System Design for GenAI Test Bot

Version 3

Group 2

Neha Asthana
Lawrence Gu
Nathan Peereboom
Bahar Abbasi Delvand
Krishika Mirpuri

April 1st 2024

TABLE OF CONTENTS

1 Revisions
2 Purpose
    2.1 System Purpose
    2.2 Document Purpose
3 Scope
4 Project Overview
    4.1 Normal Behaviour
    4.2 Component Diagram
    4.3 Connection Between Requirements and Design
    4.4 Components
    4.4.1 Core test generation module
    4.4.2 Web server
    4.4.3 VSCode Extension Client
5 Components and requirements
6 User Interface
7 Timeline
8 Appendix - Interface

1 Revisions

Version #	Description	Date
0	Original design document	26/01/2024
1	Design overhaul	09/02/2024
2	Added tables for function methods	16/02/2024
3	Final Version	01/04/2024

2 Purpose

2.1 System Purpose

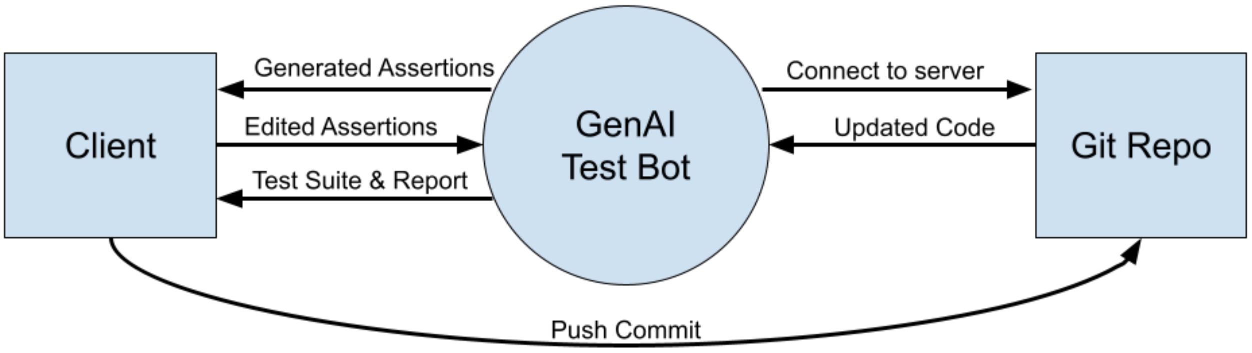
The goal we are working to achieve with the GenAI Test Bot is to create an automated unit test generation tool that can be integrated seamlessly into a developer's workflow, removing much of the manual labour that comes with testing code. The generated unit tests and other code analyses will used to evaluate the user's code on the following criteria:

1. **Reliability:** Does the code compile?
2. **Correctness:** Does the code give the expected results?
3. **Coverage:** Are all branches of the code utilised?
4. **Maintainability:** Is the code readable and written in a way to support future modifications?
5. **Efficiency:** Does the code run in an efficient and timely manner?

2.2 Document Purpose

The purpose of this design document is to lay out the different components of our system and their interactions in an easily digestible manner so as to clarify details about the inner workings of the system and to serve as reference for anyone working on or with the system.

