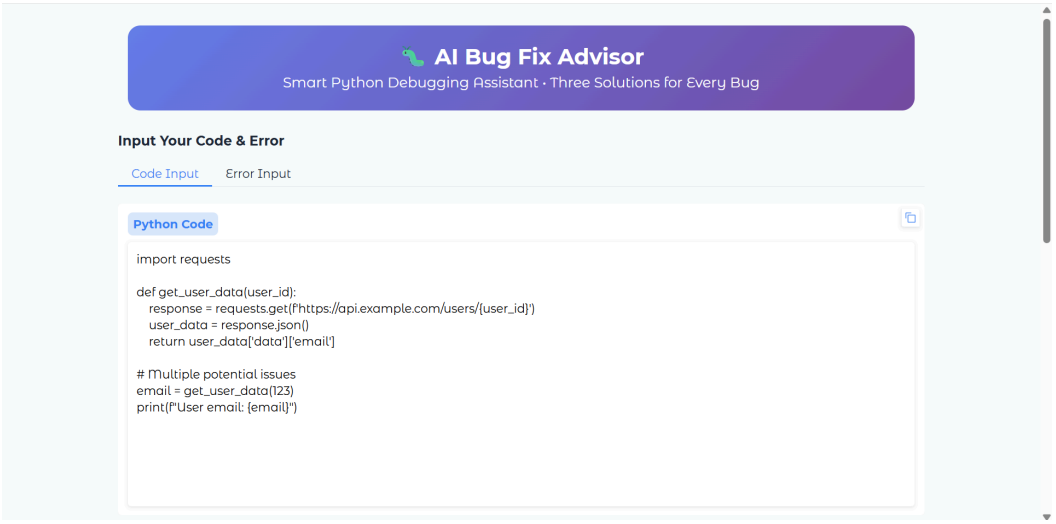
```
import csv

def process_user_data(filename):
    try:
        with open(filename, 'r', newline='') as file:
            reader = csv.reader(file)
            for row in reader:
                if len(row) == 2:  # Check if the row has exactly two values
                    name, age = row
                    print(f"Name: {name}, Age: {age}")
                else:
                    print(f"Error: Invalid data format in the file '{filename}'.")
    except FileNotFoundError:
        print(f"Error: The file '{filename}' does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")

process_user_data('users.txt')
```

This approach is more efficient and flexible, as it can handle large files and provides better error handling for invalid data formats.

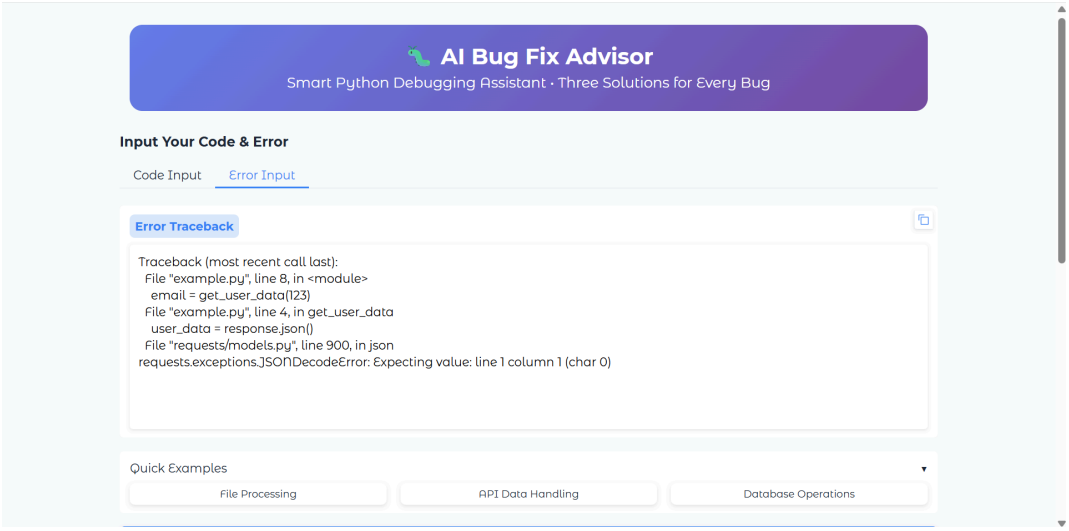
Solution Approaches

Example 2: API Data Handling

Code:



Error:



Output:

🦎 **ERROR EXPLANATION:** The error occurs because the `requests.get()` call returns a response that is not in JSON format, causing the `response.json()` method to throw a `JSONDecodeError`. This could be due to various reasons such as the API endpoint returning an error message, the server being down, or the API not returning data in the expected format. As a result, when the code attempts to parse the response as JSON, it fails and raises an exception.

