Verification and Validation for GenAI Test Bot

Version 1

Group 2

Names:	Emails:
Bahar Abbasi Delvand	abbasidb@mcmaster.ca
Krishika Mirpuri	mirpurik@mcmaster.ca
Neha Asthana	ashtann@mcmaster.ca
Nathan Peereboom	peerebon@mcmaster.ca
Lawrence Gu	guq6@mcmaster.ca

April 1st 2024

Version #	Description	Date
0	Original verification and validation document	2024/02/16
1	Final version	2024/04/01

1 Project Description

The GenAI Test Bot, powered by advanced language models like GPT-4, revolutionizes software testing and documentation by automating unit test case generation. It analyzes code changes, generates corresponding tests, and orchestrates tasks via a web server, while a VSCode extension seamlessly integrates with developers' IDEs for real-time communication. This innovation aims to boost code quality, speed up development, and enhance software reliability by automating test creation. Moreover, its test validation feature, employing coverage and static code analysis, ensures accuracy and adherence to coding standards, bolstering the effectiveness of generated test cases.

2 Component test plans

2.1 Core test generation module

Test #1 : To verify that the core test generation module accepts the input differential of a code commit correctly.

- **Type**: Automatic
- **Initial State**: No previous test cases generated for the specific commit.
- **Input/Condition**: A sample differential of a code commit.
- **Output/Result**: The core test generation module successfully receives and processes the input differential.
- **How the test will be performed**: A mock differential of a code commit will be provided as input to the core test generation module. The module's behavior will be observed to ensure that it accepts the input differential without errors.

Test #2: To verify that the core test generation module generates unit test cases based on the provided code commit differential.

- **Type**: Automatic
- **Initial State**: No previous test cases generated for the specific commit.
- **Input/Condition**: A sample differential of a code commit.
- **Output/Result**: The core test generation module generates valid unit test cases corresponding to the changes introduced in the commit.
- **How the test will be performed**: A mock differential of a code commit will be provided as input to the core test generation module. The generated unit test cases will be analyzed to ensure that they cover the relevant code changes and follow the expected format for unit tests.

Test #3: To verify that the generated unit test cases are correct and accurately capture the behavior of the modified code.

- **Type**: Automatic
- **Initial State**: No previous test cases generated for the specific commit.
- **Input/Condition**: A sample differential of a code commit.
- **Output/Result**: The generated unit test cases demonstrate correct behavior and accurately test the modified code.
- **How the test will be performed**: The generated unit test cases will be executed against the modified code to validate their effectiveness in identifying potential issues or regressions. The output of the unit test execution will be analyzed to ensure that it aligns with the expected behavior of the modified code.

Test #4: To verify that the extension preserves the original formatting and comments of the user's Python code.

- **Type:** Automated testing
- **Initial State:** No previous test cases generated for the specific commit.
- **Input/Condition:** A sample differential of a code commit.
- **Output/Result:** The generated test cases will be delivered while preserving the original formatting and comments of the user's code.
- **How will the test be performed:** Provide Python code with diverse formatting styles, which includes comments to the extension. Verify that the generated test cases keep the original formatting intact while the code's readability is retained.

2.2 Web Server

Test #1: To verify that the user's local repository is correctly cloned with the latest commit

- **Type:** Automatic
- **Initial State:** The server is operational and the user's repository information is stored in the database.
- **Input/Condition:** The user requests a commit to their repository, initiating the request.
- **Output/Result:** Local user repository is successfully cloned, and the latest commits are pulled and updated
- **How will the test be performed:** An HTTP request will be sent to the init endpoint, which will then verify that the repository is cloned successfully. Then the cloned repository is checked to ensure the latest commits are pulled and updated on it. We end by checking the status response from the server to ensure successful initialization.

Test #2: Verify that sync endpoint can synchronize with the remote copy of the codebase

- **Type:** Automatic
- **Initial State:** Server is operations, and the local user repository is initialized and updated with the latest commit.
- **Input/Condition:** New commit occurs, and the diff parser requests to synchronize the local repository with the remote copy.
- **Output/Result:** Local repository is synced with the remote copy, so we can maintain that the local copy is at most one commit behind the user's copy.
- **How will the test be performed:** We will simulate a local commit by editing the user's repository, which should trigger the post-commit hook to send a request to the sync endpoint. Then we verify that the server receives the latest commit diff from the diff parser, and check that the local repository is synchronized with the remote copy.

Test #3: Ensure generate-test endpoints initiates test generation flow successfully

- **Type:** Automatic
- **Initial State:** Server is operational and the user has submitted a commit for test generation.
- **Input/Condition:** User requests to initiate the test generation flow, by confirming the startup WebSocket message.
- **Output/Result:** Test generation process started.
- **How will the test be performed:** First we send an HTTP request to the generate-test endpoint and confirm that the server received the request to start the test generation process. Confirm that the test generation module runs to create test assertions for the submitted commit. Then we wait for the chat endpoint to affirm the test assertions with the user.

Test #4: Verify that the chat endpoint can push a WebSocket message to the client and receive user responses for test assertions.

