Software Requirements Specification for Gen AI Test Bot

Version 0

Group 2

Neha Asthana - Project Manager
Lawrence Gu - Software Developer
Nathan Peereboom - Software Developer
Bahar Abbasi Delvand - UI/UX designer
Krishika Mirpuri - Software Architect

October 12, 2023

Table of contents and Contributions

Name	Contribution
Neha Asthana	Formatting, List of abbreviations and notations, naming conventions and definitions,Table of contents and contributions, The Purpose of the Project
Krishika Mirpuri	Functional Requirements, Risks and Issues, formatting
Bahar Abbasi Delvand	Identifying the clients and stakeholders, determining the project constraints, formatting
Lawrence Gu	Data and Metrics, non-functional requirements, formatting
Nathan Peereboom	Non-functional requirements, Client/Stakeholder information, document formatting

CONTENTS

1 The Purpose of the Project
- 1.1 The User Background/ Background of project effort
- 1.2 Goals of the project

2 Naming Conventions and Terminology
- 2.1 Abbreviations
- 2.2 Terminology

3 Clients and Stakeholders
- 3.1 Client
- 3.2 Stakeholders

4 Project Constraints

5 Functional Diagram

6 Functional Requirements
- 6.1 User Roles and Permissions
- 6.2 System Overview
- 6.3 Accuracy and Reliability
- 6.4 Programming Language Support
- 6.5 Testing Methods
- 6.6 Error Handling
- 6.7 Documentation
- 6.8 Integration

7 Data and Metrics
- 7.1 Training Data
- 7.2 Dataset
- 7.3 Performance metrics

8 Non-functional requirements
- 8.1 Look and Feel Requirements
- 8.2 Usability and Humanity Requirements
- 8.3 Performance and speed requirements

9 Risks
- 9.1 Technical Challenges
- 9.2 Performance Issues / Cost bottleneck
- 9.3 Security Risks

1 The Purpose of the Project

1.1 The User Background/ Background of project effort

The genAI test BOT is part of an ambitious project undertaken by a pre-revenue company who is looking to revolutionize digital education with a generative AI smart textbook. This project aims to address challenges in software development which are encountered while creating test suites and code documentation.

The project's background is rooted in the desire to provide the developers and other stakeholders with an automated solution. As a result, project managers will enjoy improved code quality, faster development cycles, and streamlined project management. Furthermore, quality assurance teams will have access to comprehensive, automatically generated test suites, simplifying testing efforts. Lastly, end users will benefit from higher-quality software with fewer defects, ensuring a more reliable user experience.

1.2 Goals of the project

The primary goals of the Gen AI Test Bot are:

Automation: The aim is to create a fully integrated bot which will automate the process of generating accurate test suites and code documentation for code which is submitted through a Git push hook. This will reduce the need for manual work which will make it more efficient.

Accuracy: Ensure that the test suites and the documentation created have an accuracy rate which is > 90%. This will help in maintaining the quality of the code and provide reliability.

Integration: The bot will be integrated into the CI/CD(Continuous Integration/Continuous Flow) Pipeline and Git workflows. This will increase efficiency and will facilitate a smooth development process.

Feedback Mechanism: Implementation of a feedback mechanism that will allow for the evaluation and improvement of the generated test suites.

2 Naming Conventions and Terminology

2.1 Abbreviations

- **AI** - Artificial Intelligence
- **API -** Application Programming Interface
- **LLM** - Large Language Model
- **AST** - Abstract Syntax Tree
- **DOM** - Document Object Model
- **ASCII** - American Standard Code for Information Interchange
- **UI/UX** - User Interface / User Experience
- **NL** - Natural Language

2.2 Terminology

- **Differential Parsing** - a technique used to efficiently modify or update a parsed data structure when only a small portion of data has changed. The approach works by initially parsing the entire input and representing it in an AST, a DOM, or another data structure. From that point onwards, any changes to the data will be identified by the parts of the data structure that have been modified.

