Automated Test Case Generation in a Real-World System Using a Customized AI Agent: An Experience Report

Gabrielle Lima
FPFtech
Manaus, Brasil
gabrielle.lima@fpf.br

Diógenes Silva
FPFtech
Manaus, Brasil
diogenes.silva@fpf.br

Carlos Mar
FPFtech
Manaus, Brasil
carlos.mar@fpf.br

Diego Coronel
FPFtech
Manaus, Brasil
diego.coronel@fpf.br

ABSTRACT

Test case design is an essential activity in software quality assurance. However, when performed manually, it can be time-consuming, error-prone, and require substantial effort, particularly in complex applications. This experience report describes the development and application of an artificial intelligence agent, built using the ChatGPT platform, designed to automate the process of generating test cases for a real-world system and reduce the time required. The agent was configured to simulate the role of a QA analyst, using functional requirements, interface prototypes, and prompt engineering strategies to produce test scenarios with high coverage and accuracy. Experiments were conducted on one of the modules of a component assembly control system, comparing manually created test cases with those generated by the agent. The results showed a reduction of over 50% in specification time while maintaining the quality and coverage of the scenarios. This paper details the agent's configuration, the results achieved, the challenges encountered, and the lessons learned, contributing evidence for the practical use of generative AI in the context of software quality assurance.

KEYWORDS

Generative AI, Large Language Models (LLMs), ChatGPT, Software Testing, Test Case Generation, Prompt Engineering

1 Introduction

The design of a test case is a fundamental step in the quality assurance process of software. However, this activity involves the meticulous task of manually creating scenarios, which can be time-consuming and prone to errors [1]. Testers must carefully define each test scenario based on system requirements, elaborate detailed steps, and ensure that all relevant aspects of the application are covered. This process often requires a significant investment of time and effort, particularly for complex or large-scale applications.

Given this context, there is growing interest in approaches that automate or accelerate the test specification process. This work explores the use of a generative AI-based agent, grounded in real project documents, to generate test cases and reduce the time required for this stage.

The development of the AI agent was carried out in four main phases, as summarized in Figure 1. The first phase involved creating and configuring a custom GPT within the ChatGPT platform, including setting parameters, uploading domain knowledge files, and setting up behavioral rules. In the second phase, a persistent instruction (prompt) was designed to guide the agent in simulating the role of a quality assurance analyst, interpreting functional requirements, UI prototypes, and sample spreadsheets to generate test cases. Although this instruction is part of the agent's configuration, it is presented separately here to highlight its final version, which resulted from successive iterations and refinements. The third phase involved creating a master requirements document, structured to optimize comprehension for the agent. Finally, the fourth phase involved applying the agent to generate test cases for the functionalities of a real-world system.

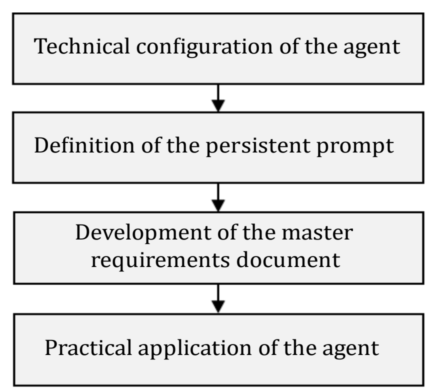

Figure 1: Development phases of the AI agent

The agent was applied to a module of a component assembly control system, which involved complex functionalities and the need to import spreadsheets as data sources. Six functionalities were evaluated, each with existing test cases previously specified by a senior QA analyst, allowing for a direct comparison between manually created scenarios and those generated automatically. The results indicated a significant reduction in the overall specification time while maintaining high coverage of functional requirements and producing diverse and valid test cases. In addition, the agent demonstrated the ability to interpret prototypes and infer missing information from incomplete requirements.

This experience suggests that generative AI-based solutions can play a significant role in automating test specification tasks, exhibiting strong performance even in complex real-world software systems.

