

# 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.

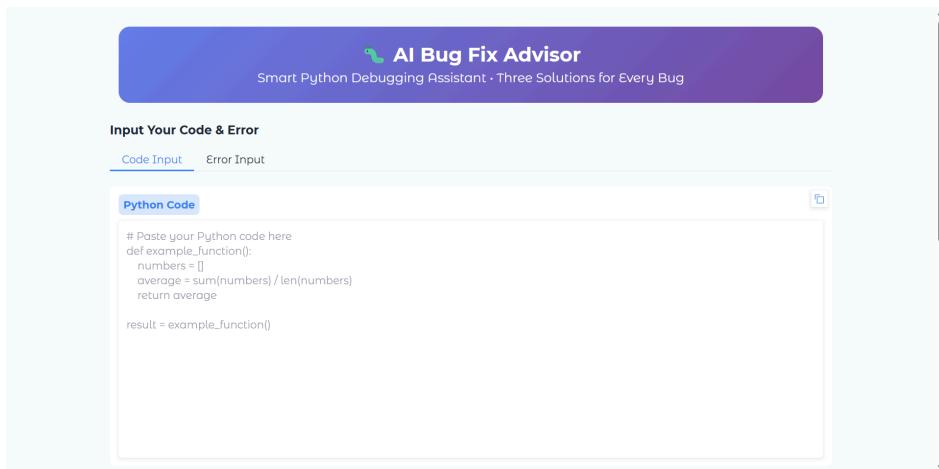


Figure 1: The opening page of the AI Bug Fix Advisor

## 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.

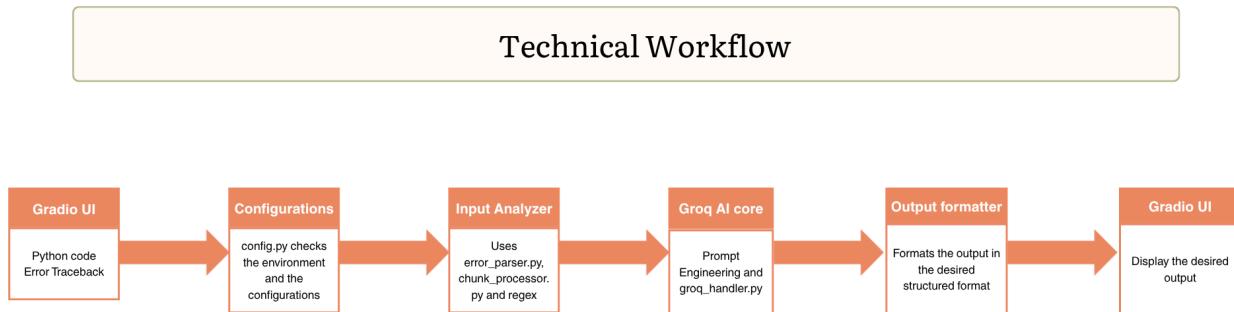


Figure 2: Technical Workflow of the project

## 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.

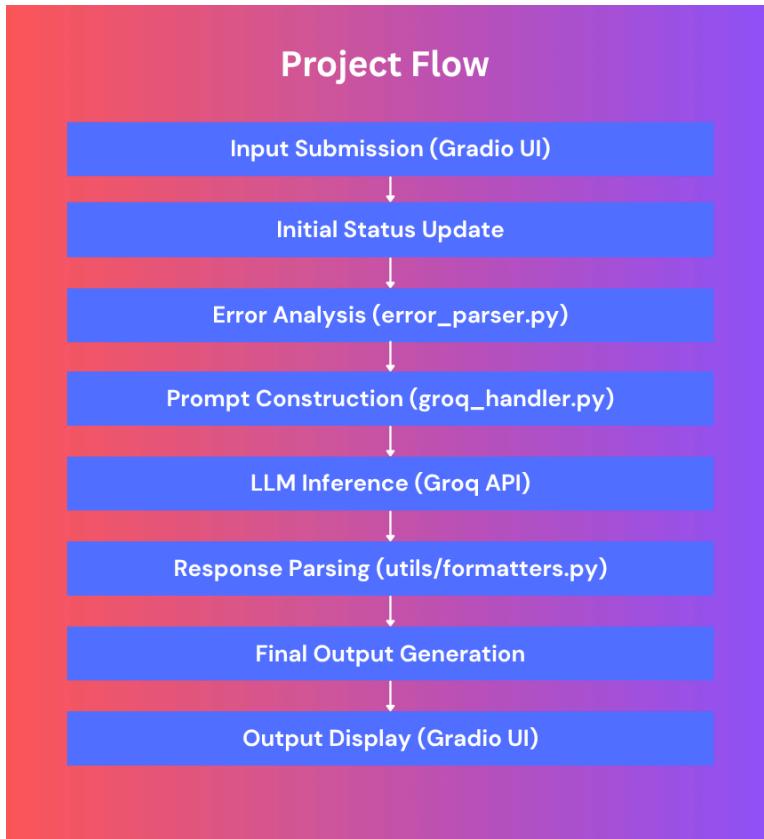


Figure 3: Project flow for AI Bug Fix Advisor

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

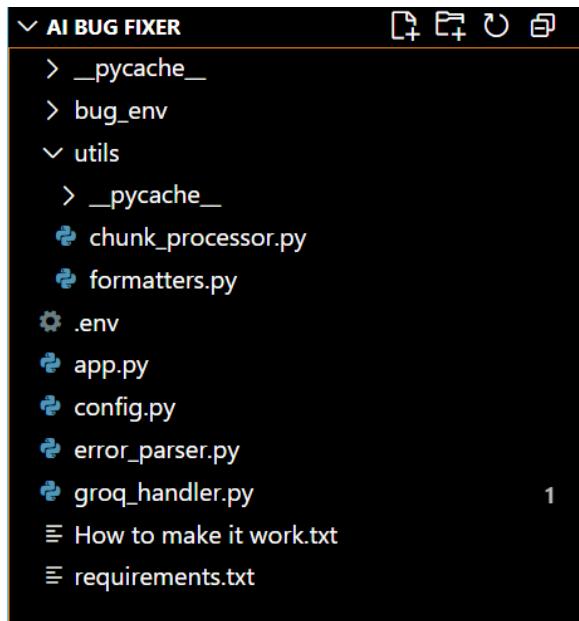


Figure 4: Folder structure of AI Bug Fixer application in VSCode interface

# 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

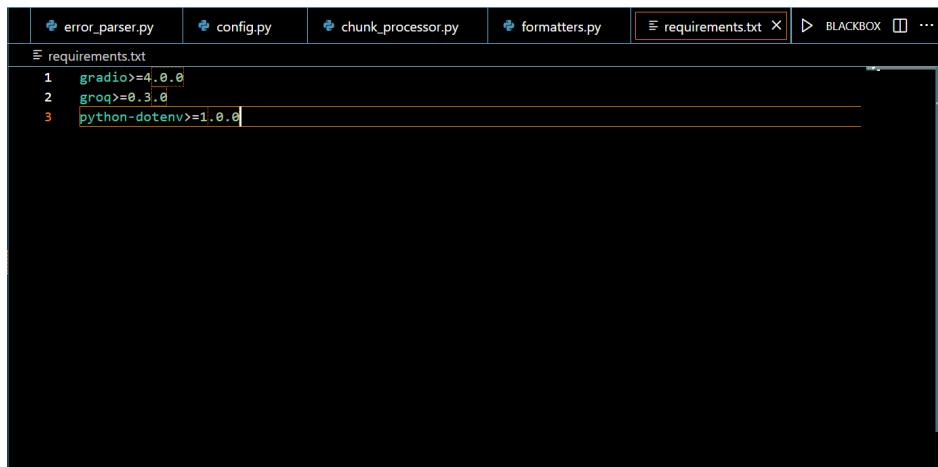


Figure 5: requirements.txt in VSCode interface

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:

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

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.