- **File Parsing** - the process of analyzing and interpreting the contents of a file to extract meaningful information. The process works by taking in a file, reading it using a software tool, breaking down the file into smaller units called tokens and lexically analyzing the tokens to understand their type and structure. The information is then parsed and extracted for use in the system.

- **Pydantic** - Pydantic, humorously described as 'pedantic + Python,' embodies a meticulous and exacting approach to data handling in Python, emphasizing precision and rigorous data validation, which aligns with the essence of 'pedantic,' while providing a powerful tool for ensuring data accuracy and reliability in Python applications.

- **Compile Rate** - The use of the phrase compile rate in our document refers to the number of test cases generated by our bot that successfully compile (without human intervention), out of the total number of generated test cases.

- **Natural Language Descriptions** - These are textual explanations or interpretations of concepts, data or information written in a human-readable language.

3 Clients and Stakeholders

3.1 Client

Our client is John Peng, currently working to create a generative AI based smart textbook.

3.2 Stakeholders

Our primary stakeholder is John Peng (Kongyiji), as this system will be used by the developers to help automate testing and debugging for the code written for the AI smart textbook. Other stakeholders include the wider open-source community and developers in general, as this system will be made more widely available for their use.

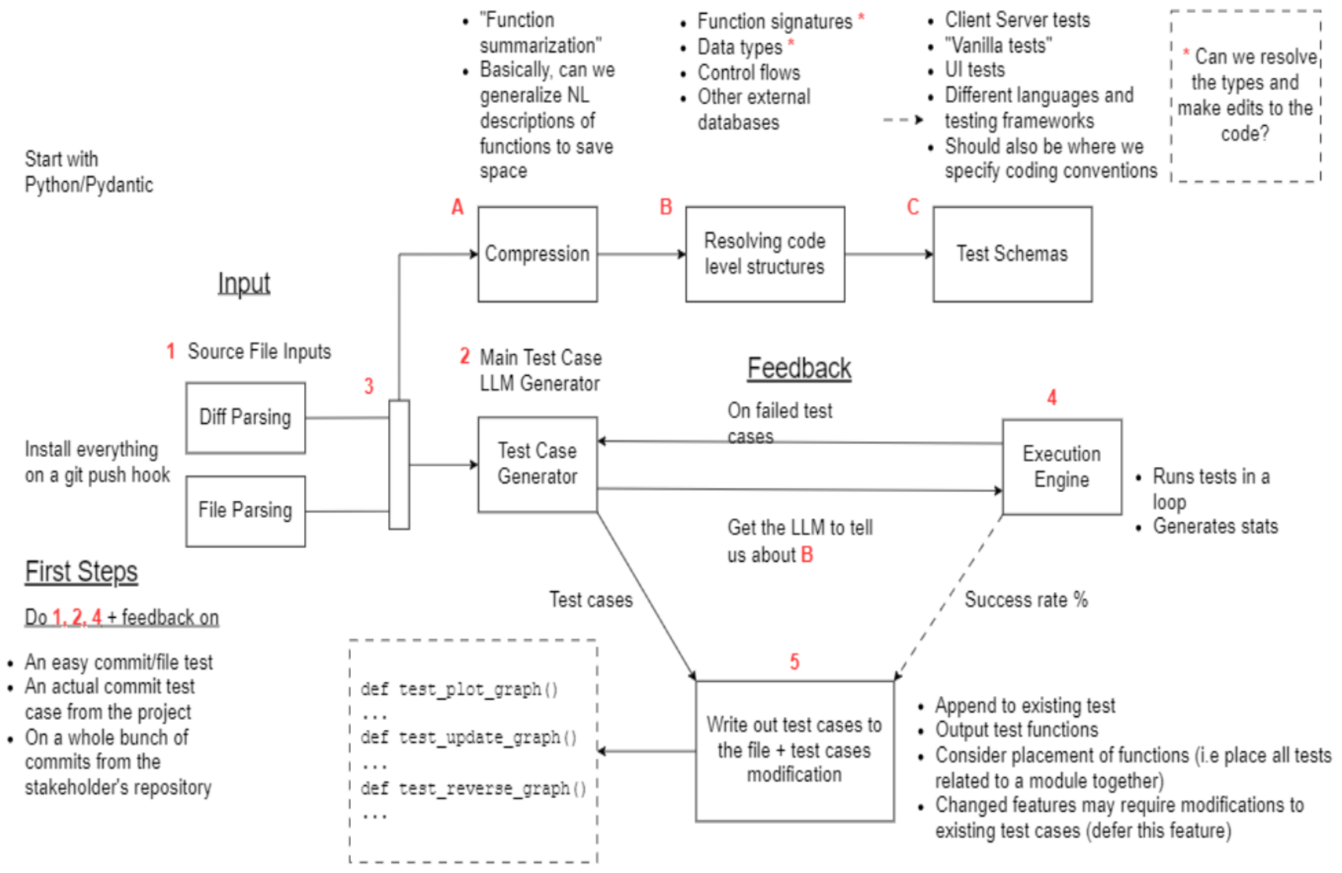
4 Project Constraints

- The project is designed to leverage the capabilities of GitHub, not only for version control, collaboration, and code management, but also to facilitate and encourage active user participation, wherein users will commit their code changes to GitHub as an integral aspect of the project's development and maintenance process.
- The utilization of both OpenAI's advanced GPT-4 and well-established GPT-3.5 models is planned as an integral part of the project's implementation, showcasing

a synergistic approach that utilizes the capabilities and features of these state-of-the-art natural language processing technologies.

- In the context of this project, it is essential to exclusively employ the Python programming language alongside the Pytest testing framework. The product is expected to generate pydantic-compatible Python source code.