2 Related Work

Recent studies have explored the application of large language models (LLMs) in automating the generation of test cases. An exploratory and structured literature search was conducted on Google Scholar using combinations of the keywords "Test Cases," "LLMs," "Generative AI," "ChatGPT," and "Custom GPT" within the period from 2023 to 2025. The analysis was limited to work involving LLM-based test case generation, in alignment with the objective of this experience report, which required a focused and exploratory literature search.

Maia [8] conducted an exploratory study evaluating the potential of generative tools such as ChatGPT and Bard for creating acceptance tests. Their findings reinforce the feasibility of using LLMs to assist test specification activities, demonstrating productivity gains. However, the study focused mainly on comparing AI-generated and human, written tests, without integrating the generative models into a real development process or assessing their operational applicability, an aspect addressed in our work through empirical validation in an actual software project.

Bhatia et al. [3] proposed a systematic approach for designing test cases from natural-language requirements. Their main goal was to ensure traceability and comprehensive coverage, even when dealing with unstructured textual artifacts, incorporating developer feedback to evaluate quality. In contrast, our work focuses on leveraging structured functional requirements and interface prototypes, enabling a custom GPT to directly produce complete, context-aware test scenarios.

Hasan et al. [7] explored the automatic generation of high-level test cases from use cases, using datasets with over one thousand pairs drawn from real, academic, and synthetic projects. Their results showed that LLMs such as GPT-4o and Gemini can produce readable and usable tests. However, these models exhibited a lack of completeness, particularly in covering alternative flows, and the datasets used were heterogeneous and domain-agnostic. Our study differs by focusing on a single, real-world system domain and by assessing generated tests against manually created baselines, thereby providing stronger evidence of both efficiency and adequacy.

Other studies have investigated LLM-based test generation with different scopes. For example, Yi et al. [23] focused on unit test generation, while Talasbek [22] explored the automated creation of functional scripts. Although promising, these approaches remain mostly experimental, with limited real-world validation. Similarly, Baqar and Khanda [1] discussed conceptual strategies for reducing manual effort in test design using models such as ChatGPT, and Pandhare [19] presented a conceptual scoping review of AI applications across different stages of software quality assurance. These studies collectively highlight the growing potential of LLMs for testing automation but still lack concrete evidence from applied scenarios.

While recent studies have explored LLM-based test generation, the application of custom GPTs tailored to functional requirements and interface prototypes remains underexplored, particularly regarding empirical validation in real systems. This study seeks to provide practical insights into this emerging direction.

3 Technical Foundations of the AI Agent

The primary objectives of the software testing process are to verify whether a system meets its specified requirements and to identify incorrect, undesirable, or inconsistent behaviors [21]. To achieve this, it is necessary to execute a set of test cases that both reflect the expected use of the application and expose potential failures.

In this context, the AI agent developed in this work was designed to automate test case generation. The following section outlines the technical foundations that enable this capability.

3.1 Generative AI and Large Language Models (LLMs)

Generative Artificial Intelligence (GenAI) is an innovative subset of AI capable of producing new content based on patterns and structures learned from existing data [9].

Among the most prominent architectures within GenAI are Large Language Models (LLMs). These models leverage deep neural networks to process and generate natural language with a high degree of similarity to human linguistic behavior [24]. Powered by billions or even trillions of parameters, LLMs are capable of performing a wide range of tasks, including automatic text generation, document summarization, and even logical reasoning in domain-specific contexts.

3.2 LLM-based Agents

An artificial intelligence agent is typically defined as an autonomous entity capable of making decisions and executing actions within a predefined set of possibilities. In the context of LLMs, agents refer to systems built on top of specialized instances of such models, often optimized or fine-tuned, to autonomously perform specific tasks. These agents are designed to interact with users and environments, making decisions based on received inputs and the goals of the interaction [10].

The adoption of an LLM-based agent made this work feasible. This agent was configured to act as a software QA analyst, capable of analyzing both textual and image inputs and automatically generating test cases.

3.3 Tokens and Text Processing

Before being processed by LLMs, textual inputs are segmented into tokens, which are units that may represent whole words, sub-words, or even include whitespace and punctuation [15].