3 Scope



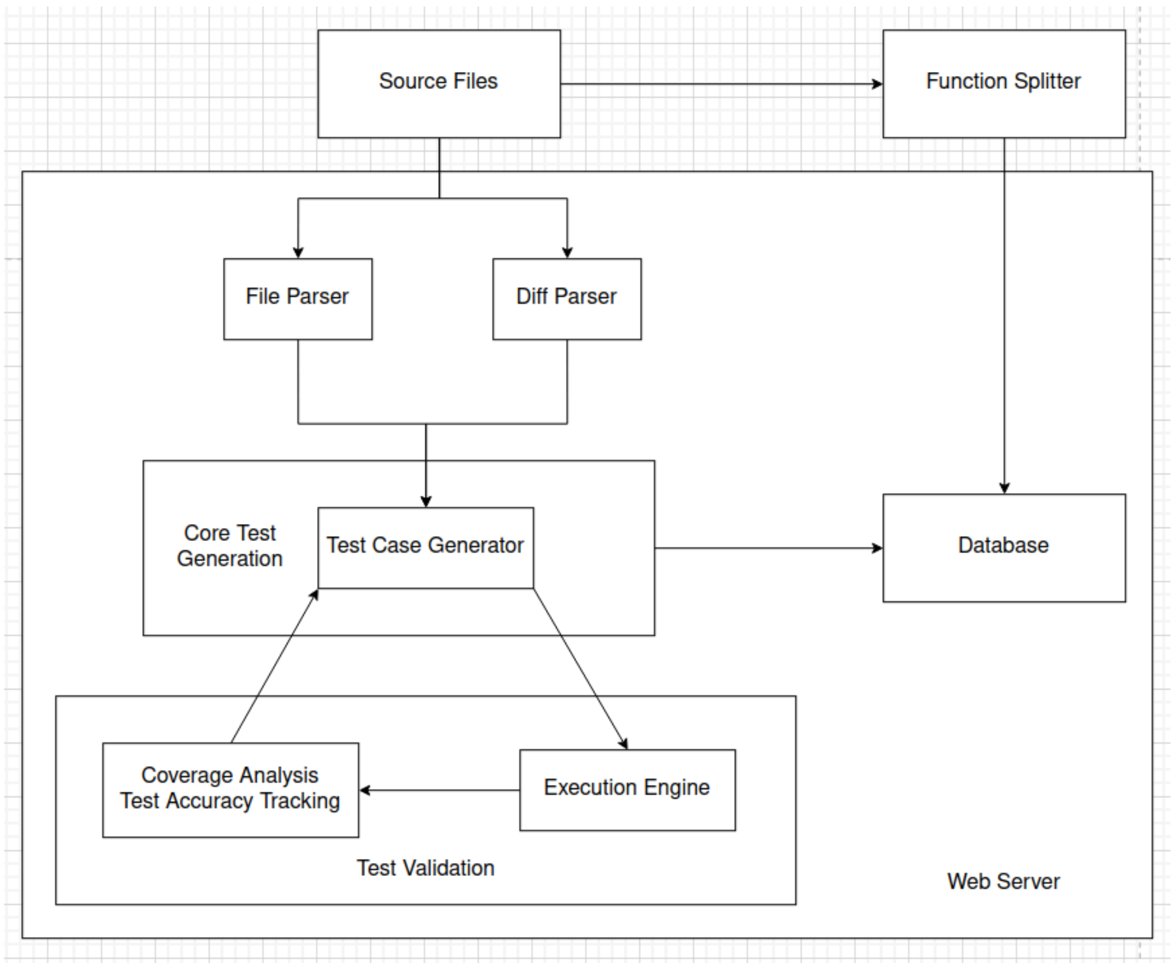
4 Project Overview

4.1 Normal Behaviour

The Gen AI Test Bot seamlessly operates under normal conditions, catering to users in the generation of comprehensive test cases for Python and FastAPI codebases. Employing advanced AI models like GPT-4, the bot facilitates an efficient workflow, incorporating automated documentation generation, interactive API documentation, and seamless integration with CI/CD pipelines. Users can interact with the bot to generate test cases based on their code, and the automated documentation provides valuable insights, streamlining the testing process for developers

4.2 Component Diagram

We have decomposed our system into the following components and sub-components:



Note: The database component is exclusively utilised for data storage and retrieval by multiple components. Subsequent modules will specify when a component interfaces with the database.

4.3 Connection Between Requirements and Design

- The first ‘Look and Feel Requirement’ we had was that the system would be accessible from a webpage, which we have kept in mind as we’ve designed our SPA.
- We kept the minimalist design in mind, so that the user does not have to learn much or be bombarded by many features at once. Most features are self-explanatory, and are decorated with a text which describes their functions, or what the box wants from or is presenting to the user.
- Additionally, we incorporated a section for documentation, should the user need further clarification, as well as a readMe for the extension in Visual Studio Marketplace clarifying use of it.
- Users will get a report describing accuracy, static and coverage rating of the test cases we developed for the user, and they will be presented with the test cases as well. They will have the option to regenerate the test cases by refreshing the browser, should they be unsatisfied with the code coverage and the percentage of successful test cases.

- The report page also includes a button that opens a pop-up with additional documentation explaining what each component on the report means.
- The report generated includes all the crucial metrics that the user would like to know, and gives the user tangible and comprehensible statistics.

4.4 Components

Below is a chart showing how the modules were decomposed.

Name	Responsibility
Core test generation module	Take code commit differential and generate a unit test suite for it
Web server	Host other modules, including core test generation, parsers, database and test validation; interact with the client to facilitate user flow
VSCode extension client	Bridge the user and the server; integrate with the VSCode environment
Test validation module	Execute generate test cases; validate their compliance with coverage and accuracy standards

4.4.1 Core test generation module

Input Name	Input Type	Description
code	str	The source code to be tested
diff	bool	Whether the given code is a result of “git diff”
model	str	Either “3.5” or “4”, referring to the GPT model to use
error	str	Error message after executing the generated test cases
coverage	str	The percentage of statements covered returned by the Test Validation module

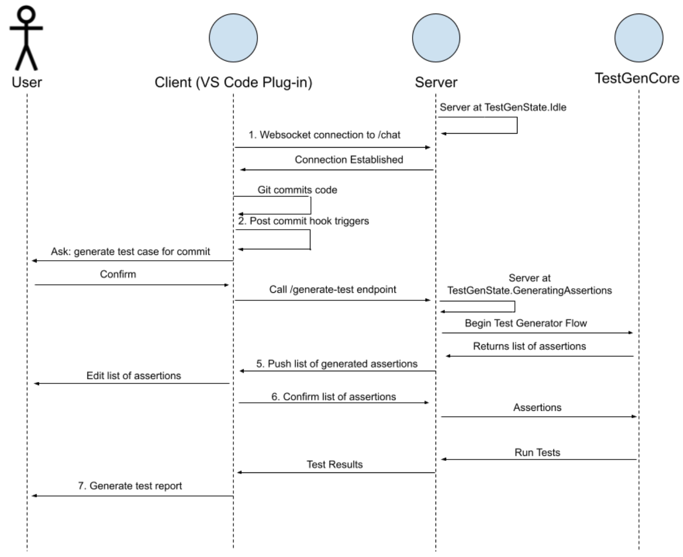
Output Name	Output Type	Description
test_suite	str	Test cases generated by GPT for the given source code