- It is crucial to implement rigorous and robust type checking mechanisms to ensure that users are restricted from inputting arbitrary or unintended data, thereby maintaining the integrity and reliability of the system by preventing the inadvertent introduction of data that does not conform to the specified requirements or data types.
- The product is going to be accessible through web-based platforms.
- Considering that LLM imposes a charge for each token usage, we need to optimize our token requests to minimize costs, all the while remaining vigilant about the potential cost constraints we may encounter.

5 Functional Diagram


• "Function summarization"
• Basically, can we generalize NL descriptions of functions to save space
• Function signatures *
• Data types *
• Control flows
• Other external databases
• Client Server tests
• "Vanilla tests"
• UI tests
• Different languages and testing frameworks
• Should also be where we specify coding conventions
* Can we resolve the types and make edits to the code?
Start with Python/Pydantic
A
Compression
B
Resolving code level structures
C
Test Schemas
Input
1 Source File Inputs
Diff Parsing
File Parsing
3
Install everything on a git push hook
2 Main Test Case LLM Generator
Test Case Generator
Feedback
On failed test cases
4
Execution Engine
• Runs tests in a loop
• Generates stats
Get the LLM to tell us about B
First Steps
Do 1, 2, 4 + feedback on
• An easy commit/file test
• An actual commit test case from the project
• On a whole bunch of commits from the stakeholder's repository
Test cases
Success rate %
5
Write out test cases to the file + test cases modification
• Append to existing test
• Output test functions
• Consider placement of functions (i.e place all tests related to a module together)
• Changed features may require modifications to existing test cases (defer this feature)

```
def test_plot_graph()
...
def test_update_graph()
...
def test_reverse_graph()
...
```

6 Functional Requirements

Priority levels for functional requirements will have one of 4 levels, P0 - P3. P0 is our highest priority, P1 is high priority but not urgent, P2 is important but can be set aside for later, and P3 is the lowest priority.

6.1 User Roles and Permissions

- Due to the uniform approach we are adopting, all users will have the same level of access and permission across all aspects of our codebase.
- Users are not restricted or segmented by sections or components of our code. They have unrestricted access to every part of our codebase.
- Uniform permissions reduce administrative complexity and enable users to work with agility and efficiency.

