Intel Unnati Industrial Training Program 2025

Karunya Institute of Technology and Science

Software Code Bug Detection & Fixing

Faculty Mentor: Dr. Daniel Madan Raja

Team members:

David Rosario Selvaraj (URK22CS1007)

Shannan Dhirshath T (URK22CS1045)

C Joshan Sam (URK22CS1078)

Software Code Bug Detection & Fixing

Abstract:

This project presents a VS Code extension for automated code bug detection and fixing, developed for the Intel Unnati Industrial Training Program 2025. Leveraging deep learning and generative AI, it detects functional python bugs and suggests syntactically and semantically correct fixes. The system integrates open-source models like Mistral via Hugging Face API, Optimized for GPU-limited environments while adhering to project constraints.

Designed for seamless real-time code analysis, the extension provides a dedicated UI panel to initiate bug detection, display diagnostic insights, and present AI-generated fixes. By reducing debugging time and improving code quality, this tool enhances developer productivity and streamlines the software development process using cutting-edge AI technologies.

Setup:

1. Install the compiled Extension:

Method 1: Using VS Code UI

1. Open VS Code
2. Go to Extensions (Ctrl+Shift+X or Cmd+Shift+X on Mac)
3. Click More Actions (⁝) → Choose Install from VSIX...
4. Select the downloaded .vsix file from the project folder

Method 2: Using the Command Line

Alternatively, install the extension via the terminal:

```
code --install-extension <path-to-vsix-file>
```

2. Run the Extension:

1. Open VS Code
2. Press Ctrl+Shift+P (or Cmd+Shift+P on Mac) to open the command palette
3. Search for "Bugfixer Extension" in the command palette
4. Press Enter

3. Set Up the Hugging Face API Key:

1. Press Ctrl+Shift+P (or Cmd+Shift+P on macOS) to open the Command Palette.
2. Type "Set Hugging Face API Key" and select it from the list.
3. Alternatively, you can proceed by pressing the "Check" button in the extension panel.
4. The user is prompted to enter a Hugging Face access token (must start with hf_...). Generate it from Hugging Face, paste it, and press Enter to securely store it in VS Code.

The extension is now active. Users can click 'Check' in the left panel to analyze and fix code issues in VS Code. By ensuring cleaner commits, the extension enhances code quality, supporting a smoother CI/CD workflow.

Methodology:

This project employs a zero-shot learning approach using a pre-trained generative AI model to detect and fix functional bugs in python code. It integrates seamlessly with Visual Studio Code to enhance code quality and developer efficiency.

1. Zero-Shot Bug Detection and Fixing

- The system uses **Mistral-7B-Instruct-v0.3**, accessed via the Hugging Face Inference API.
- Given a code snippet from the VS Code editor, the extension sends a prompt to the model instructing it to return a structured **JSON** response containing:
    - **Bug Status** (e.g., buggy or clean),
    - **Type of Bug** (e.g., syntax, logic, etc.),
    - **Fixed Code** (corrected version of the original input).

2. VS Code Extension Workflow

- The extension captures the active code from the editor when the user initiates analysis.
- It sends this code to the inference API with a pre-defined prompt that instructs the model to return only the required structured JSON output.
- Once the response is received, the extension:
    - Parses the JSON,
    - Displays the **bug status**, **bug type**, and **AI-generated fix** in a custom UI panel,
    - Offers an option to **integrate the fix** directly into the code editor for seamless correction.

3. Real-Time Usability

- The system delivers real-time results by leveraging a highly optimized inference workflow, ensuring efficient analysis and response without added latency.
- The architecture prioritizes developer productivity, enabling quick diagnosis and correction within the coding workflow.

Model selection (Mistral-7B-Instruct-v0.3):

This project utilizes the **Mistral-7B-Instruct-v0.3** model hosted on Hugging Face for zero-shot bug classification and fix generation. The model was selected for its strong performance in code understanding and instruction-following tasks, which aligns well with the project's objective of detecting and resolving functional code issues without any fine-tuning.

The integration is handled through Hugging Face’s Inference API, allowing the extension to offload computation and benefit from scalable inference capabilities. This design supports flexibility: the selected model can be changed easily by modifying a single line in the Bug-Fixer-Extension/src/api.ts file. Specifically, the model identifier:

```
const model = "mistralai/Mistral-7B-Instruct-v0.3";
```

Efficient Use of API & Call Optimization:

To ensure optimal performance and minimize unnecessary API usage, the extension employs a demand-driven interaction model. API calls to Hugging Face are triggered only when the user explicitly initiates a bug analysis through the interface. This approach prevents redundant or continuous requests during development, conserving both bandwidth and API quota.

Additionally, the system validates the presence of a stored API key before making any inference requests. The key is securely stored using Visual Studio Code's SecretStorage mechanism, eliminating repeated prompts and streamlining the user experience. The API logic is centralized in a single module (Bug-Fixer-Extension/src/api.ts), making it easy to audit, control, and update usage patterns.