🔧 **SOLUTION 1 (SIMPLE FIX):** To fix this issue, we can add a simple check to ensure that the response is successful and contains JSON data before attempting to parse it. Here's a modified version of the code:

```
import requests

def get_user_data(user_id):
    response = requests.get(f'https://api.example.com/users/{user_id}')
    if response.status_code == 200 and 'application/json' in response.headers['Content-Type']:
        user_data = response.json()
        return user_data.get('data', {}).get('email')
    else:
        return None

email = get_user_data(123)
print(f"User email: {email}")
```

🛡️ **SOLUTION 2 (TRY-EXCEPT HANDLING):** We can use try-except blocks to catch and handle the `JSONDecodeError` exception. Additionally, we can also catch other potential exceptions that may occur during the request, such as `requests.exceptions.RequestException`. Here's an example:

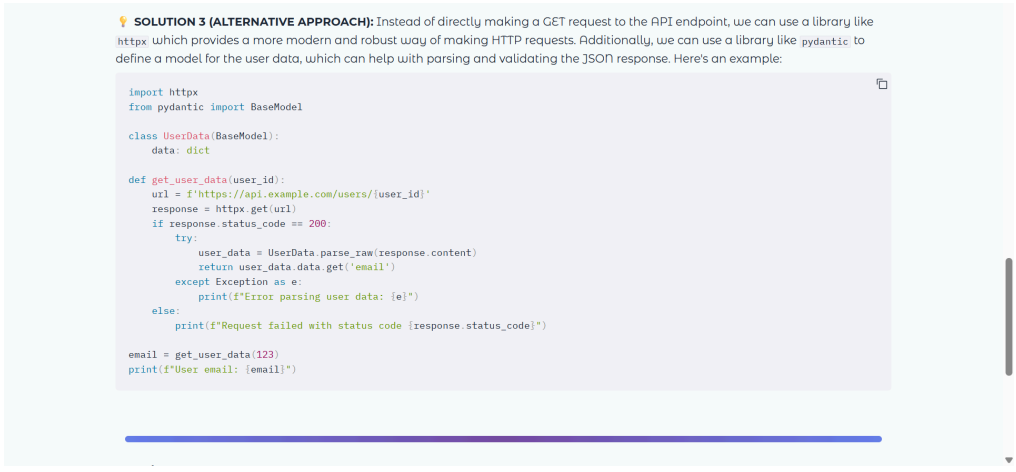
```
import requests

def get_user_data(user_id):
    try:
        response = requests.get(f'https://api.example.com/users/{user_id}')
```

🛡️ **SOLUTION 2 (TRY-EXCEPT HANDLING):** We can use try-except blocks to catch and handle the `JSONDecodeError` exception. Additionally, we can also catch other potential exceptions that may occur during the request, such as `requests.exceptions.RequestException`. Here's an example:

```
import requests

def get_user_data(user_id):
    try:
        response = requests.get(f'https://api.example.com/users/{user_id}')
        response.raise_for_status()  # Raise an exception for 4xx or 5xx status codes
        user_data = response.json()
        return user_data.get('data', {}).get('email')
    except requests.exceptions.RequestException as e:
        print(f"Request error: {e}")
    except ValueError as e:
        print(f"JSON parsing error: {e}")
    except Exception as e:
        print(f"Unexpected error: {e}")

email = get_user_data(123)
print(f"User email: {email}")
```

💡 **SOLUTION 3 (ALTERNATIVE APPROACH):** Instead of directly making a GET request to the API endpoint, we can use a library like `httpx` which provides a more modern and robust way of making HTTP requests. Additionally, we can use a library like `pydantic` to define a model for the user data, which can help with parsing and validating the JSON response. Here's an example:

```
import httpx
from pydantic import BaseModel

class UserData(BaseModel):
    data: dict
```