6.2 System Overview

System Components:

- The system consists of 5 main parts; source processor, test case generator, input formatter, execution engine, output generator.

Source Processor:

Priority Level: P0

- The Source Processor takes input from a new Git commit, and it will use differential parsing to map changes and file parsing to organize the file content.
- The Source Processor will be built as a GitHub push hook.

Input Formatter:

Priority Level: P1

- The input formatter will consist of 3 smaller sections; Compressor, Code Structure Resolver and Test Schema.
- The Compressor will try summarizing the functions, i.e. generating natural language descriptions of the functions that use fewer tokens than trying to put all the code of the functions into the LLM directly, which will lead to reduced cost.
- The Code Structure Resolver will resolve function signatures, data types, control flows, and other external databases if necessary.
- The Test Schema is where the tests to be performed are broken down into categories: client server tests, “vanilla tests”, UI tests, etc. This is also where the testing framework and the coding convention should be specified.
- The three above functions will be connected to an interface that interacts with the Source Processor and the Test Case Generator.

The Interface:

Priority Level: P2

- This interface will take in the user’s input when they perform a Git Commit and feed that input into the Input Formatter which will then compile the data from these sources and format them to reduce tokens, remove redundant data, and resolve necessary code structures to give the Test Case Generator a better view.
- The interface will then pass this formatted data into the Test Case Generator.

Test Case Generator:

Priority Level: P0

- The Test Case Generator will take input provided by the input formatter and will build test cases for code modifications identified by the GitHub push hook.
- Test cases will be built using OpenAI APIs.

Execution Engine:

Priority Level: P0

- The Execution Engine runs tests on the test cases created to determine their compile rate and code coverage. If it does not meet our expectation of 90% accuracy for each, it will be passed back into the Test Case Generator in a feedback loop.
- The loop can be re-run as many times as needed till the test cases pass the 90% success rate bar.

Output Generator:

Priority Level: P1

- Once the run of our bot is complete, the Output Generator will produce a document as output with all the updated test cases written in Pydantic-compatible code, and an ASCII table that documents the test cases submitted and their compile status and success rate.
- The Output Generator will take the placement of functions into consideration. It will try placing all tests related to a module together if possible.

6.3 Accuracy and Reliability

- The bot will have a 90% compile rate per test case generated.

6.4 Programming Language Support

- The project will be coded in Python.
- The project will use OpenAI API’s to access pre-trained LLMs
- The project will output Pydantic-compatible code.

6.5 Testing Methods

- The 90% compile rate we will guarantee comes from 90% of the test cases generated compiling successfully by the end of the process.
- The second 90% guarantee we will ensure is that the test cases generated will cover 90% of the code submitted to the bot.

6.6 Error Handling

- Errors present in the source files will culminate in the generation of test cases that, while compiling successfully, will yield a 'Test Case Failed' error upon execution against the provided codebase.
- This will allow the system to inform the user of errors in their source files, and this will be displayed on the interface.

- Errors with the test cases generated will not compile or pass testing, and the bot will automatically rewrite the test cases and re-run on its own till it achieves the 90% compile rate guarantee.

6.7 Integration

- The bot will use a GitHub push hook to make sure that it runs whenever a new commit is pushed to the repository.
- The bot will need to be integrated into the AI Textbook that our client John Peng is building.
- It will be used to generate test cases for the textbook as he designs it.

7 Data and Metrics

7.1 Training Data

- We will not train the LLM from scratch, rather we would utilize pre-trained models from OpenAI, leveraging their extensive training data and sophisticated model architectures.

7.2 Dataset

- For the practical application of our project, John Peng will provide access to one of his pre-established code repositories. This will allow us to evaluate the bot in real-world coding environments, thereby enhancing the reliability of our application.