a. Normal Behaviour: This component will take a diff of a code commit and generate a unit test suite for it using a series of LLM prompts.
b. API: The latest git commit is the input and the module outputs the test cases generated by the LLM.The code base will link to the user's Visual Studio Code Workspace and lookup their linked GitHub repository, which will then be used to pass built-in GitHub APIs that will fetch the latest commit of the repository to pass to the test generator.
c. Potential Undesired Behaviours: LLM may generate incomplete or inaccurate test cases, and the component might produce different test cases for the same input which shows inconsistency. Excessive re-running of the generator could lead to overconsumption of resources.

4.4.2 Web server

a. Normal Behaviour: Our web server hosts the core test generation module, the parsers, the test validation module, and the database. This component interacts with the client, through pop-up messages and console logs, to implement the user flow.
b. API: The web server does not have any specific inputs and outputs as it is made up of a series of endpoints, specifically the init, sync, generate and chat endpoints (further detailing alongside the sequence diagram below). The init and sync endpoints pull the repository information from the database to create its clone and update the clone with the latest commits stored in the database.
c. Potential Undesired Behaviours: Users can face issues with Websocket communication failures that could lead to disrupted real-time interactions between the user and the server, and syncing difficulties which could lead to incorrect or outdated test generation content.

The following is a sequence diagram that describes our current test generation user flow:


User
Client (VS Code Plug-in)
Server
TestGenCore
Server at TestGenState.Idle
1. Websocket connection to /chat
Connection Established
Git commits code
2. Post commit hook triggers
Ask: generate test case for commit
Confirm
Call /generate-test endpoint
Server at TestGenState.GeneratingAssertions
Begin Test Generator Flow
Returns list of assertions
5. Push list of generated assertions
Edit list of assertions
6. Confirm list of assertions
Assertions
Run Tests
Test Results
7. Generate test report

The functionalities of the web server are separated into the following REST API endpoints, and are exposed to our client component described in the **next** section.

a. **/init**:

   Mirrors a local user repository on our server.

b. **/sync**

   Syncs with the remote copy of the code base by uploading a diff of the latest local user commit. Triggered via a post commit hook on the client side. We assume that our mirror is at most 1 commit behind the user's local copy.

c. **/generate-test**

   Starts the test generation flow, which kicks off /chat by pushing a Websocket message to the client by asking them to confirm the test assertions.

d. **/chat**

   Websocket endpoint on the server. The current chat flow works as follows:

   - Once the extension is loaded, users will run the Gaib: Edit Assertions to open an editable file on the interface that they must enter at least three sample assertions into.
   - Once that is done, they can run the command GAIB: Generate Assertions to generate additional test cases with our bot.
   - The next step is to run GAIB: Run Tests and it will run tests on generated test cases to validate them.
   - Finally, the user has the option to run the command GAIB: View Report to run analysis on the generated test cases that will load on the Website's Main page.
   - Users will receive confirmation and transition messages guiding them through this process to imitate a chat workflow as they submit commands and receive feedback.

4.4.3 VSCode Extension Client

a. Normal Behaviour: The VSCode Extension Client acts as a bridge between the user and the server, facilitating the initialization of Websocket connections, detecting Git workspace openings, and interacting with the server to mirror repositories, sync commits, generate tests, and handle the UI aspects of the chat workflow. It integrates with the VSCode environment to provide a seamless user experience.
b. API: The VSCode Extension Client does not have any specific inputs and outputs as it is made up of a series of endpoints, specifically the init websocket, repo init, post commit hook, generate and chat endpoints (further detailing below). Any updates to the repository, user information or interaction will be stored in the database by the post commit hook and the generated endpoints.
c. Potential Undesired behaviour: Undesired behaviours in the VSCode Extension Client may involve WebSocket initialization problems, difficulties in repository detection and initialization, failures in post-commit hook installation affecting automatic diff uploads, issues with user notification handling for test generation, and potential UI communication problems during the chat workflow, impacting smooth user-server interaction and hindering synchronisation and test generation

processes.

This component is responsible for:

a. **Init Websocket**: initialises a websocket connection with the server
b. **Repo Init**: Detecting when a Git workspace is opened within VSCode, and initialises it with the remote repository via the /init method. A Github access token is requested via the VSCode user interface. The detection of the Git repository can be implemented via VSCode APIs, by looking for the presence of .git file in the current workspace
c. **Post Commit Hook**: During the repo init phase, the extension will also install a Git post-commit hook inside the folder. This post commit hook, when triggered by a commit, will upload a diff of the commit to the /sync endpoint described in the Web server above
d. **Generate Test**: After the post-commit hook finishes running (server responds), pop a notification to the user asking if they would like to generate a test for the current commit. This will hit the /generate-test endpoint on the web server described above
e. **Chat:** The VSCode extension client will be responsible for implementing the UI side of the chat workflow described above, where user input through commands is replied to with responsive and guiding messages.