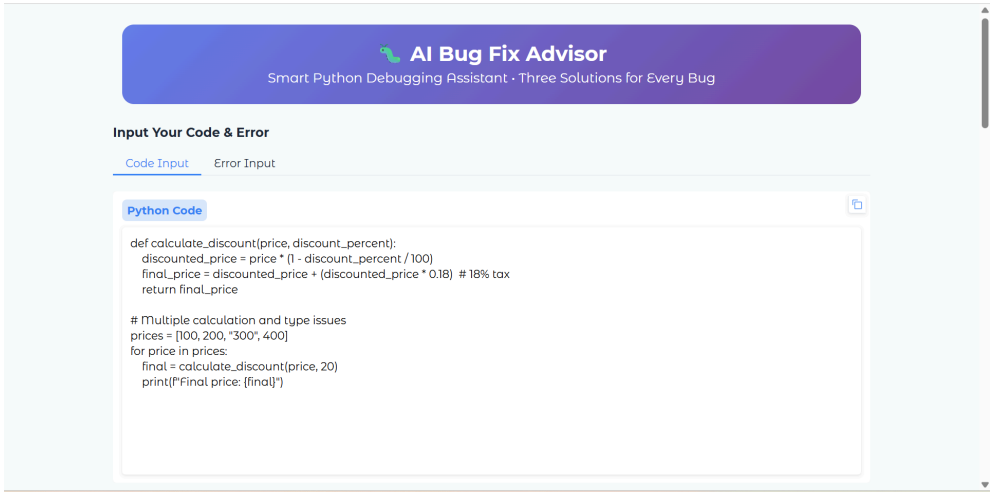


💡 **SOLUTION 3 (ALTERNATIVE APPROACH):** Instead of directly making a GET request to the API endpoint, we can use a library like `httpx` which provides a more modern and robust way of making HTTP requests. Additionally, we can use a library like `pydantic` to define a model for the user data, which can help with parsing and validating the JSON response. Here's an example:

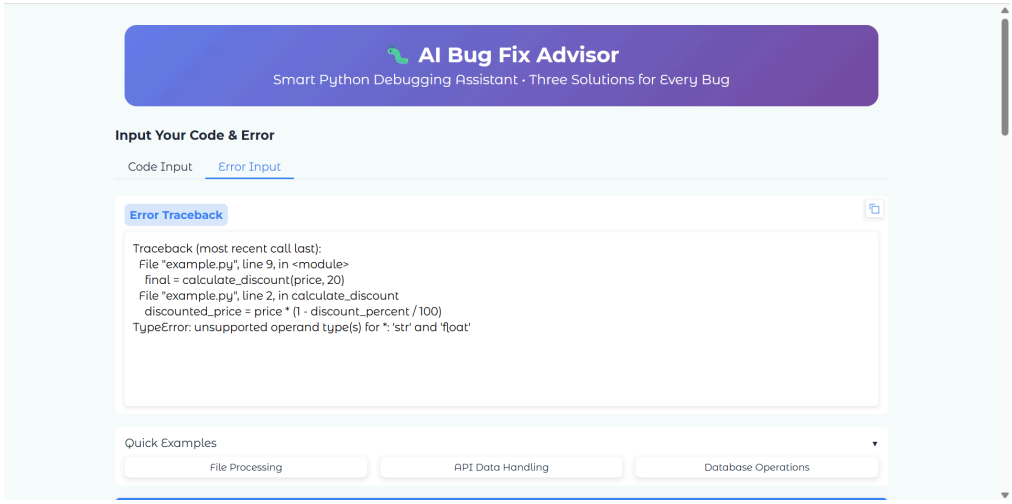
```
import httpx
from pydantic import BaseModel

class UserData(BaseModel):
    data: dict

def get_user_data(user_id):
    url = f'https://api.example.com/users/{user_id}'
    response = httpx.get(url)
    if response.status_code == 200:
        try:
            user_data = UserData.parse_raw(response.content)
            return user_data.data.get('email')
        except Exception as e:
            print(f"Error parsing user data: {e}")
    else:
        print(f"Request failed with status code {response.status_code}")

email = get_user_data(123)
print(f"User email: {email}")
```

Example 3: Database Operations

Code:



Error:



Output:

🦠 **ERROR EXPLANATION:** The error occurs because the `calculate_discount` function is being called with a string value ("300") for the `price` parameter. This string value is then attempted to be multiplied by a float value, which is not a supported operation in Python. The `calculate_discount` function expects both `price` and `discount_percent` to be numeric values, but the provided `prices` list contains a string value. This mismatch in data types causes a TypeError to be raised.