7.3 Performance metrics

- **Compile rate.** This will be determined by the ratio of test cases that are successfully executed without any discrepancies in a Python interpreter to the total number of test cases administered. This metric provides a clear indication of the reliability of the code generated.
- **Code coverage.** This will be assessed through two distinct methodologies. First, we will employ static analysis tools to trace every possible execution path within the code, such as "if" conditional statements. Additionally, the LLM itself will be used to evaluate the thoroughness of the code's execution paths. Our aspiration is to achieve 90% code coverage. This means that, post a singular execution (which may include running through the feedback loop between the Test Case Generator and the Execution Engine several times), 90% of all potential execution trajectories have been effectively covered.
- **Average run time.** We recognize the importance of efficiency. This will be gauged based on the meantime of 10 consecutive runs. This approach accounts for the inherent variability in the feedback loop duration between the Test Case Generator and the Execution Engine.
- **Financial cost.** This is directly proportional to the number of tokens processed. Monitoring this will ensure that while aiming for optimal performance, we remain within viable economic boundaries.

8 Non-functional requirements

8.1 Look and Feel Requirements

- The system’s functionality will be accessed from a webpage, allowing the user to upload code and download the bot’s report as well as the updated code.
- Because there is little the user needs to do while interacting with the bot, the webpage will have a minimalist design that is easily comprehensible. Upon loading the page, the user will be presented with a clear and concise explanation of the bot and a button allowing them to upload their code.
- After the bot performs its work on the uploaded code, they will be presented with three more buttons: 1 to download the report, 1 to download the updated code, and 1 to commit the code to github.
- In the report generated by the bot crucial metrics, including the efficacy of the generated tests, shall be illustrated using a table constructed with ASCII characters. This presentation method ensures that users receive meaningful, immediate, and easily interpretable feedback on the software's performance metrics.

8.2 Usability and Humanity Requirements

- A 'README' document will accompany the software package. This document will provide users with an overview of the software's capabilities. Furthermore, it will include step-by-step guidelines, ensuring that even users with less technical expertise can utilize the software's full range of functionalities with ease.

8.3 Performance and speed requirements

- We expect that 90% of generated tests, after navigating the iterative feedback mechanism between the Test Case Generator and the Execution Engine without any human intervention, should compile without error.
- For every code commit, we aim for the generated tests to traverse and validate 90% of the execution trajectories, as delineated by our static analysis tools.
- It is anticipated that in 80% of the instances, each commit will circulate through the feedback loop between the Test Case Generator and the Execution Engine no more than 6 times. This ensures that the software remains time-efficient while still delivering high-quality results.

9 Risks

9.1 Technical Challenges

- The LLM might grapple with edge cases, boundary scenarios, and a web of interdependencies between various code sections. Such complex nuances make it considerably challenging for larger code commits to consistently achieve the aspired 90% compilation and success rate.
- As Python is dynamically typed, we will need to explore methodologies to discern function signatures and ascertain data types, potentially by employing graph structures. This poses a novel challenge, given our limited prior experience in this area.

- As with all LLMs, OpenAI models have inherent restrictions related to response length and the volume of information it can process. This poses a risk, especially with large commits, where the bot might inadvertently omit or be incapable of managing the requisite data.
- Although we strive to ensure a coverage of a minimum of 90% of the input code, there remains a risk that pivotal sections of the code might be left out by the bot.

9.2 Performance Issues / Cost bottleneck

- Despite the Input Formatter, there could be scenarios where we may not achieve a significant reduction in the tokens consumed for certain code lines. With OpenAI's API billing model premised on a per-token basis, there is a concern about the long-term economic feasibility of our bot's operations.

9.3 Security Risks

- One of our foundational operational premises involves entrusting OpenAI with both the underlying code of our GenAI Bot and the stakeholder's repository code. We will be leaving all data privacy and security in the hands of OpenAI.
- Sensitive or proprietary code being submitted to the bot may break OpenAI rules, or raise data privacy and security concerns. There have been instances where contents fed to ChatGPT were accidentally leaked due to improper management by the OpenAI team, which are unfortunately out of our control.