By avoiding polling, auto-triggered scans, or background analysis, the design ensures that resources are used efficiently and inference costs remain predictable, especially in resource-constrained environments. This makes the extension both scalable and cost-effective for individual developers and teams.

VS Code Extension Architecture

The bug detection and fixing workflow is powered by a well-organized Visual Studio Code extension that facilitates seamless real-time interaction with a large language model via API.

1. Extension Activation

Upon installation, the extension registers three core commands:

- bugFixer.detectBugs: Launches the main bug detection and fix UI panel.
- bugFixer.setApiKey: Allows users to securely input and store their Hugging Face API key.
- bugFixer.resetApiKey: Deletes the stored API key when needed.

These commands are defined in extension.ts, which also wires up the secure storage mechanism for the API key using VS Code's native SecretStorage.

2. Webview Panel Interface

When the user triggers the bug detection command, a dynamic UI is rendered using the BugFixPanel class in panel.ts. This panel:

- Displays a "Check" button.
- Triggers bug analysis on click by sending the current code in the editor to the backend logic.
- Displays the bug status, the detected bug type, and the model-generated fixed code in visually distinct sections.
- Includes an "Integrate Fix" button, enabling users to replace the existing code with the fixed version directly inside the editor.

3. Editor & Code Handling

The extension is designed to always work on the currently focused editor window. If no editor is immediately found, the extension attempts to re-focus the active editor group and ensures valid interaction before proceeding.
Once the code is fetched from the editor:

- It is sent to the Hugging Face model via the queryHuggingFace() function in api.ts.
- The model is instructed to return a JSON object with bug_status, bug_type, and fixed_code.

4. Backend Communication & Model Query

The api.ts module handles all communication with the Hugging Face Inference API. It abstracts the API call logic and automatically prompts the user for an API key if not already stored. The model in use is: mistralai/Mistral-7B-Instruct-v0.3. This can be swapped with any other Hugging Face model by changing the model value in api.ts, making the architecture flexible and easily upgradable.

5. Code Integration & Update

When the user accepts the suggested fix:

- The extension calculates the full range of the current code.
- Replaces it with the fixed version returned by the model.
- displays a success message.

Security & API Key Management:

1. Secure Token Storage Using VS Code SecretStorage

The API key is securely stored using Visual Studio Code’s native SecretStorage API, which encrypts and saves sensitive data within the user’s VS Code environment. This ensures the token is:

- **Persistent** across sessions.
- **Protected** from unauthorized access.
- **Isolated** within the extension’s own secret scope.

All storage and retrieval operations are done in-memory during runtime, ensuring no token is exposed in source files or logs.

2. Token Setup & Management Commands

The extension offers a user-friendly interface to manage their token lifecycle with three commands:

- **Automatic Token Handling in API Calls:**
  The queryHuggingFace() function checks for a stored token before making any API request. If no token is found, the extension proactively prompts the user to set one, enforcing a secure and authenticated access pattern.

- **bugFixer.setApiKey:**
  Prompts the user to input their Hugging Face token via a secure VS Code input box. Once entered, the token is immediately encrypted and stored using context.secrets.

- **bugFixer.resetApiKey:**
  Deletes the stored token securely. This allows users to revoke and remove access at any time.

3. Token Scope Principle

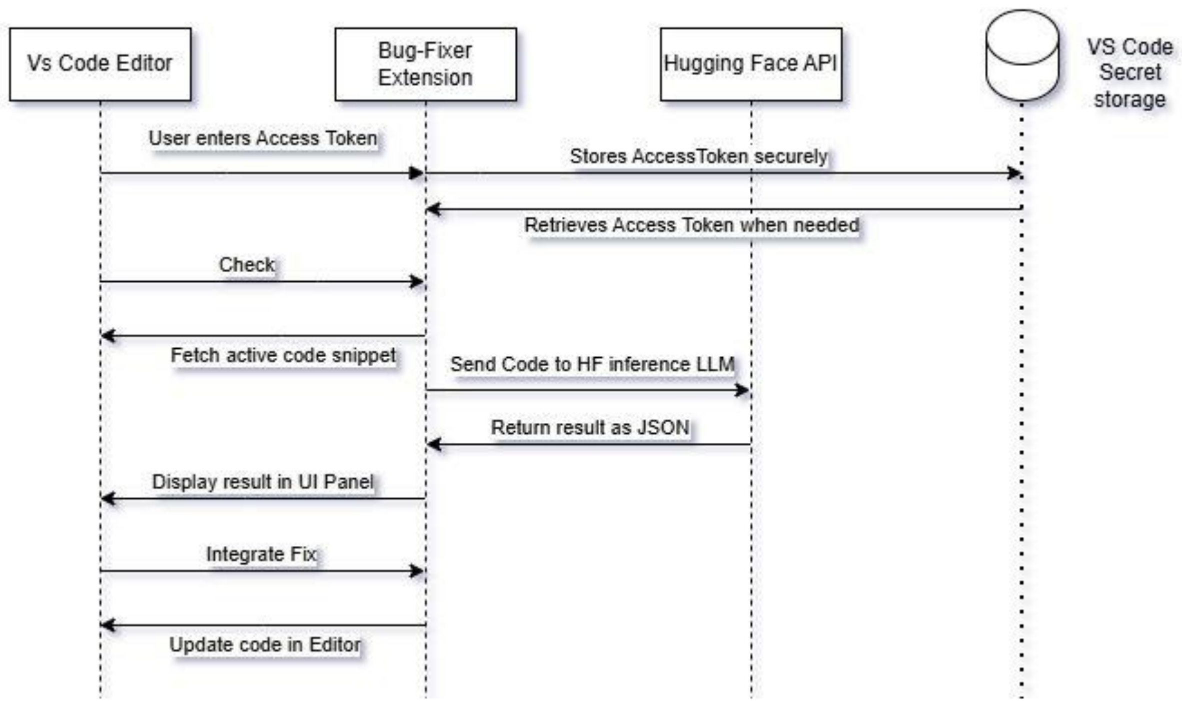
Users are encouraged to generate **read-only or limited-scope tokens** from Hugging Face that are restricted to inference, ensuring that even in the unlikely event of compromise, access is limited.