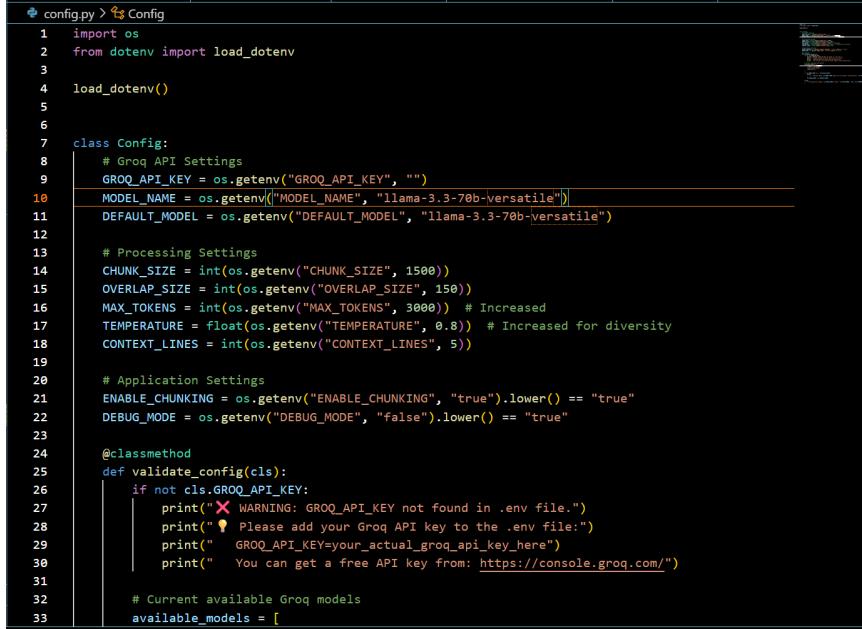
## Activity 1: Loading Environment Variables and Validating Configuration (config.py)

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

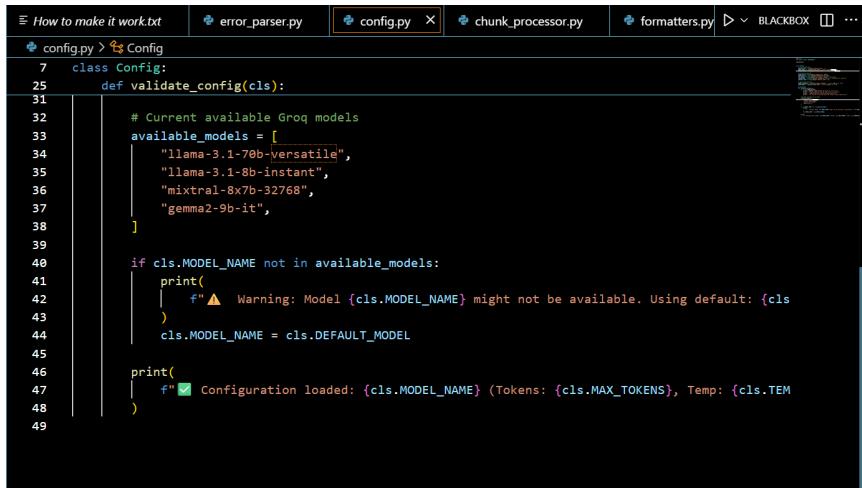
### Explanation

- **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:



```
config.py > Config
1 import os
2 from dotenv import load_dotenv
3
4 load_dotenv()
5
6
7 class Config:
8     # Groq API Settings
9     GROQ_API_KEY = os.getenv("GROQ_API_KEY", "")
10    MODEL_NAME = os.getenv("MODEL_NAME", "llama-3.3-70b-versatile")
11    DEFAULT_MODEL = os.getenv("DEFAULT_MODEL", "llama-3.3-70b-versatile")
12
13    # Processing Settings
14    CHUNK_SIZE = int(os.getenv("CHUNK_SIZE", 1500))
15    OVERLAP_SIZE = int(os.getenv("OVERLAP_SIZE", 150))
16    MAX_TOKENS = int(os.getenv("MAX_TOKENS", 3000)) # Increased
17    TEMPERATURE = float(os.getenv("TEMPERATURE", 0.8)) # Increased for diversity
18    CONTEXT_LINES = int(os.getenv("CONTEXT_LINES", 5))
19
20    # Application Settings
21    ENABLE_CHUNKING = os.getenv("ENABLE_CHUNKING", "true").lower() == "true"
22    DEBUG_MODE = os.getenv("DEBUG_MODE", "false").lower() == "true"
23
24    @classmethod
25    def validate_config(cls):
26        if not cls.GROQ_API_KEY:
27            print("⚠ WARNING: GROQ_API_KEY not found in .env file.")
28            print("Please add your Groq API key to the .env file:")
29            print("  GROQ_API_KEY=your_actual_groq_api_key_here")
30            print("  You can get a free API key from: https://console.groq.com/")
31
32        # Current available Groq models
33        available_models = [
34            "llama-3.1-70b-versatile",
35            "llama-3.1-8b-instant",
36            "mixtral-8x7b-32768",
37            "gemma2-9b-it",
38        ]
39
40        if cls.MODEL_NAME not in available_models:
41            print(
42                f"⚠ Warning: Model {cls.MODEL_NAME} might not be available. Using default: {cls.DEFAULT_MODEL}"
43            )
44            cls.MODEL_NAME = cls.DEFAULT_MODEL
45
46        print(
47            f"✓ Configuration loaded: {cls.MODEL_NAME} (Tokens: {cls.MAX_TOKENS}, Temp: {cls.TEMPERATURE})"
48        )
49
```



```
How to make it work.txt error_parser.py config.py chunk_processor.py formatters.py BLACKBOX ...
config.py > Config
7 class Config:
25    def validate_config(cls):
31        # Current available Groq models
32        available_models = [
33            "llama-3.1-70b-versatile",
34            "llama-3.1-8b-instant",
35            "mixtral-8x7b-32768",
36            "gemma2-9b-it",
37        ]
38
39        if cls.MODEL_NAME not in available_models:
40            print(
41                f"⚠ Warning: Model {cls.MODEL_NAME} might not be available. Using default: {cls.DEFAULT_MODEL}"
42            )
43            cls.MODEL_NAME = cls.DEFAULT_MODEL
44
45        print(
46            f"✓ Configuration loaded: {cls.MODEL_NAME} (Tokens: {cls.MAX_TOKENS}, Temp: {cls.TEMPERATURE})"
47        )
48
```

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.