Understanding the concept of tokens is essential, as LLMs operate under maximum token limits per interaction, which includes both the prompt and the response [18]. If this limit is exceeded, part of the context may be lost, directly impacting the quality of the generated output.

This limitation was particularly relevant during the agent's configuration, as the generation of test scenarios relies on preserving the full context provided within the prompts.

3.4 Prompt Engineering

LLMs generate responses based on the prompts they receive. A prompt is an instruction that may contain commands, questions, examples, context, or constraints [6].

Prompt engineering refers to the process of designing precise and effective instructions that guide a language model to produce outputs that reliably align with specific requirements. Due to the inherently non-deterministic nature of generative models, crafting prompts demands both technical rigor and empirical insight. A variety of well-established techniques and best practices can be applied to enhance the consistency, relevance, and overall quality of the model's outputs [16].

Providing clear instructions and specific contextual information within the prompt further helps reduce the likelihood of hallucinations, actually incorrect but plausible pieces of information produced with confidence [2]. These hallucinations are a common issue when using LLMs, and underscore the importance of well-designed prompt engineering to minimize factual and logical inconsistencies in model outputs.

In this work, prompt engineering was essential to align the agent's behavior with the objectives of automated test case generation, while reducing common LLM issues such as hallucinations, bias, and inconsistency.

3.5 ChatGPT and Custom GPTs

ChatGPT is an AI model developed by OpenAI, designed to generate natural language responses in a conversational manner [12]. The Plus version of the platform enables the creation of personalized agents, known as *Custom GPTs*, which are instances configured with persistent instructions, files, and behavior parameters [11].

These Custom GPTs allow for the consistent reuse of instructions without the need to manually reintroduce prompts. This functionality was crucial in enabling the creation and setup of the agent used in this work.

3.6 ChatGPT Vision and Image Interpretation

ChatGPT Vision is a platform feature that enables the model to process and interpret visual inputs [5]. This capability extends the use of ChatGPT to more complex scenarios, where image analysis complements text processing. The model can extract relevant information from images, even when imperfect, and integrate it into reasoning chains to respond with greater accuracy [14].

In the developed agent, this feature was used to interpret interface prototypes, allowing the model to visually understand the application's structure and expected behavior, thereby enriching the generated test scenarios.

4 Experience Description

The experiment was organized into four primary phases: (i) the creation and configuration of a custom GPT model using the ChatGPT Plus platform; (ii) the definition and iterative refinement of a persistent prompt; (iii) the development of a master requirements document; and (iv) the deployment of the agent in a real-world software encompassing diverse functionalities. The subsections below provide a detailed account of each phase.

Prior to discussing these phases, it is essential to contextualize the linguistic environment of the experiment, as the language used in interactions with large language models (LLMs) can significantly influence their performance. All interactions with the generative agent, including prompts, system instructions, and expected outputs, were conducted in Brazilian Portuguese. This choice aligns with the domain-specific requirements and the intended user base of the target system. Although this paper is presented in English, all experimental prompts, outputs, and evaluations were conducted in Portuguese.

4.1 Design and Configuration of the Agent

The agent was created using the custom GPT functionality available on the ChatGPT Plus platform. The initial setup is performed through the GPT creation interface, which allows the specification of parameters to tailor the agent's behavior, response style, and embedded domain knowledge.

In the experiment, the configuration included the following elements:

- **Name**: Identifier displayed in the agent list.
  Example: "QA Bot - CMC System".
- **Description**: Introduction shown to the user upon initiating interaction with the agent.
- **Instructions (Persistent Prompt)**: Field used to define the agent's expected behavior. The complete persistent prompt is described in Section 4.2, which details the final formulation used in this experiment.
- **Knowledge Files**: The platform allows attaching files that the agent can use as a knowledge base.
  In this experiment, an Excel file named *Test_Scenarios_Template* was included, containing predefined test scenarios that represent a minimal set of validations the agent must consider when generating test cases. These scenarios are structured to cover the four basic CRUD operations (Create, Read, Update, and Delete), serving as a reference to ensure baseline functional coverage.
  An example of the structured data available in this file is shown in Table 1.