4.4.4 Test Validation Module

Input Name	Input Type	Description
filepath	str	The path to the file containing the Python code to be tested
src	str	The name of the file containing the Python code to be tested
i	int	To track how many times the cycle of generating test cases -> running the test cases -> fixing error has been executed
output_file	path-like object	The path to the file that contains the error message after executing generated test cases

Output Name	Output Type	Description
error	str	An error message after executing the generated test cases, assuming that such error is present
metrics	str	Detailed report generated by coverage.py that indicates insufficient coverage

a. Normal Behaviour: The Test Validation Module executes the test cases produced by the core test generator, validating their compliance with coverage, accuracy, class, and coding standards. It evaluates whether these methods are covered by the generated test cases through coverage analysis, static code analysis, equivalent partitioning and test accuracy tracking.
b. API: The Test Validation Module assesses generated test cases, verifying their adherence to coverage, accuracy, class, and code standards. It provides a percentage of tests meeting the set standards and identifies failures for refinement by the core test generator.
c. Potential Undesired behaviour: Undesired behaviours in the Test Validation Module may include issues related to database connectivity, inaccuracies in coverage analysis, errors in static code analysis, inadequate test partitioning, inaccurate test accuracy metrics, integration problems with VSCode plugins, user interface inconsistencies, server setup errors, difficulties in diff parser integration, and challenges with chatbot interactions.

This component is responsible for:

a. **Coverage Analysis**: Utilises a coverage matrix to assess code coverage achieved by the test suites.
b. **Static Code Analysis**:Implements static code analysis techniques to enhance code quality using 'pylint'.
c. **Equivalent Partitioning**:Utilises equivalent partitioning testing methodologies to design comprehensive test cases.
d. **Test Accuracy tracking**: Monitors and reports the accuracy of test cases.

5 Components and requirements

Component	Requirements
Core Test Generation Module	Accuracy and reliability Error handling Runtime efficiency Financial Constraint 6.3, 7.1, 7.3, 8.3 (SRS)
Web Server	Integration with GitHub hook 6.7 (SRS)
VSCode Extension Client	Programming language support UI design 6.4, 7.3, 8.1 (SRS)
Test Validation Module	Accuracy and reliability 90% Compile rate 90% Code coverage Error handling Runtime efficiency 6.3, 6.5, 6.6, 7.1, 7.3 (SRS)

6 User Interface

(Visual representation is provided in the appendix)

Revised: The color scheme features a dark gradient background (`background: rgb(3,6,55); background: linear-gradient(rgba(3,6,55,1) 13%, rgba(10,47,84,1) 100%);`) to uphold a programmer-friendly dark mode. We've chosen to utilize gradient formatting to mitigate the monotony of a solid background. Clickable buttons are designated by colors such as #1976D2, #D40000, or #9C27B0 to provide contrast against the dark background. Text against these buttons or the dark background is #EFFCFF, offering a softer alternative to white. 'Back' buttons and text boxes also utilize #EFFCFF. These colors are consistently applied throughout our frontend to maintain thematic coherence.

1. Initial Encounter:

- When users first visit our website, they'll come across a Landing Page (Fig 1). Here, they'll find options to either register or login to their account, download our extension, or explore our documentation (Fig 2), which provides a brief overview of our product's functions and features.

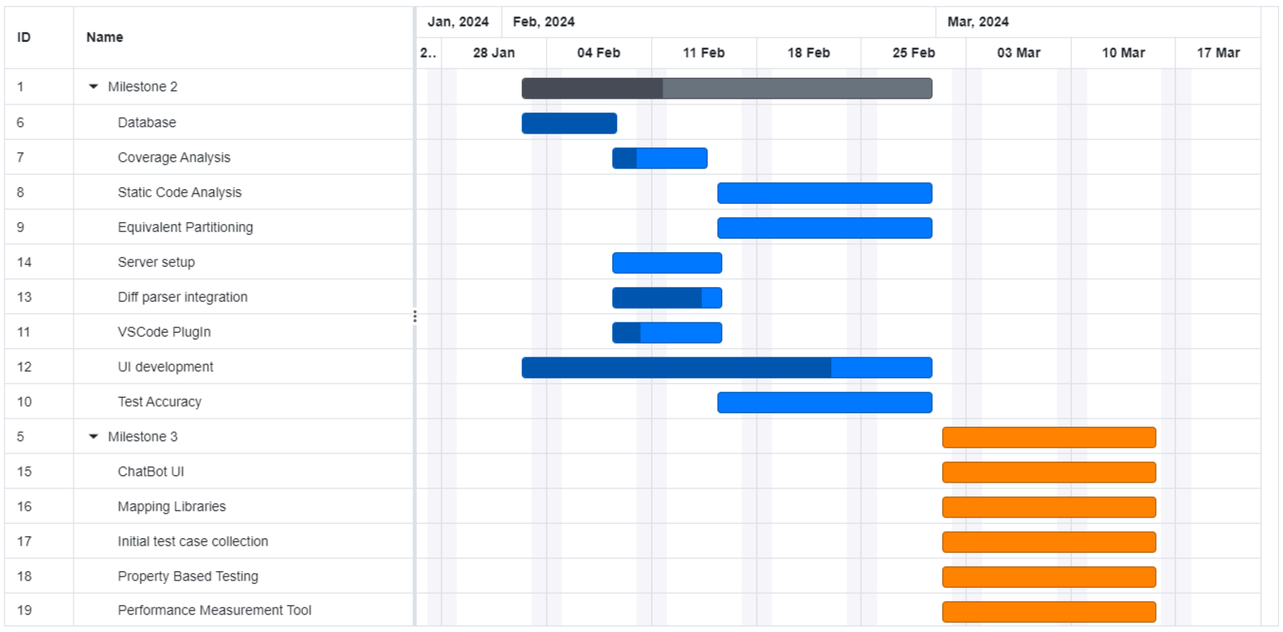
2. Registration (Fig 3) and Login (Fig 4):