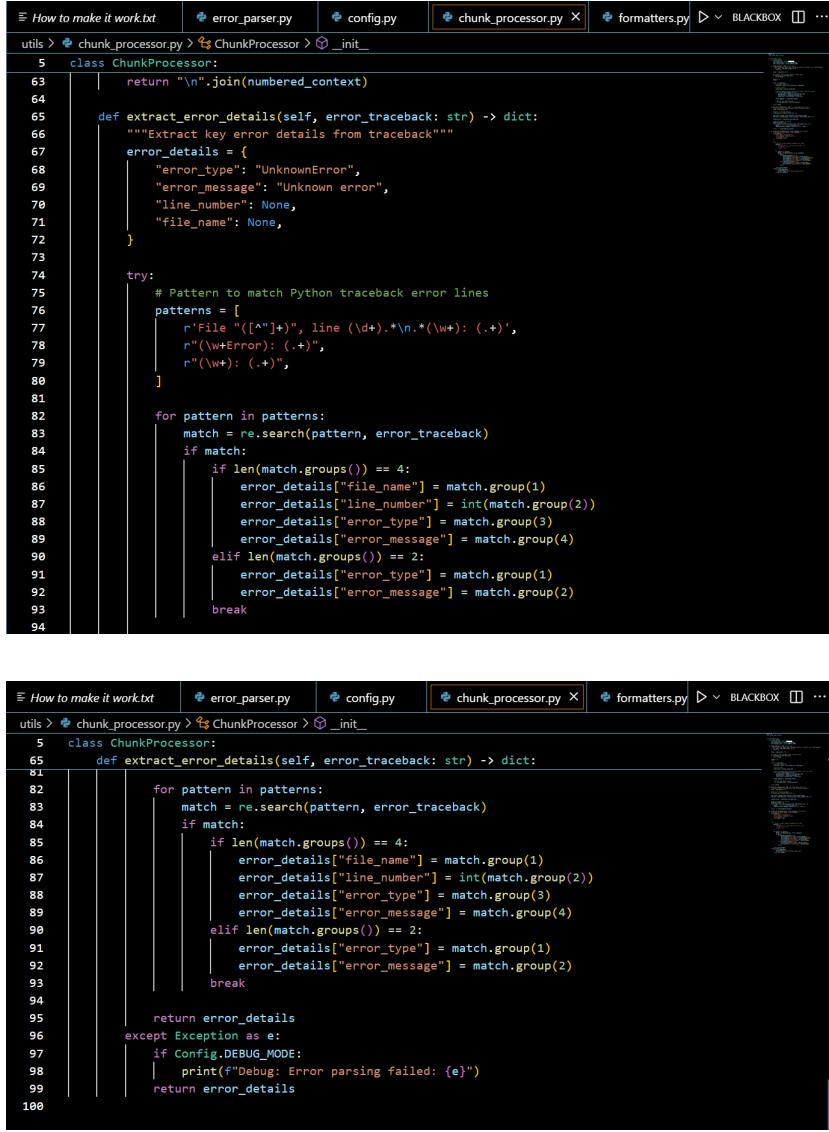
#### Explanation

- **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):

```
 1 import re
 2 from config import Config
 3
 4
 5 class ChunkProcessor:
 6     def __init__(self):
 7         self.chunk_size = Config.CHUNK_SIZE
 8         self.overlap_size = Config.OVERLAP_SIZE
 9         self.context_lines = Config.CONTEXT_LINES
10
11     def chunk_code(self, code: str) -> list:
12         """Split code into chunks with overlapping context for better error understanding"""
13         if not code or len(code.strip()) == 0:
14             return []
15
16         lines = code.split("\n")
17
18         # If code is small enough, return as single chunk
19         if len(lines) <= self.chunk_size:
20             return [code]
21
22         chunks = []
23         i = 0
24
25         while i < len(lines):
26             # Calculate chunk end
27             chunk_end = min(i + self.chunk_size, len(lines))
28
29             # Extract chunk
30             chunk_lines = lines[i:chunk_end]
31
32             # Add overlapping context from previous chunk if not first chunk
33             if i > 0 and self.overlap_size > 0:
```

```
 5 class ChunkProcessor:
 6     def chunk_code(self, code: str) -> list:
 7
 8         # Add overlapping context from previous chunk if not first chunk
 9         if i > 0 and self.overlap_size > 0:
10             overlap_start = max(0, i - self.overlap_size)
11             overlap_lines = lines[overlap_start:i]
12             chunk_lines = overlap_lines + chunk_lines
13
14             chunks.append("\n".join(chunk_lines))
15
16             # Move to next chunk position
17             i += self.chunk_size - self.overlap_size
18
19         return chunks
20
21     def get_error_context(self, code: str, error_line: int) -> str:
22         """Extract context around the error line for better analysis"""
23         lines = code.split("\n")
24
25         # Adjust for 0-based indexing
26         error_line_idx = max(0, error_line - 1)
27
28         start_line = max(0, error_line_idx - self.context_lines)
29         end_line = min(len(lines), error_line_idx + self.context_lines + 1)
30
31         context_lines = lines[start_line:end_line]
32
33         # Add line numbers for clarity
34         numbered_context = []
35         for i, line in enumerate(context_lines, start=start_line + 1):
36             marker = ">>> " if i == error_line else "    "
37             numbered_context.append(f"{marker}{line} {i}: {line}")
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
69
```



```

How to make it work.txt error_parser.py config.py chunk_processor.py formatters.py BLACKBOX ...
utils > chunk_processor.py > ChunkProcessor > __init__ ...

5 class ChunkProcessor:
63     return "\n".join(numbered_context)
64
65     def extract_error_details(self, error_traceback: str) -> dict:
66         """Extract key error details from traceback"""
67         error_details = {
68             "error_type": "UnknownError",
69             "error_message": "Unknown error",
70             "line_number": None,
71             "file_name": None,
72         }
73
74         try:
75             # Pattern to match Python traceback error lines
76             patterns = [
77                 r'File "([^"]+)", line (\d+).*\n.*(\w+): (.+)',
78                 r"(\w+Error): (.+)",
79                 r"(\w+): (.+)"
80             ]
81
82             for pattern in patterns:
83                 match = re.search(pattern, error_traceback)
84                 if match:
85                     if len(match.groups()) == 4:
86                         error_details["file_name"] = match.group(1)
87                         error_details["line_number"] = int(match.group(2))
88                         error_details["error_type"] = match.group(3)
89                         error_details["error_message"] = match.group(4)
90                     elif len(match.groups()) == 2:
91                         error_details["error_type"] = match.group(1)
92                         error_details["error_message"] = match.group(2)
93                     break
94
95             return error_details
96         except Exception as e:
97             if Config.DEBUG_MODE:
98                 print(f"Debug: Error parsing failed: {e}")
99             return error_details
100

```

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.

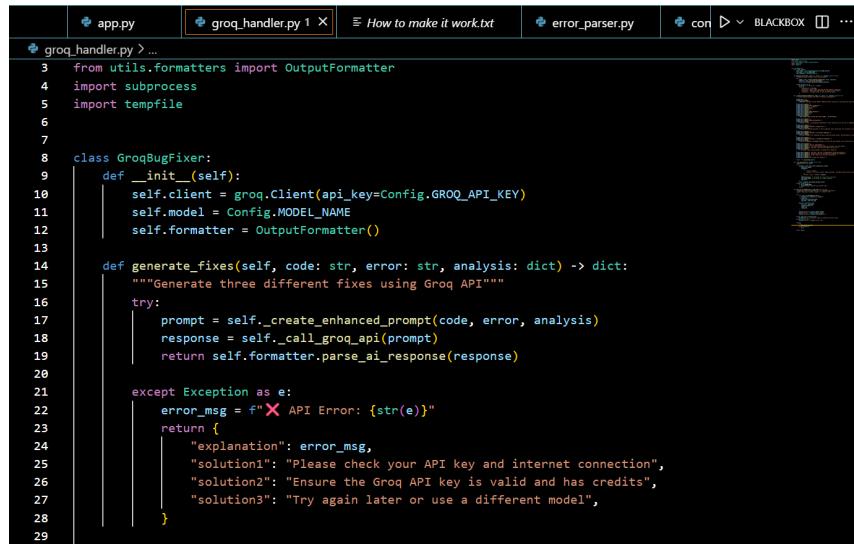
### Explanation

- **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:



The screenshot shows a terminal window with several tabs open. The active tab is `groq_handler.py`, which contains the following Python code:

```
 3  from utils.formatters import OutputFormatter
 4  import subprocess
 5  import tempfile
 6
 7
 8  class GroqBugFixer:
 9      def __init__(self):
10          self.client = groq.Client(api_key=Config.GROQ_API_KEY)
11          self.model = Config.MODEL_NAME
12          self.formatter = OutputFormatter()
13
14      def generate_fixes(self, code: str, error: str, analysis: dict) -> dict:
15          """Generate three different fixes using Groq API"""
16          try:
17              prompt = self._create_enhanced_prompt(code, error, analysis)
18              response = self._call_groq_api(prompt)
19              return self.formatter.parse_ai_response(response)
20
21          except Exception as e:
22              error_msg = f"API Error: {str(e)}"
23              return {
24                  "explanation": error_msg,
25                  "solution1": "Please check your API key and internet connection",
26                  "solution2": "Ensure the Groq API key is valid and has credits",
27                  "solution3": "Try again later or use a different model",
28              }
```

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:

The image shows two screenshots of a terminal window with multiple tabs open. The active tab is 'groq\_handler.py 1'. The code in the first screenshot (lines 8-61) defines a class 'GroqBugFixer' with a method '\_create\_enhanced\_prompt'. This method constructs a multi-solution prompt template. It starts with a header, followed by a blank line, a code block, another blank line, an error message, another blank line, an explanation, another blank line, a solution 1 header, another blank line, a solution 2 header, another blank line, and a solution 3 header. The second screenshot (lines 8-83) continues the method definition, adding more solutions and critical requirements. Both screenshots show the code being typed into the terminal.

```
app.py groq_handler.py 1 > ... How to make it work.txt error_parser.py con ▶ BLACKBOX ...
```

```
8 class GroqBugFixer:
9     def _create_enhanced_prompt(self, code: str, error: str, analysis: dict) -> str:
10         """Create enhanced prompt with STRICT formatting requirements"""
11
12         prompt_parts = []
13         prompt_parts.append(
14             | "IMPORTANT: You MUST provide EXACTLY THREE different solutions in the specified format"
15         )
16         prompt_parts.append("")
17         prompt_parts.append("CODE TO ANALYZE:")
18         prompt_parts.append(f"```python")
19         prompt_parts.append(code)
20         prompt_parts.append(f"````")
21         prompt_parts.append("")
22         prompt_parts.append("ERROR MESSAGE:")
23         prompt_parts.append(error)
24         prompt_parts.append("")
25         prompt_parts.append(
26             | "YOUR RESPONSE MUST FOLLOW THIS EXACT FORMAT - NO DEVIATIONS:"
27         )
28         prompt_parts.append("")
29         prompt_parts.append("ERROR EXPLANATION:")
30         prompt_parts.append(
31             | "[Provide a clear, one-paragraph explanation of what caused the error and why it happened]"
32         )
33         prompt_parts.append("")
34         prompt_parts.append("SOLUTION 1 (SIMPLE FIX):")
35         prompt_parts.append(
36             | "[Provide the SIMPLEST possible fix that a beginner would understand. This should be a single line of code]"
37         )
38         prompt_parts.append("")
39         prompt_parts.append("SOLUTION 2 (TRY-EXCEPT HANDLING):")
40         prompt_parts.append(
41             | "[Provide a ROBUST error handling solution using try-except blocks. Include specific exception handling]"
42         )
43         prompt_parts.append("")
44         prompt_parts.append("SOLUTION 3 (ALTERNATIVE APPROACH):")
45         prompt_parts.append(
46             | "[Provide a COMPLETELY DIFFERENT approach to solve the same problem. This could involve a completely different algorithm or data structure]"
47         )
48         prompt_parts.append("")
49         prompt_parts.append("CRITICAL REQUIREMENTS:")
50         prompt_parts.append("1. All three solutions MUST be different from each other")
51         prompt_parts.append("2. Solution 1 must be the simplest direct fix")
52         prompt_parts.append(
53             | "3. Solution 2 must include proper try-except error handling"
54         )
55         prompt_parts.append(
56             | "4. Solution 3 must be a fundamentally different approach"
57         )
58         prompt_parts.append("5. Each solution must include actual Python code examples")
59         prompt_parts.append("6. Do NOT skip any of the three solutions")
60         prompt_parts.append("")
61         prompt_parts.append("Now provide your analysis:")
62
63     return "\n".join(prompt_parts)
```

```
app.py groq_handler.py 1 > ... How to make it work.txt error_parser.py con ▶ BLACKBOX ...
```

```
8 class GroqBugFixer:
9     def _create_enhanced_prompt(self, code: str, error: str, analysis: dict) -> str:
10         """
11         prompt_parts.append("")
12         prompt_parts.append("SOLUTION 2 (TRY-EXCEPT HANDLING):")
13         prompt_parts.append(
14             | "[Provide a ROBUST error handling solution using try-except blocks. Include specific exception handling]"
15         )
16         prompt_parts.append("")
17         prompt_parts.append("SOLUTION 3 (ALTERNATIVE APPROACH):")
18         prompt_parts.append(
19             | "[Provide a COMPLETELY DIFFERENT approach to solve the same problem. This could involve a completely different algorithm or data structure]"
20         )
21         prompt_parts.append("")
22         prompt_parts.append("CRITICAL REQUIREMENTS:")
23         prompt_parts.append("1. All three solutions MUST be different from each other")
24         prompt_parts.append("2. Solution 1 must be the simplest direct fix")
25         prompt_parts.append(
26             | "3. Solution 2 must include proper try-except error handling"
27         )
28         prompt_parts.append(
29             | "4. Solution 3 must be a fundamentally different approach"
30         )
31         prompt_parts.append("5. Each solution must include actual Python code examples")
32         prompt_parts.append("6. Do NOT skip any of the three solutions")
33         prompt_parts.append("")
34         prompt_parts.append("Now provide your analysis:")
35
36     return "\n".join(prompt_parts)
```

Figure 10: Engineer the Multi-Solution Prompt Template

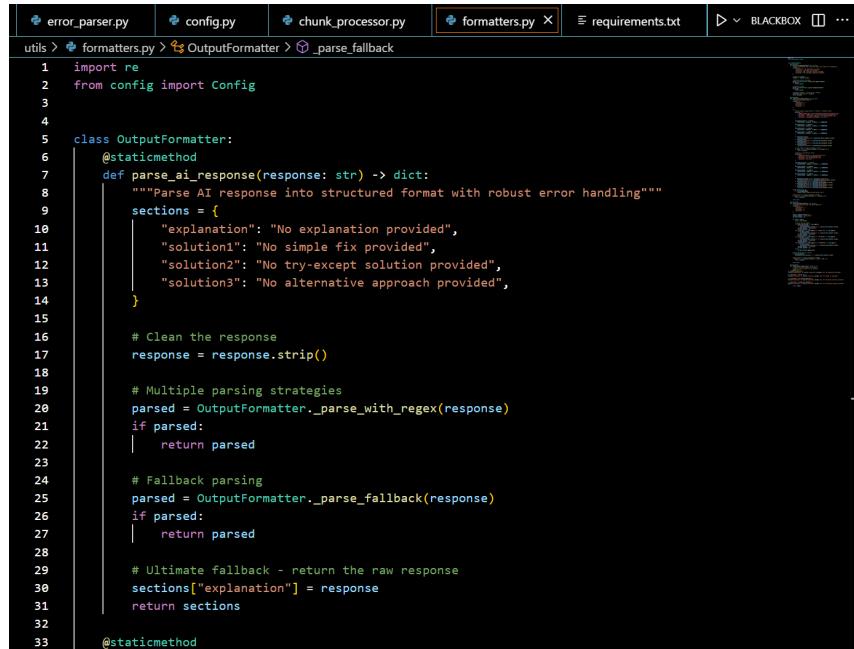
## Activity 2: Develop Robust Output Formatting Utilities (utils/formatters.py)

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):



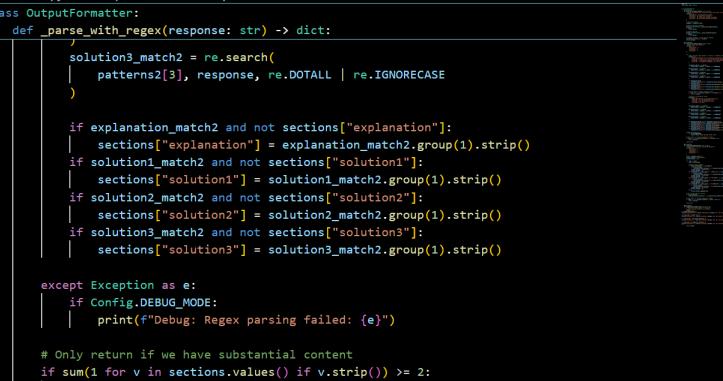
```
utils > formatters.py > OutputFormatter > _parse_fallback
1 import re
2 from config import Config
3
4
5 class OutputFormatter:
6     @staticmethod
7     def parse_ai_response(response: str) -> dict:
8         """Parse AI response into structured format with robust error handling"""
9         sections = {
10             "explanation": "No explanation provided",
11             "solution1": "No simple fix provided",
12             "solution2": "No try-except solution provided",
13             "solution3": "No alternative approach provided",
14         }
15
16         # Clean the response
17         response = response.strip()
18
19         # Multiple parsing strategies
20         parsed = OutputFormatter._parse_with_regex(response)
21         if parsed:
22             return parsed
23
24         # Fallback parsing
25         parsed = OutputFormatter._parse_fallback(response)
26         if parsed:
27             return parsed
28
29         # Ultimate fallback - return the raw response
30         sections["explanation"] = response
31         return sections
32
33     @staticmethod
```

utils > formatters.py > OutputFormatter > \_parse\_fallback

```
5 class OutputFormatter:
33     @staticmethod
34     def _parse_with_regex(response: str) -> dict:
35         """Parse using regex patterns"""
36         sections = {
37             "explanation": "",
38             "solution1": "",
39             "solution2": "",
40             "solution3": ""
41         }
42
43     try:
44         # More flexible regex patterns - Pattern 1: Standard format
45         patterns = [
46             r"ERROR EXPLANATION:\s*(.*?)(?=SOLUTION 1|SOLUTION 2|SOLUTION 3|$)",
47             r"SOLUTION 1 \(\SIMPLE FIX\):\s*(.*?)(?=SOLUTION 2|SOLUTION 3|$)",
48             r"SOLUTION 2 \(\TRY-EXCEPT HANDLING\):\s*(.*?)(?=SOLUTION 3|$)",
49             r"SOLUTION 3 \(\ALTERNATIVE APPROACH\):\s*(.*?)$",
50         ]
51
52         explanation_match = re.search(
53             | patterns[0], response, re.DOTALL | re.IGNORECASE
54         )
55         solution1_match = re.search(
56             | patterns[1], response, re.DOTALL | re.IGNORECASE
57         )
58         solution2_match = re.search(
59             | patterns[2], response, re.DOTALL | re.IGNORECASE
60         )
61         solution3_match = re.search(
62             | patterns[3], response, re.DOTALL | re.IGNORECASE
63         )
64     
```

utils > formatters.py > OutputFormatter > \_parse\_fallback

```
5 class OutputFormatter:
34     def _parse_with_regex(response: str) -> dict:
35         if explanation_match:
36             sections["explanation"] = explanation_match.group(1).strip()
37         if solution1_match:
38             sections["solution1"] = solution1_match.group(1).strip()
39         if solution2_match:
40             sections["solution2"] = solution2_match.group(1).strip()
41         if solution3_match:
42             sections["solution3"] = solution3_match.group(1).strip()
43
44         # If we found at least 3 sections, return
45         if sum(1 for v in sections.values() if v.strip()) >= 3:
46             return sections
47
48         # Pattern 2: Alternative format
49         patterns2 = [
50             r"ERROR EXPLANATION:\s*(.*?)(?=SOLUTION 1|$)",
51             r"SOLUTION 1:\s*(.*?)(?=SOLUTION 2|$)",
52             r"SOLUTION 2:\s*(.*?)(?=SOLUTION 3|$)",
53             r"SOLUTION 3:\s*(.*?)$",
54         ]
55
56         explanation_match2 = re.search(
57             | patterns2[0], response, re.DOTALL | re.IGNORECASE
58         )
59         solution1_match2 = re.search(
60             | patterns2[1], response, re.DOTALL | re.IGNORECASE
61         )
62         solution2_match2 = re.search(
63             | patterns2[2], response, re.DOTALL | re.IGNORECASE
64         )
65         solution3_match2 = re.search(
```