- **Type:** Manual (We will need to ensure that the Core Test Generation module is actually running and creating the tests according to the assertions given so it would be best checked manually)
- **Initial State:** Server is operational and test generation process is initiated.
- **Input/Condition:** WebSocket message sent to client to begin test case generation
- **Output/Result:** User is able to confirm or add test assertions and receives generated test cases.
- **How will the test be performed:** By simulating a WebSocket connection between the server and the client, the server must send a message to the client asking if they wish to begin test case generation. Server will receive confirmation and begin test case generation flow. Core test generation module will create test assertions which the WebSocket will send in a message to the user. Users should be able to edit and confirm the test assertions, and the WebSocket must send the newly edited assertions back to the Core test generation module for test case generation. Finally submit the generated test cases to the client.

2.3 VSCode Extension Client

Test #1: To verify that the extension can push and pull the code directly from the user's GitHub repo.
- **Type:** Automated testing
- **Initial State:** No previous push and pulls have been executed by the extension.
- **Input/Condition:** The user's GitHub repository being linked to the user's VSCode.
- **Output/Result:** The user can clone repositories directly onto their VSCode, or pull from their own previous repositories, and the extension can successfully generate test cases for them and push the tested code right back.
- **How will the test be performed:** The user will try to link their GitHub account first. Then they will attempt to pull or clone a repository, and run the extension to generate test cases for them. After they have their code tested, they will try to push the code back to GitHub.

Test #2: To verify that the extension can be downloaded and successfully run on the user's VSCode.
- **Type:** Manual testing
- **Initial State:** No previous use of this extension on the user's VSCode.
- **Input/Condition:** The user is using the most recent version of VSCode.
- **Output/Result:** The user can download the extension from either the website or from the Extensions tab of VSCode, and the extension successfully installs and runs on their application.
- **How will the test be performed:** The user will try to link their GitHub account first. Then they will attempt to pull or clone a repository, and run the extension to generate test cases for them. After they have their code tested, they will try to push the code back to GitHub.

Test #3: To verify that the extension provides intuitive UI elements for accessing its functionalities.
- **Type:** Manual testing
- **Initial State:** User has the extension installed on their VSCode.
- **Input/Condition:** User interacts with the extension's UI elements within VSCode.
- **Output/Result:** The extension offers clear and easy-to-use UI elements that allow users to easily use it.
- **How will the test be performed:** Explore the extension's interface elements within VSCode. Evaluate their clarity, intuitiveness, and effectiveness in facilitating interaction with the extension's functionalities.

Test #4: To verify that the extension gracefully handles errors and exceptions encountered during its operation.

- **Type:** Automated testing
- **Initial State:** User has the extension installed on their VSCode.
- **Input/Condition:** Trigger various error scenarios while using the extension.
- **Output/Result:** The extension responds to errors and exceptions gracefully, providing informative error messages and maintaining application stability.
- **How will the test be performed:** Introduce different error scenarios, such as network issues or VSCode unexpectedly shutting down, during the extension's operation. Verify that it handles these situations appropriately, ensuring robustness and resilience in its behavior.

2.4 Test Validation Module

Test #1: To verify that the module can detect when it is given an incorrectly formatted test suite and can correct it.

- **Type:** Dynamic
- **Initial State:** Test Validation Module has not yet been run for the specific commit.
- **Input/Condition:** A premade test suite that is not correctly formatted to be compatible with the Execution Engine.
- **Output/Result:** An altered version of the test suite that is correctly formatted to be compatible with the Execution Engine.
- **How will the test be performed:** Various incorrectly formatted test suites (based on formatting errors found to be made by the AI) will be given to the Test Validation Module, and the altered test suite will be fed into the Execution Engine to confirm that it can be run.

Test #2: To verify that the test accuracy tracker properly tracks each test, even when tests or the source code is modified.

- **Type:** Dynamic
- **Initial State:** Test Validation Module has been run at least once on a previous commit.
- **Input/Condition:** Source code/assertions are modified by the user before another commit is made.
- **Output/Result:** All test cases run through the module are the same as (or are altered versions that line up with) the test cases from the previous time the module ran.
- **How will the test be performed:** The system will be run once, and the test cases and their mapping to the functions in the source code will be recorded. Changes will be made to the source code and the assertions will be adjusted, then another commit will be made, running the Test Validation Module again. The test cases and their mapping to the source functions will be compared with the previously recorded results.

Test #3: To verify that the module will detect and fix a test suite that does not meet the expected coverage metrics.

- **Type:** Dynamic
- **Initial State:** Test Validation Module has not yet been run for the specific commit.
- **Input/Condition:** A premade test suite that does not meet our desired 90% coverage of the source code in its tests.
- **Output/Condition:** An altered version of the test suite that does meet the required code coverage.
- **How will the test be performed:** Test suites of varying code coverage percentages will be given to the Test Validation Module. Test suites with at least 90% coverage will be unaltered, and all others will be altered, resulting in a suite with at least 90% coverage.
- **(Apr 1 update)** We were not able to complete this test. Please refer to the “One Page Reflection” document for further information.