Summary of Technologies Used:

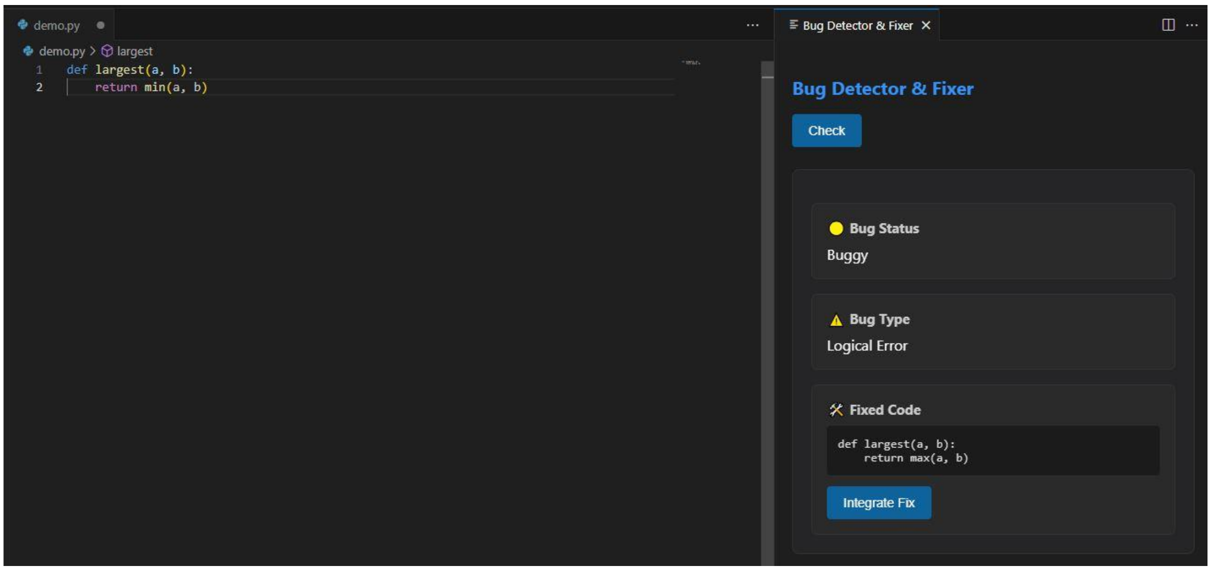
- **VS Code API**: For UI, editor access, and secure storage.
- **Hugging Face Inference API**: For zero-shot bug detection and fixing using a large language model.
- **Node Fetch**: To make HTTP requests from the extension backend.
- **Webview**: For an interactive and intuitive UI experience.

This streamlined, modular architecture ensures a smooth user experience with minimal overhead, while offering flexibility to adapt the backend model or extend the workflow in future iterations.

Sequence Diagram:



Sample Working:



```python
def largest(a, b):
    return min(a, b)
```

Bug Detector & Fixer

● Bug Status
Buggy

⚠ Bug Type
Logical Error

Fixed Code

```python
def largest(a, b):
    return max(a, b)
```

Conclusion:

By integrating secure API key management, efficient inference workflows, and a streamlined UX/UI, this extension enhances developer productivity by reducing debugging time and improving code quality—all while operating efficiently in resource-constrained environments. The system is designed to be scalable, cost-effective, and adaptable, allowing developers to swap models easily, optimize API calls, and maintain security without compromising performance.