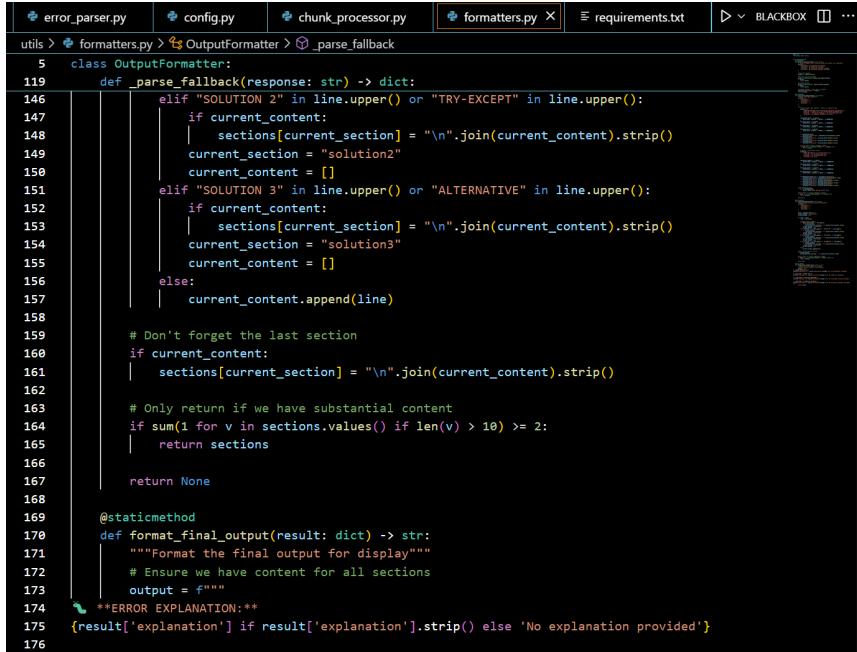


The screenshot shows a terminal window with several tabs open at the top. The current tab is 'formatters.py'. Below the tabs, the code for the `_parse_fallback` method is displayed. The code uses regular expressions to extract sections from a response string. It handles three main sections: 'explanation', 'solution1', 'solution2', and 'solution3'. If none of these sections are found, it returns `None`. A cursor is visible on line 107, where the `except` block begins. To the right of the code, there are several vertical scroll bars and some small preview windows showing other parts of the codebase.

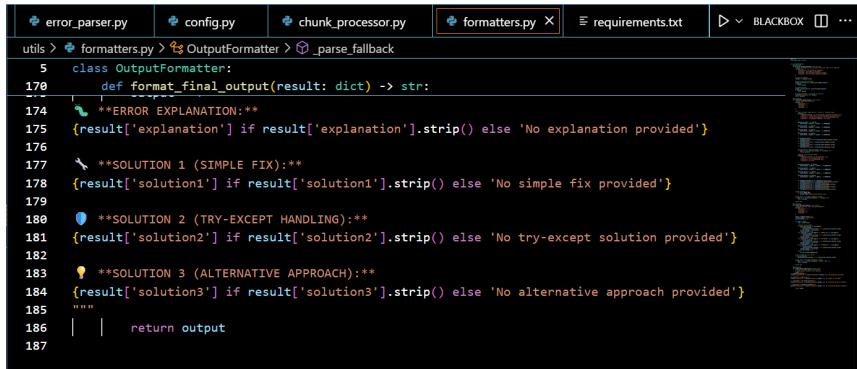
```
5 class OutputFormatter:
6     def _parse_with_regex(response: str) -> dict:
7         solution3_match2 = re.search(
8             patterns2[3], response, re.DOTALL | re.IGNORECASE
9         )
10
11         if explanation_match2 and not sections["explanation"]:
12             sections["explanation"] = explanation_match2.group(1).strip()
13         if solution1_match2 and not sections["solution1"]:
14             sections["solution1"] = solution1_match2.group(1).strip()
15         if solution2_match2 and not sections["solution2"]:
16             sections["solution2"] = solution2_match2.group(1).strip()
17         if solution3_match2 and not sections["solution3"]:
18             sections["solution3"] = solution3_match2.group(1).strip()
19
20         except Exception as e:
21             if Config.DEBUG_MODE:
22                 print(f"Debug: Regex parsing failed: {e}")
23
24         # Only return if we have substantial content
25         if sum(1 for v in sections.values() if v.strip()) >= 2:
26             return sections
27
28     return None
```

The screenshot shows a terminal window with several tabs open. The current tab displays a Python file named `formatters.py`. The code implements a `OutputFormatter` class with a `_parse_fallback` method. This method splits the response into sections based on line-based approach. It then iterates through each line, detecting section headers like "ERROR EXPLANATION" or "SIMPLE FIX" to update the current section and content lists. The terminal also shows the path to the file (`utils > formatters.py`) and some search results for "OutputFormatter" and "parse\_fallback".

```
utils > formatters.py > ↵ OutputFormatter > ⌘_parse_fallback
5   class OutputFormatter:
118     @staticmethod
119     def _parse_fallback(response: str) -> dict:
120         """Fallback parsing using line-based approach"""
121         sections = {
122             "explanation": "",
123             "solution1": "",
124             "solution2": "",
125             "solution3": ""
126         }
127
128         lines = response.split("\n")
129         current_section = "explanation"
130         current_content = []
131
132         for line in lines:
133             if "ERROR EXPLANATION" in line.upper():
134                 # Detect section headers
135                 if "ERROR EXPLANATION" in line.upper():
136                     if current_content:
137                         if current_content[-1].strip() != "\n":
138                             sections[current_section] = "\n".join(current_content).strip()
139                         current_section = "explanation"
140                         current_content = []
141                 elif "SOLUTION 1" in line.upper() or "SIMPLE FIX" in line.upper():
142                     if current_content:
143                         if current_content[-1].strip() != "\n":
144                             sections[current_section] = "\n".join(current_content).strip()
145                         current_section = "solution1"
146                         current_content = []
147                 elif "SOLUTION 2" in line.upper() or "TRY-EXCEPT" in line.upper():
148                     if current_content:
149                         if current_content[-1].strip() != "\n":
```



```
utils > formatters.py > OutputFormatter > _parse_fallback
5     class OutputFormatter:
119         def _parse_fallback(response: str) -> dict:
146             elif "SOLUTION 2" in line.upper() or "TRY-EXCEPT" in line.upper():
147                 if current_content:
148                     | sections[current_section] = "\n".join(current_content).strip()
149                     | current_section = "solution2"
150                     | current_content = []
151             elif "SOLUTION 3" in line.upper() or "ALTERNATIVE" in line.upper():
152                 if current_content:
153                     | sections[current_section] = "\n".join(current_content).strip()
154                     | current_section = "solution3"
155                     | current_content = []
156             else:
157                 | current_content.append(line)
158
159             # Don't forget the last section
160             if current_content:
161                 | sections[current_section] = "\n".join(current_content).strip()
162
163             # Only return if we have substantial content
164             if sum(1 for v in sections.values() if len(v) > 10) >= 2:
165                 | return sections
166
167             return None
168
169     @staticmethod
170     def format_final_output(result: dict) -> str:
171         """Format the final output for display"""
172         # Ensure we have content for all sections
173         output = f"""
174             **ERROR EXPLANATION:**\n
175             {result['explanation']} if result['explanation'].strip() else 'No explanation provided'
176
177             **SOLUTION 1 (SIMPLE FIX):**\n
178             {result['solution1']} if result['solution1'].strip() else 'No simple fix provided'
179
180             **SOLUTION 2 (TRY-EXCEPT HANDLING):**\n
181             {result['solution2']} if result['solution2'].strip() else 'No try-except solution provided'
182
183             **SOLUTION 3 (ALTERNATIVE APPROACH):**\n
184             {result['solution3']} if result['solution3'].strip() else 'No alternative approach provided'
185             """
186
187             | | return output
188
189
```



```
utils > formatters.py > OutputFormatter > _parse_fallback
5     class OutputFormatter:
119         def format_final_output(result: dict) -> str:
174             **ERROR EXPLANATION:**\n
175             {result['explanation']} if result['explanation'].strip() else 'No explanation provided'
176
177             **SOLUTION 1 (SIMPLE FIX):**\n
178             {result['solution1']} if result['solution1'].strip() else 'No simple fix provided'
179
180             **SOLUTION 2 (TRY-EXCEPT HANDLING):**\n
181             {result['solution2']} if result['solution2'].strip() else 'No try-except solution provided'
182
183             **SOLUTION 3 (ALTERNATIVE APPROACH):**\n
184             {result['solution3']} if result['solution3'].strip() else 'No alternative approach provided'
185             """
186
187             | | return output
188
189
```