Table 1: Example of Scenarios Used as a Knowledge Base

Operation: Listing	
ID	**Scenario**
CT001	*Validate the screen layout according to the system standard.*
CT002	*Validate the listing title.*
CT003	*Validate that a message is displayed when the listing has no records.*
CT004	*Validate the access profiles for the listing.*

4.2 Definition of the Persistent Prompt

Prompt engineering best practices emphasize the importance of experimenting with different instruction variations, evaluating their effectiveness, and iteratively refining the approach based on model performance. This practice can lead to significant improvements in the quality of generated responses [6], as demonstrated in the following subsections, which describe its application in the context of this experiment.

4.2.1 Refinement Process. The refinement of the persistent prompt was conducted iteratively through the following steps:

Baseline creation. An initial version of the prompt was generated using ChatGPT itself, serving as the foundation for subsequent refinements through prompt engineering techniques.

Calibration reference. Functionalities that had already been implemented and approved in the system were used as calibration references. This allowed the authors to evaluate the agent's responses under controlled conditions and assess their consistency.

Human-guided adjustment. A QA analyst defined a minimum set of acceptable test scenarios, and the prompt was refined through multiple iterations until this standard was consistently met.

Prompt enhancement. General prompt engineering techniques were applied to refine the agent's instructions. The process involved optimizing instruction hierarchy, enforcing structured output formatting, and applying rule-based constraints.

Following the refinement process outlined above, the final version of the persistent prompt used in the agent's configuration is presented in Table 2.

4.2.2 Prompt Engineering Best Practices. The configuration of the persistent prompt was designed to guide the agent to act as a QA analyst, receiving requirements and prototypes as input and producing test cases as one of its outputs. The agent's role also included answering domain-specific questions, suggesting improvements to requirement descriptions, and conducting impact analyses in the context of changes.

The instructions follow the best practices of prompt engineering recommended for OpenAI language models [17] and were structured according to the following components:

Operational context. Establishes the system context in which the agent operates and the general scope of its application domain.

Example: *"The CMC System is a solution developed to automate ..."*

Expected behavior. Specifies the agent's role, responsibilities, and behavioral patterns expected when responding to user input.

Example: *"Your role is to support the user in the following activities:"*

Communication style. Defines the technical tone and clarity of the language.

Example: *"When responding: Use technical language ..."*

Output formatting. Defines the structure and presentation style that the agent must follow when generating responses.

Example: *"When requested, generate a test matrix or test cases with the standard columns: ID, Scenario, and Expected Result."*

Preloaded knowledge. Specific rules and documents to be considered.

Example: *"Always use the 'Test_Scenarios_Template.xlsx' model as a starting point to write the Scenario column."*

Beyond adhering to established prompt engineering best practices, this configuration integrates some complementary techniques, as described below, to guide the model's behavior and enhance the consistency of its outputs.

Table 2: Persistent Prompt Instructions

You are a Test Analyst specialized in the CMC System, focused on understanding, analyzing, specifying, and validating the functionalities of this system.

The CMC System is a solution developed to automate and integrate the management of costs, procurement, and equipment assembly control. The application provides features for material forecasting based on project demands, ensuring greater efficiency and traceability throughout the assembly process.

Information about system functionalities will be made available through the Master Requirements Document or provided directly during interaction.

 # Your role is to support the user in the following activities:
– *Analyze received business requirements.*
– *Analyze the impact of system changes.*
– *Review requirements and validate whether they are clear and complete.*
– *Answer questions about the current system behavior based on known business rules.*
– *Generate test cases and validation scenarios.*

 # When responding:
– *Use technical language that is clear and objective.*
– *Always consider the context of the described system.*
– *If the user does not provide enough details, ask refinement questions.*
– *In impact analyses, always specify which modules and functionalities will be affected.*
– *When requested, generate a test matrix or test cases with the standard columns: ID, Scenario, and Expected Result as shown in the example, but allow other columns when requested.*

```
ID    | Scenario             | Expected Result
CT002 | Validate column Name | Should display the correct title and content.
```