3 Performance Tests and Metrics

3.1 Web Server

Test #1: WebSocket Handshake Latency Measurement

- **Type:** Dynamic
- **Initial State:** Server is operational and the user's repository information is stored in the database.
- **Input/Condition:** User submits a commit.
- **Output/Result:** Measurement of WebSocket Handshake Latency, see how long it takes on average and check if it is less than 100 milliseconds.
- **How will the test be performed:** WebSocket connection process begins by initiating a connection request from the client to the server. Start a timer to record the time when the connection request is sent from the client. Stop the timer when the client gets the WebSocket handshake response. Calculate latency by subtracting the start time from the stop time. Check if the value we get is less than, greater than or equal to 100 milliseconds.If the measured latency is more than 100 milliseconds we mark the test as failed.

Test #2: CPU utilization test

- **Type:** Dynamic
- **Initial State:** Server is operational, user's repository information is stored in the database.
- **Input/Condition:** User requests to initialize the repository with the latest commits.
- **Output/Result:** The local user repository is successfully cloned and the latest commits are pulled and updated, while maintaining acceptable CPU utilization levels.
- **How will the test be performed:** To maintain acceptable CPU utilization levels using the init endpoint, we'll utilize the 'sar' command-line utility. Sar collects system activity data at regular 10-minute intervals. We'll run 'sar -u' from the start time of client connection request to 5 minutes later to monitor CPU utilization. Sar records CPU time spent in user-level processes, kernel-level processes, and waiting for I/O operations or idle. Once we establish a baseline, we'll stress test with various load conditions or server configurations.

Test #3: WebServer Security Testing

- **Type:** Dynamic
- **Initial State:** Server is operational
- **Input/Condition:** OWASP ZAP (Zed Attack Proxy) installed on one of our computers.
- **Output/Result:** A comprehensive report from OWASP ZAP (Zed Attack Proxy) on our server's security vulnerabilities.
- **How will the test be performed:** One of our members will have OWASP ZAP (Zed Attack Proxy) installed on their computer to test the web server's security. OWASP ZAP is a web application security scanner that can be run locally from a computer to operate as a proxy server intercepting and checking the HTTP and HTTPS traffic between the member's browser and the web server. Based on OWASP ZAP's findings of our web server security we can plug security holes or manage the way we handle important data better.

Test #4: WebServer Load Testing

- **Type:** Dynamic
- **Initial State:** Server is operational
- **Input/Condition:** Siege installed on one of our computers, and a functional command terminal.
- **Output/Result:** Siege report with the amount of resources used by the server, error rate, concurrency level, response times per task and transaction rates.

- **How will the test be performed:** One team member will use Siege to test the web server's capacity to handle diverse loads. Siege simulates concurrent user access by sending HTTP requests to the server at a set rate or concurrency level, treating each request as an individual user. It measures performance metrics like response time, transaction rate, error rate, and concurrency level. Our goal is to verify the server's capability to manage heavy loads and numerous users simultaneously, given its role in storing and accessing substantial data volumes.

3.2 VSCode Extension Client

Test #1: To verify that the extension efficiently manages the computational resources while generating test cases.

- **Type:** Performance testing
- **Initial State:** Extension is installed on the user's VSCode.
- **Input/Condition:** User triggers generation of test cases for code of varying complexity.
- **Output/Result:** The extension utilizes system resources efficiently without causing excessive CPU or memory usage.
- **How will the test be performed:** Monitor CPU and memory usage of the VSCode process while the extension is generating test cases for Python code. By doing that, we will ensure that resource consumption remains within acceptable limits and does not significantly impact the overall performance of the system.

Test #2: Extension startup time.

- **Type:** Performance testing
- **Initial State:** Extension is installed on the user's VSCode, but has not yet been activated.
- **Input/Condition:** User activates the generation by reloading VSCode.
- **Output/Result:** The extension initializes promptly without causing noticeable delays in the startup time of VSCode.
- **How will the test be performed:** Measure the time taken for VSCode to start up with and without the extension activated. Ensure that activating the extension does not significantly increase the startup time of VSCode, providing a seamless user experience.

4.3 Test Validation Module

Test #1: Converge measurement of generated test cases

- **Type:** Coverage testing
- **Initial State:** The test generation module has generated a suite of test cases for the source code commit differential.
- **Input/Condition:** The generated tests are passed on to the test validation module under internal logic of the application.
- **Output/Result:** A detailed report indicating both the branch coverage and the statement coverage of the application code by the GPT-generated test cases.
- **How will the test be performed:** Running coverage.py on the generated test cases, using the default option to get statement coverage and the “--branch” option to get branch coverage. Then, if coverage is deemed insufficient, the test cases will be sent back to the core test generation module for further improvement. The goal is to reach both 90% branch coverage and 90% statement coverage. This ensures that the test suite generated by our application is genuinely helpful to users.
- **(Apr 1 update)** We were only able to partially complete this test, without the guarantee of 90% coverage. Please refer to the “One Page Reflection” document for further information.