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

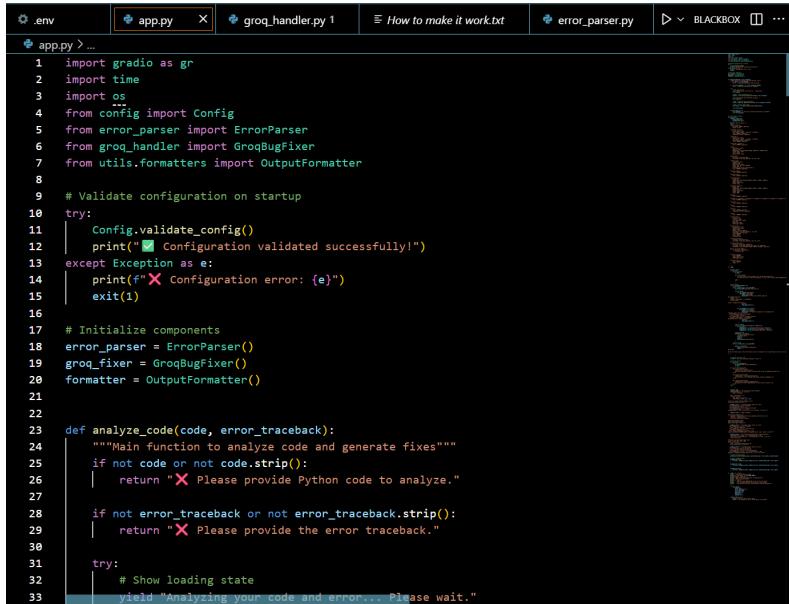
This final activity binds the UI elements to the backend logic, defining the end-to-end user journey and launching the application.

#### 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.

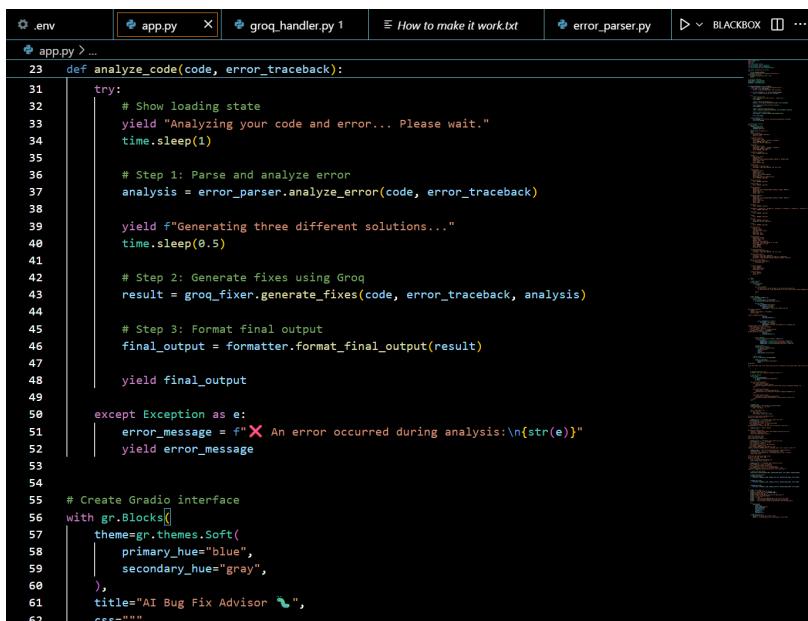
## Code snippets of app.py:



```

. env app.py X groq_handler.py 1 How to make it work.txt error_parser.py BLACKBOX ...
app.py > ...
1 import gradio as gr
2 import time
3 import os
4 from config import Config
5 from error_parser import ErrorParser
6 from groq_handler import GroqBugFixer
7 from utils.formatters import OutputFormatter
8
9 # Validate configuration on startup
10 try:
11     Config.validate_config()
12     print("Configuration validated successfully!")
13 except Exception as e:
14     print(f"X Configuration error: {e}")
15     exit(1)
16
17 # Initialize components
18 error_parser = ErrorParser()
19 groq_fixer = GroqBugFixer()
20 formatter = OutputFormatter()
21
22
23 def analyze_code(code, error_traceback):
24     """Main function to analyze code and generate fixes"""
25     if not code or not code.strip():
26         return "X Please provide Python code to analyze."
27
28     if not error_traceback or not error_traceback.strip():
29         return "X Please provide the error traceback."
30
31     try:
32         # Show loading state
33         yield "Analyzing your code and error... Please wait."
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62

```



```

. env app.py X groq_handler.py 1 How to make it work.txt error_parser.py BLACKBOX ...
app.py > ...
23 def analyze_code(code, error_traceback):
24     try:
25         # Show loading state
26         yield "Analyzing your code and error... Please wait."
27         time.sleep(1)
28
29         # Step 1: Parse and analyze error
30         analysis = error_parser.analyze_error(code, error_traceback)
31
32         yield f"Generating three different solutions..."
33         time.sleep(0.5)
34
35         # Step 2: Generate fixes using Groq
36         result = groq_fixer.generate_fixes(code, error_traceback, analysis)
37
38         # Step 3: Format final output
39         final_output = formatter.format_final_output(result)
40
41         yield final_output
42
43     except Exception as e:
44         error_message = f"X An error occurred during analysis:\n{str(e)}"
45         yield error_message
46
47
48
49
50
51
52
53
54
55
56     # Create Gradio interface
57     with gr.Blocks([
58         theme=gr.themes.Soft(
59             primary_hue="blue",
60             secondary_hue="gray",
61         ),
62         title="AI Bug Fix Advisor 🚀",
63         css="",
64     ]):
65
66         # Add components here
67
68
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
126
127
128
129
129
130
131
132
133
134
135
136
137
138
139
139
140
141
142
143
144
145
146
147
148
149
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
187
188
189
189
190
191
192
193
194
195
196
197
197
198
199
199
200
201
202
203
204
205
205
206
207
207
208
209
209
210
211
212
213
214
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
132
```

```
⚙ .env | ⚙ app.py X | ⚙ groq_handler.py 1 | ⚙ How to make it work.txt | ⚙ error_parser.py | ▶ v BLACKBOX ⌂ ...  
⚙ 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     }  
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%;
```

```
⚙ .env | ⚙ app.py X | ⚙ groq_handler.py 1 | ⚙ How to make it work.txt | ⚙ error_parser.py | ▶ v BLACKBOX ⌂ ...  
⚙ app.py > ...  
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%;
```

The screenshot shows a code editor interface with several tabs open. The tabs include ".env", "app.py", "groq\_handler.py", "How to make it work.txt", "error\_parser.py", and "BLACKBOX". The "app.py" tab is currently active, displaying CSS-like code for styling solution cards. The sidebar on the right contains a tree view of files and a search bar.

```
151 .solution-card {
152     background: #ffffff;
153     border-radius: 15px;
154     padding: 20px;
155     border: 2px solid #667eee;
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 Debugging</p>
197                 </div>
198                 """
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 X ✘ 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         "Analyse & 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 the  
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 X    ⚙ groq\_handler.py 1    ⓘ How to make it work.txt    ⚙ error\_parser.py    ▶ v BLACKBOX    ...

app.py > ...