🔧 **SOLUTION 1 (SIMPLE FIX):** To fix this issue, we need to ensure that all prices are numeric values. We can do this by converting the string value to a float before passing it to the `calculate_discount` function. Here's the corrected code:

```
def calculate_discount(price, discount_percent):
    discounted_price = price * (1 - discount_percent / 100)
    final_price = discounted_price + (discounted_price * 0.18)  # 18% tax
    return final_price

prices = [100, 200, float("300"), 400]
for price in prices:
    final = calculate_discount(price, 20)
    print(f"Final price: {final}")
```

🛡️ **SOLUTION 2 (TRY-EXCEPT HANDLING):** We can add try-except blocks to handle the TypeError and provide a meaningful error message. We'll also add a check to ensure that the price is a positive number. Here's the modified code:

```
def calculate_discount(price, discount_percent):
    try:
        price = float(price)
        if price <= 0:
            raise ValueError("Price must be a positive number")
        discounted_price = price * (1 - discount_percent / 100)
        final_price = discounted_price + (discounted_price * 0.18)  # 18% tax
        return final_price
    except ValueError as e:
```

🛡️ **SOLUTION 2 (TRY-EXCEPT HANDLING):** We can add try-except blocks to handle the TypeError and provide a meaningful error message. We'll also add a check to ensure that the price is a positive number. Here's the modified code:

```
def calculate_discount(price, discount_percent):
    try:
        price = float(price)
        if price <= 0:
            raise ValueError("Price must be a positive number")
        discounted_price = price * (1 - discount_percent / 100)
        final_price = discounted_price + (discounted_price * 0.18)  # 18% tax
        return final_price
    except ValueError as e:
        print(f"Error: {e}")
        return None
    except TypeError:
        print("Error: Price must be a numeric value")
        return None

prices = [100, 200, "300", 400]
for price in prices:
    final = calculate_discount(price, 20)
    if final is not None:
        print(f"Final price: {final}")
```

💡 **SOLUTION 3 (ALTERNATIVE APPROACH):** Instead of using a simple function to calculate the discount, we can create a `Product` class that encapsulates the price and discount percentage. We'll also use the `decimal` module to ensure accurate decimal arithmetic. Here's the alternative code:

```
from decimal import Decimal

class Product:
```

💡 **SOLUTION 3 (ALTERNATIVE APPROACH):** Instead of using a simple function to calculate the discount, we can create a `Product` class that encapsulates the price and discount percentage. We'll also use the `decimal` module to ensure accurate decimal arithmetic. Here's the alternative code:

```
from decimal import Decimal

class Product:
    def __init__(self, price, discount_percent):
        self.price = Decimal(str(price))
        self.discount_percent = Decimal(str(discount_percent))

    def calculate_discount(self):
        discounted_price = self.price * (1 - self.discount_percent / 100)
        final_price = discounted_price + (discounted_price * Decimal('0.18'))  # 18% tax
        return final_price

prices = [100, 200, 300, 400]
for price in prices:
    product = Product(price, 20)
    final = product.calculate_discount()
    print(f"Final price: {final}")
```

This approach provides a more object-oriented solution and ensures accurate decimal arithmetic.

Conclusion

The **AI Bug Fix Advisor** successfully integrates state-of-the-art Generative AI with practical developer workflow tools to create a robust and highly efficient debugging utility. By strategically combining high-speed inference from the **Groq LPU platform** with the intuitive frontend delivery of **Gradio**, the project achieves its core objective: providing instant, multifaceted solutions to Python code errors.

This project is a testament to the powerful synergy between specialized AI hardware and intelligent software design, demonstrating that LLMs can be utilized not just for generation, but for specialized, reliable, and structured analysis.

Key Achievements

- **Speed and Efficiency:** By using **Groq** for the core inference, the application overcomes the latency challenges typically associated with detailed LLM analysis, delivering comprehensive solutions in near real-time.
- **Structured Intelligence:** The strict prompt engineering logic developed in **groq_handler.py** successfully compels the LLM to provide **three distinct, non-overlapping solutions** (Simple Fix, Try-Except Handling, and Alternative Approach), offering the developer a spectrum of choices tailored to production quality, speed, and architectural best practices.
- **Robustness and Reliability:** The meticulous **Input Processing** (parsing traceback via regex, generating contextual code snippets) and the **Multi-Strategy Output Parsing** in **utils/formatters.py** ensure that the system handles imperfect inputs and outputs gracefully, guaranteeing consistent, structured results for the user.
- **Accessibility:** The deployment via **Gradio** provides a professional, user-friendly interface that makes this advanced debugging power accessible to developers of all skill levels, turning the frustrating process of bug fixing into an efficient, educational, and streamlined task.

In summary, the AI Bug Fix Advisor represents a significant step forward in AI-assisted development, offering a practical, deployable solution that dramatically accelerates the time spent on debugging and enhances code quality.