- Once users decide to register or login, they can easily switch between these pages by clicking the blue links at the bottom. Additionally, they can navigate back to the landing page using the "back" button located at the top right corner. Users can input their information to either login or register.
- Please be aware that users will receive an error message displayed in a snackbox at the bottom of the page if they leave any textboxes unfilled, input an incorrect email format, attempt to register with an email that's already in use, enter the wrong password for login, or if their passwords don't match during registration.

3. Stats/Main Page:

- After users enter their credentials, they are directed to their main page. Each user is assigned a unique URL based on their user ID stored in our MongoDB database. Here, they can view various statistics related to the code they ran on the extension.
- Additionally, there is a button at the bottom that provides explanations for each specific statistic.
- It must be noted that the main page will appear blank until tests have been generated using the extension at least once.

7 Timeline

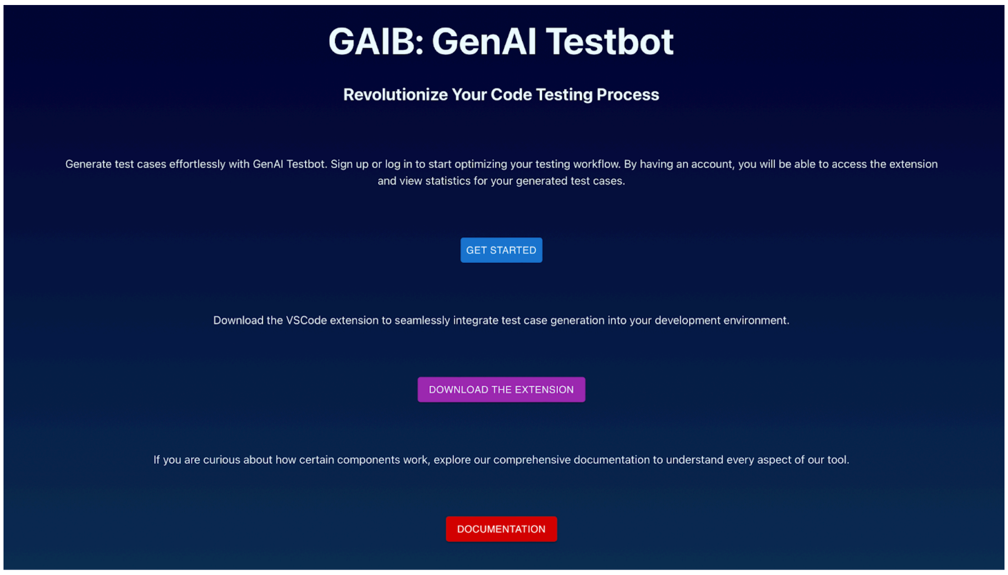


Task Name	Team Member
Milestone 2	
Set Up Database and Connect to Server	Neha and Bahar
Coverage Analysis	Lawrence
Connect Diff Parser	Krishika
Static Code Analysis	Neha and Nathan
VSCode Plugin	Krishika and Bahar
Server Setup	Nathan
Equivalent Partitioning	Lawrence
UI development	Bahar
Test Accuracy	Nathan and Lawrence
Milestone 3	
Chatbot UI	Krishika and Bahar
Mapping Libraries	Nathan and Lawrence
Initial Test Case Collection	Neha
Property Based Testing	Nathan and Lawrence
Performance Measurement Tool	Krishika, Bahar and Neha

8 Appendix - Interface

This section contains our interface screenshots:

Fig 1. Landing Page:



GAIB: GenAI Testbot

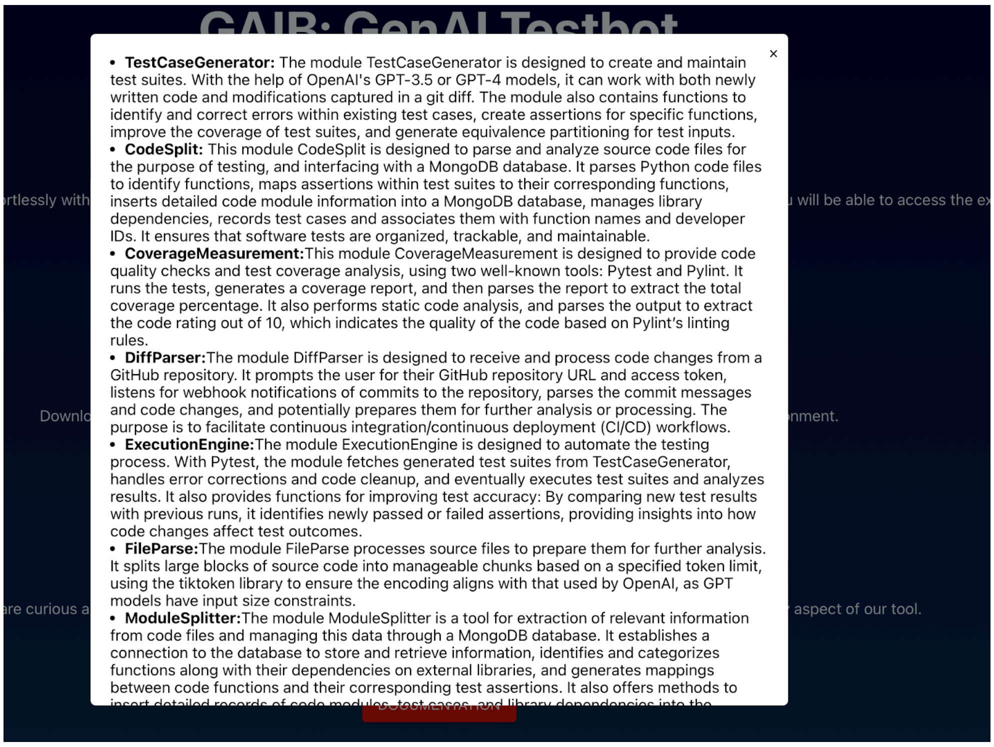
Revolutionize Your Code Testing Process

Generate test cases effortlessly with GenAI Testbot. Sign up or log in to start optimizing your testing workflow. By having an account, you will be able to access the extension and view statistics for your generated test cases.

Download the VSCode extension to seamlessly integrate test case generation into your development environment.

If you are curious about how certain components work, explore our comprehensive documentation to understand every aspect of our tool.

Fig 2. Documentation Popup:



- **TestCaseGenerator:** The module TestCaseGenerator is designed to create and maintain test suites. With the help of OpenAI's GPT-3.5 or GPT-4 models, it can work with both newly written code and modifications captured in a git diff. The module also contains functions to identify and correct errors within existing test cases, create assertions for specific functions, improve the coverage of test suites, and generate equivalence partitioning for test inputs.
- **CodeSplit:** This module CodeSplit is designed to parse and analyze source code files for the purpose of testing, and interfacing with a MongoDB database. It parses Python code files to identify functions, maps assertions within test suites to their corresponding functions, inserts detailed code module information into a MongoDB database, manages library dependencies, records test cases and associates them with function names and developer IDs. It ensures that software tests are organized, trackable, and maintainable.
- **CoverageMeasurement:**This module CoverageMeasurement is designed to provide code quality checks and test coverage analysis, using two well-known tools: Pytest and Pylint. It runs the tests, generates a coverage report, and then parses the report to extract the total coverage percentage. It also performs static code analysis, and parses the output to extract the code rating out of 10, which indicates the quality of the code based on Pylint's linting rules.
- **DiffParser:**The module DiffParser is designed to receive and process code changes from a GitHub repository. It prompts the user for their GitHub repository URL and access token, listens for webhook notifications of commits to the repository, parses the commit messages and code changes, and potentially prepares them for further analysis or processing. The purpose is to facilitate continuous integration/continuous deployment (CI/CD) workflows.
- **ExecutionEngine:**The module ExecutionEngine is designed to automate the testing process. With Pytest, the module fetches generated test suites from TestCaseGenerator, handles error corrections and code cleanup, and eventually executes test suites and analyzes results. It also provides functions for improving test accuracy: By comparing new test results with previous runs, it identifies newly passed or failed assertions, providing insights into how code changes affect test outcomes.
- **FileParse:**The module FileParse processes source files to prepare them for further analysis. It splits large blocks of source code into manageable chunks based on a specified token limit, using the tiktoken library to ensure the encoding aligns with that used by OpenAI, as GPT models have input size constraints.
- **ModuleSplitter:**The module ModuleSplitter is a tool for extraction of relevant information from code files and managing this data through a MongoDB database. It establishes a connection to the database to store and retrieve information, identifies and categorizes functions along with their dependencies on external libraries, and generates mappings between code functions and their corresponding test assertions. It also offers methods to insert detailed records of code modules, test cases, and library dependencies into the

Fig 3. Registration Page:

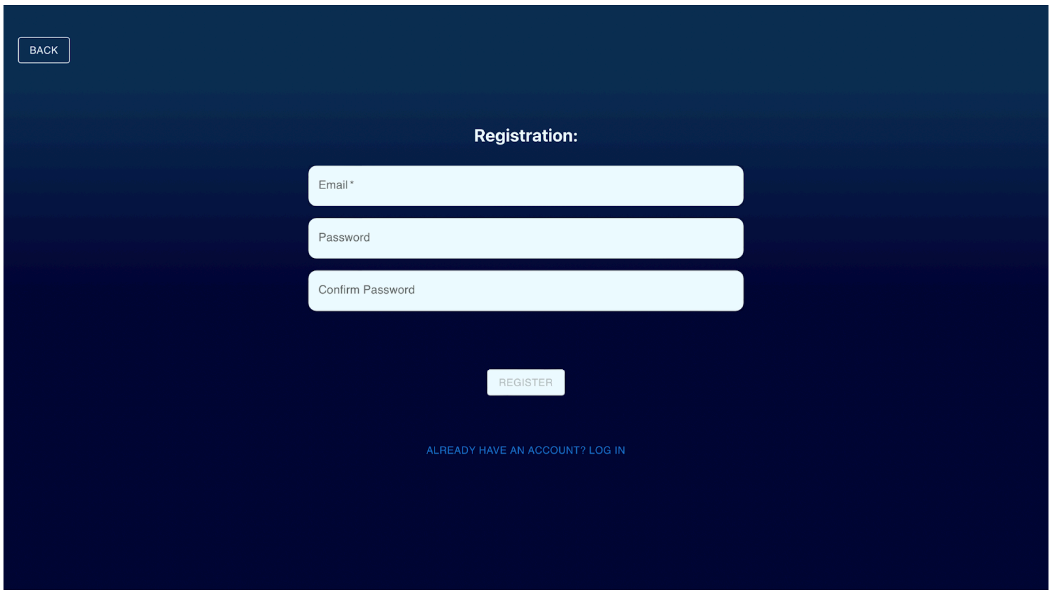


Fig 4. Login Page:

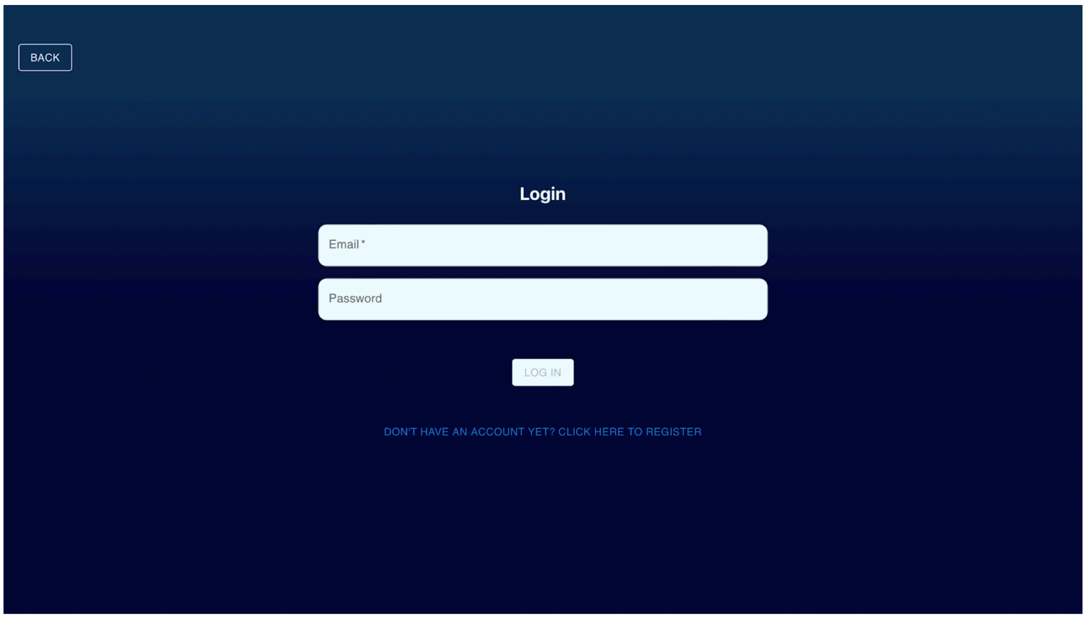
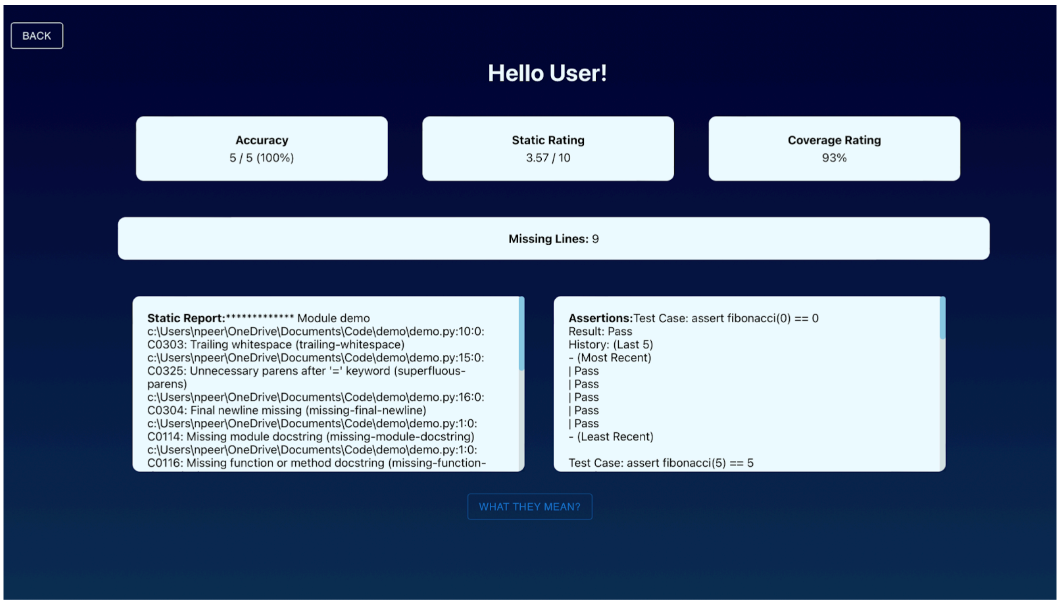