```
299     users = data.split('\\n')
300     for user in users:
301         name, age = user.split(',')
302         print(f"Name: {name}, Age: {age}")
303
304     # This will cause multiple potential errors
305     process_user_data('users.txt')"""
306
307     example_1_error = """Traceback (most recent call last):
308     File "example.py", line 9, in <module>
309         process_user_data('users.txt')
310     File "example.py", line 2, in process_user_data
311         with open(filename, 'r') as file:
312     FileNotFoundError: [Errno 2] No such file or directory: 'users.txt'"""
313
314     example_2_code = """import requests
315
316     def get_user_data(user_id):
317         response = requests.get(f'https://api.example.com/users/{user_id}')
318         user_data = response.json()
319         return user_data['data'][['email']]
320
321     # Multiple potential issues
322     email = get_user_data(123)
323     print(f"User email: {email}")"""
324
325     example_2_error = """Traceback (most recent call last):
326     File "example.py", line 8, in <module>
327         email = get_user_data(123)
328     File "example.py", line 4, in get_user_data
329         user_data = response.json()
330     File "requests/models.py", line 900, in json
331     requests.exceptions.JSONDecodeError: Expecting value: line 1 column 1 (char 0)"""
332
333     example_3_code = """def calculate_discount(price, discount_percent):
334     discounted_price = price * (1 - discount_percent / 100)
335     final_price = discounted_price + (discounted_price * 0.18) # 18% tax
336     return final_price
337
338     # Multiple calculation and type issues
339     prices = [100, 200, "300", 400]
340     for price in prices:
341         final = calculate_discount(price, 20)
342         print(f"Final price: {final}")"""
343
344     example_3_error = """Traceback (most recent call last):
345     File "example.py", line 9, in <module>
346         final = calculate_discount(price, 20)
347     File "example.py", line 2, in calculate_discount
348         discounted_price = price * (1 - discount_percent / 100)
349     TypeError: unsupported operand type(s) for *: 'str' and 'float'"""
350
351     # Connect the main button
352     analyze_btn.click(fn=analyze_code, inputs=[code_input, error_input], outputs=output)
353
354     # Connect example buttons
355     example1_btn.click(
356         fn=lambda: [example_1_code, example_1_error], outputs=[code_input, error_input]
357     )
358
359     example2_btn.click(
360         fn=lambda: [example_2_code, example_2_error], outputs=[code_input, error_input]
361     )
362
```

⚙ .env    ⚙ app.py X    ⚙ groq\_handler.py 1    ⓘ How to make it work.txt    ⚙ error\_parser.py    ▶ v BLACKBOX    ...

app.py > ...

```
330     File "requests/models.py", line 900, in json
331     requests.exceptions.JSONDecodeError: Expecting value: line 1 column 1 (char 0)"""
332
333     example_3_code = """def calculate_discount(price, discount_percent):
334     discounted_price = price * (1 - discount_percent / 100)
335     final_price = discounted_price + (discounted_price * 0.18) # 18% tax
336     return final_price
337
338     # Multiple calculation and type issues
339     prices = [100, 200, "300", 400]
340     for price in prices:
341         final = calculate_discount(price, 20)
342         print(f"Final price: {final}")"""
343
344     example_3_error = """Traceback (most recent call last):
345     File "example.py", line 9, in <module>
346         final = calculate_discount(price, 20)
347     File "example.py", line 2, in calculate_discount
348         discounted_price = price * (1 - discount_percent / 100)
349     TypeError: unsupported operand type(s) for *: 'str' and 'float'"""
350
351     # Connect the main button
352     analyze_btn.click(fn=analyze_code, inputs=[code_input, error_input], outputs=output)
353
354     # Connect example buttons
355     example1_btn.click(
356         fn=lambda: [example_1_code, example_1_error], outputs=[code_input, error_input]
357     )
358
359     example2_btn.click(
360         fn=lambda: [example_2_code, example_2_error], outputs=[code_input, error_input]
361     )
362
```



The image shows a code editor interface with several tabs open. The active tab is 'app.py'. The code in 'app.py' is as follows:

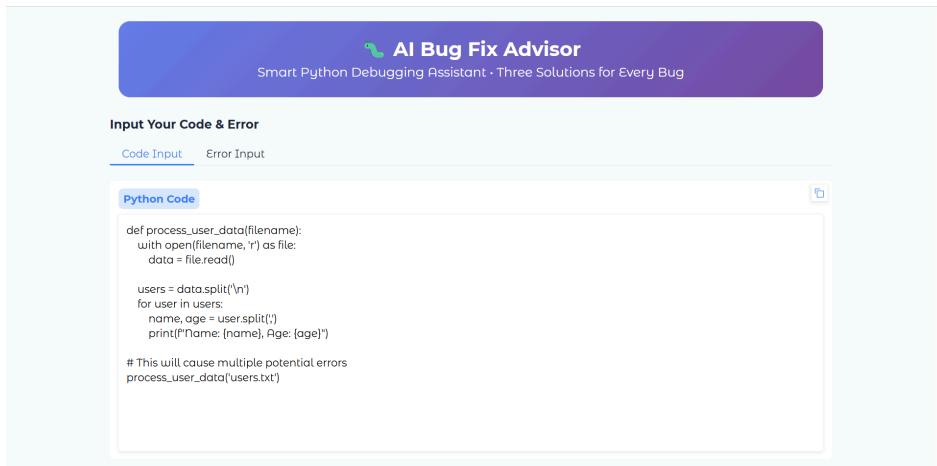
```
363     example3_btn.click()
364     | fn=lambda: [example_3_code, example_3_error], outputs=[code_input, error_input]
365     )
366
367 if __name__ == "__main__":
368     print("🚀 Starting AI Bug Fix Advisor...")
369     print(f"💡 Using model: {Config.MODEL_NAME}")
370     print("🌐 Server starting...")
371     print("👉 Open the URL below to access the application:")
372     print("  http://localhost:7860")
373     print("\n💡 Tips:")
374     print("  - Make sure your GROQ_API_KEY is set in the .env file")
375     print("  - Try the example buttons to quickly test the system")
376     print("  - Each analysis provides three different solution approaches")
377
378 try:
379     demo.launch(
380         share=False,
381         server_name="0.0.0.0",
382         server_port=7860,
383         show_error=True,
384         show_api=False,
385         quiet=False,
386         inbrowser=True,
387     )
388 except Exception as e:
389     print(f"🔴 Failed to start server: {e}")
390     print("💡 Try changing the port: python app.py --port 7861")
```

Figure 12: app.py

# Output Preview

## Example 1: File Processing

Code:



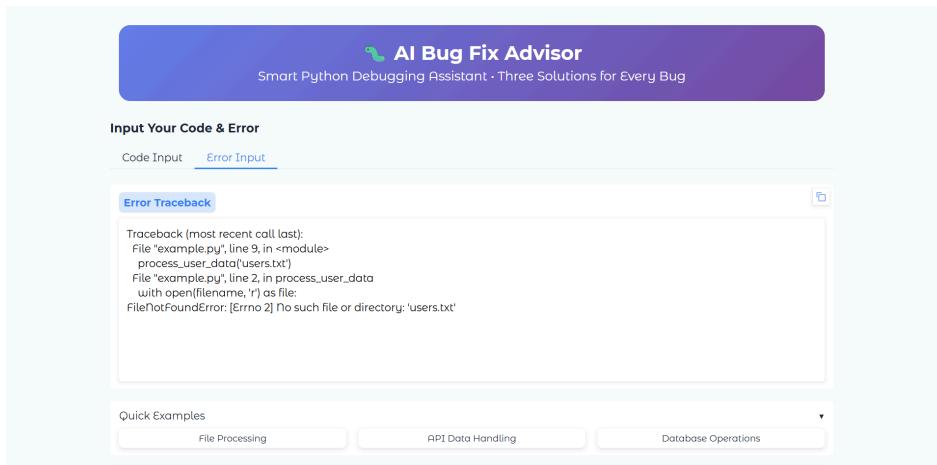
The screenshot shows the AI Bug Fix Advisor interface. At the top, a purple header bar displays the logo and text "AI Bug Fix Advisor" and "Smart Python Debugging Assistant - Three Solutions for Every Bug". Below the header, a light blue input field is labeled "Input Your Code & Error". Underneath the input field, there are two tabs: "Code Input" (which is selected) and "Error Input". A large text area titled "Python Code" contains the following code:

```
def process_user_data(filename):
    with open(filename, 'r') as file:
        data = file.read()

    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')
```

Error:



The screenshot shows the AI Bug Fix Advisor interface. At the top, a purple header bar displays the logo and text "AI Bug Fix Advisor" and "Smart Python Debugging Assistant - Three Solutions for Every Bug". Below the header, a light blue input field is labeled "Input Your Code & Error". Underneath the input field, there are two tabs: "Code Input" and "Error Input" (which is selected). A large text area titled "Error Traceback" displays the following traceback:

```
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'
```

At the bottom of the interface, there is a "Quick Examples" section with three categories: "File Processing", "API Data Handling", and "Database Operations".

## 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 `FileNotFoundException`. 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}")

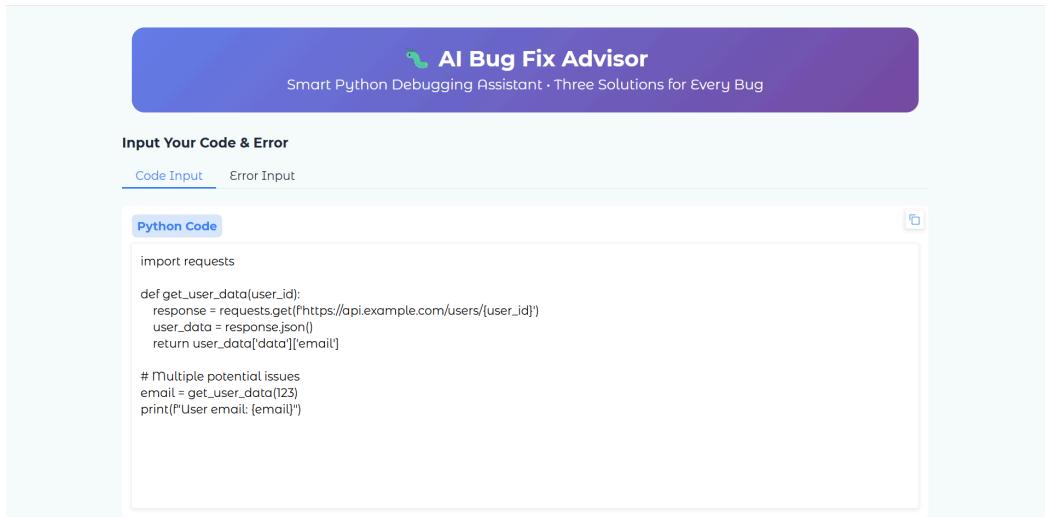
process_user_data('users.txt')
```

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:



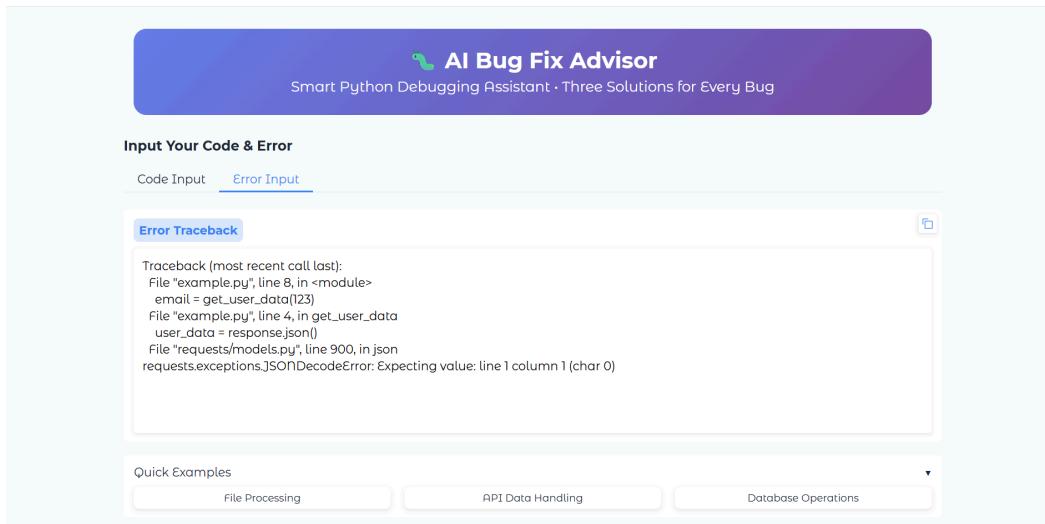
The screenshot shows the AI Bug Fix Advisor interface. At the top, there's a purple header bar with the title "AI Bug Fix Advisor" and the subtitle "Smart Python Debugging Assistant · Three Solutions for Every Bug". Below the header, there's a section titled "Input Your Code & Error". Under this, there are two tabs: "Code Input" (which is selected) and "Error Input". A large text area labeled "Python Code" contains the following Python 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}")
```

### Error:



The screenshot shows the AI Bug Fix Advisor interface. At the top, there's a purple header bar with the title "AI Bug Fix Advisor" and the subtitle "Smart Python Debugging Assistant · Three Solutions for Every Bug". Below the header, there's a section titled "Input Your Code & Error". Under this, there are two tabs: "Code Input" and "Error Input" (which is selected). A large text area labeled "Error Traceback" contains the following Python error message:

```
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)
```

At the bottom of the interface, there's a "Quick Examples" section with three buttons: "File Processing", "API Data Handling" (which is selected), and "Database Operations".

## 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

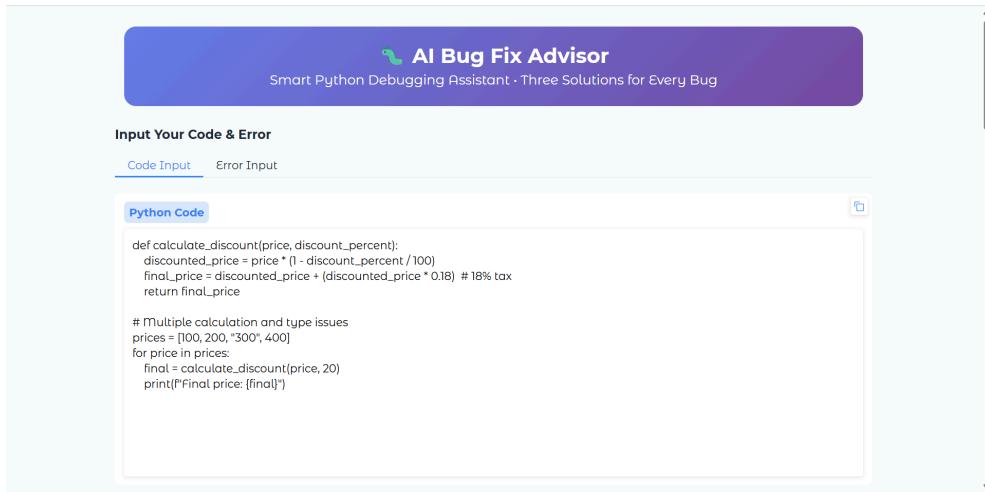
class UserData(BaseModel):
    data: dict

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

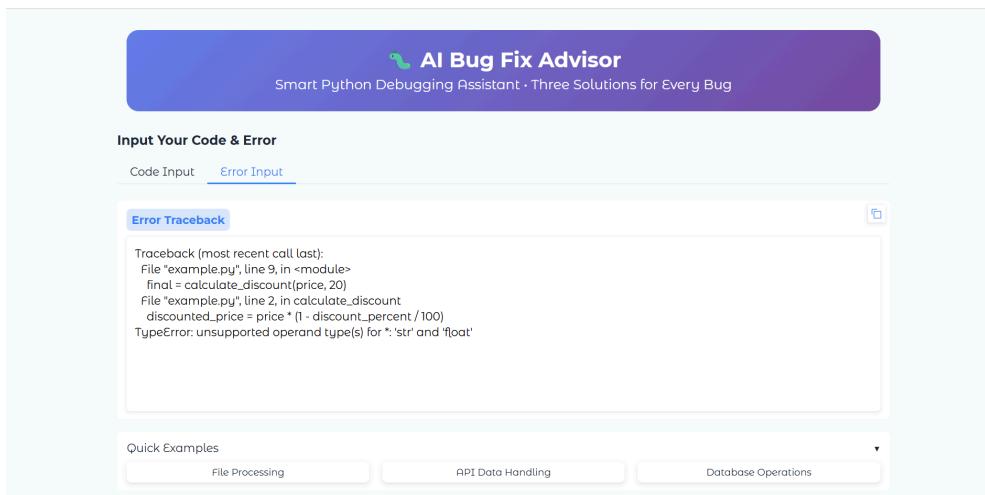
Code:



The screenshot shows the AI Bug Fix Advisor interface. At the top, there's a purple header bar with the title "AI Bug Fix Advisor" and the subtitle "Smart Python Debugging Assistant - Three Solutions for Every Bug". Below the header, there's a section titled "Input Your Code & Error" with two tabs: "Code Input" (which is selected) and "Error Input". Under the "Code Input" tab, there's a code editor window with the title "Python Code". The code in the editor is: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}")

```
This code defines a function to calculate a discount and then applies it to a list of prices, including a string value. It also includes a print statement.
```

Error:



The screenshot shows the AI Bug Fix Advisor interface. At the top, there's a purple header bar with the title "AI Bug Fix Advisor" and the subtitle "Smart Python Debugging Assistant - Three Solutions for Every Bug". Below the header, there's a section titled "Input Your Code & Error" with two tabs: "Code Input" and "Error Input" (which is selected). Under the "Error Input" tab, there's a code editor window with the title "Error Traceback". The tracebacks in the editor are: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'

```
This shows a TypeError occurring due to the multiplication of a string and a float in the original code.
```

## 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:
        print(f"Error: {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.