Fig 5. Stats/Main page:



Hello User!

Accuracy
5 / 5 (100%)

Static Rating
3.57 / 10

Coverage Rating
93%

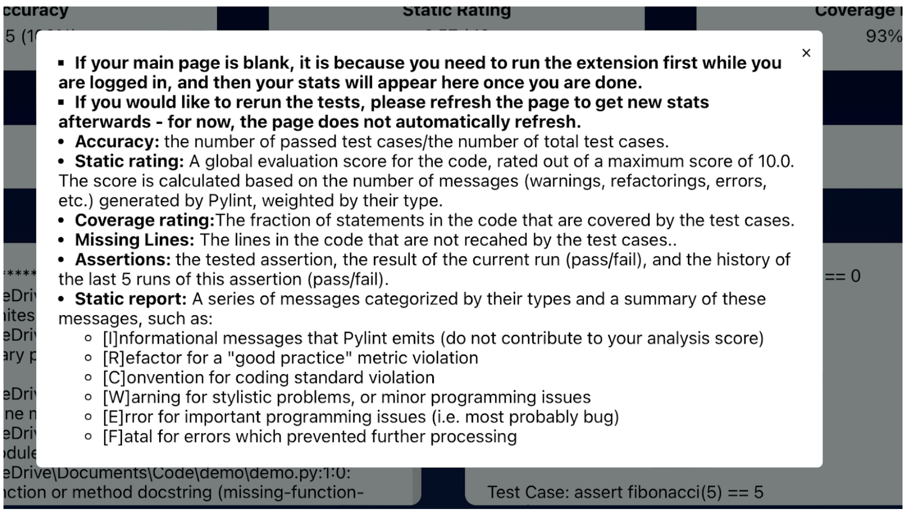
Missing Lines: 9

```
Static Report:************* Module demo
c:\Users\npeer\OneDrive\Documents\Code\demo\demo.py:10:0:
C0303: Trailing whitespace (trailing-whitespace)
c:\Users\npeer\OneDrive\Documents\Code\demo\demo.py:15:0:
C0325: Unnecessary parens after '=' keyword (superfluous-
parens)
c:\Users\npeer\OneDrive\Documents\Code\demo\demo.py:16:0:
C0304: Final newline missing (missing-final-newline)
c:\Users\npeer\OneDrive\Documents\Code\demo\demo.py:1:0:
C0114: Missing module docstring (missing-module-docstring)
c:\Users\npeer\OneDrive\Documents\Code\demo\demo.py:1:0:
C0116: Missing function or method docstring (missing-function-
```

```
Assertions:Test Case: assert fibonacci(0) == 0
Result: Pass
History: (Last 5)
- (Most Recent)
| Pass
| Pass
| Pass
| Pass
| Pass
- (Least Recent)

Test Case: assert fibonacci(5) == 5
```

Fig 6. 'What They Mean' Popup:



- **If your main page is blank, it is because you need to run the extension first while you are logged in, and then your stats will appear here once you are done.**
- **If you would like to rerun the tests, please refresh the page to get new stats afterwards - for now, the page does not automatically refresh.**
- **Accuracy:** the number of passed test cases/the number of total test cases.
- **Static rating:** A global evaluation score for the code, rated out of a maximum score of 10.0. The score is calculated based on the number of messages (warnings, refactorings, errors, etc.) generated by Pylint, weighted by their type.
- **Coverage rating:**The fraction of statements in the code that are covered by the test cases.
- **Missing Lines:** The lines in the code that are not recahed by the test cases..
- **Assertions:** the tested assertion, the result of the current run (pass/fail), and the history of the last 5 runs of this assertion (pass/fail).
- **Static report:** A series of messages categorized by their types and a summary of these messages, such as:
  - [I]nformational messages that Pylint emits (do not contribute to your analysis score)
  - [R]efactor for a "good practice" metric violation
  - [C]onvention for coding standard violation
  - [W]arning for stylistic problems, or minor programming issues
  - [E]rror for important programming issues (i.e. most probably bug)
  - [F]atal for errors which prevented further processing