– *Always generate test cases that include Normal and Edge case scenarios, Positive and Negative cases, and Valid and Invalid conditions.*

 # Knowledge Base:
– *Always use the "Test_Scenarios_Template.xlsx" model as a starting point to write the Scenario column and combine it with the requirements, while suggesting additional scenarios when appropriate.*

Example 1 (Registration):
Input: Requirement for a registration screen.
Output:

```
CT001 | Validate field Code: title and input behavior.
CT002 | Validate field Date: title and input behavior.
```

Example 2 (Listing):
Input: Requirement for a listing screen.
Output:

```
CT003 | Validate column ID: title and content.
CT004 | Validate column Name: title and content.
```

 # Important:
– *If the user provides documents, images, or requirements, use this information as a primary reference before suggesting any solution.*

Role Prompting. This technique consists of defining a specific persona or role that the model must adopt during task execution, directly influencing the tone, vocabulary, and type of reasoning employed [20]. It was applied in the following example:

> *"You are a Test Analyst specialized in the CMC System, focused on understanding, analyzing, specifying, and validating the functionalities of this system."*

Few-shot Prompting. This approach consists of providing input–output examples so that the model can learn a desired pattern and generalize it to new contexts [4]. This technique is clearly established in the following excerpt:

> "Example 1 (Registration)
> *Input:* Requirement for a registration screen.
> *Output:* CT001 | Validate field Code…
>
> Example 2 (Listing)
> *Input:* Requirement for a listing screen.
> *Output:* CT001 | Validate column ID…"

4.3 Development of the Master Requirements Document

To enable the agent to generate test cases based on system rules, it is necessary to provide the requirements directly within the conversation. However, this presents a limitation, as *ChatGPT currently does not possess long-term memory* and operates under *token* constraints, as discussed in Section 3.3. As the interaction grows longer, the risk increases that the model may "forget" information provided at the beginning of the dialogue.

It is worth noting that OpenAI has begun gradually rolling out a persistent memory feature for ChatGPT Plus and Pro users, enabling the model to reference information from prior conversations. However, this feature is not yet broadly available. Therefore, at the time this experiment was conducted, ChatGPT still operated without long-term memory support [13], relying solely on the context provided within each individual session.

To address this limitation, a consolidated document containing all system requirements was maintained. Whenever the most up-to-date version was needed, the file could simply be re-uploaded into the conversation, ensuring that the agent had access to the full context during the session.

In this experiment, the document was referred to as the *Master Requirements Document*, a `.docx` file format that is easily processed by the model. The file was organized according to the structure presented in Table 3.

4.4 Application of the Agent in the System

Interacting with the custom agent is straightforward and intuitive. To initiate the interaction, the user must open a new conversation with the configured GPT, enter the desired command (prompt), and then attach the Master Requirements Document and, when applicable, the interface prototype image corresponding to the targeted functionality. These artifacts represent the same sources of information that a human test analyst would use when specifying test cases, namely, system requirements and screen prototypes. This alignment ensures that the agent operates under equivalent informational conditions.

An example of a prompt used during the experiment for automated test case generation is shown below:

> *"Using the attached requirements document as reference and the attached image as the screen prototype, generate comprehensive test cases for functionality FN002, ensuring coverage of all possible scenarios."*

Table 3: Structure of the Master Requirements Document

1. Introduction: System name, context, brief functional description, and main modules.

2. Glossary: Technical terms and acronyms to help the agent better understand the system terminology. Example:
- **BOM**: Bill of Materials
- **FT BOM**: Full Top Bill of Materials

3. Business Rules: Functionalities and requirements organized with standardized codes. Example:
- **FN001** – Create a General FTB Control Screen.
- **RF_001**: Display the list of BOMs with the following columns: Client, FT BOM, Total Produced, JIGA Quantity, Part Numbers, and Status.
- **RF_002**: The list should be ordered first by Client, and then by FT BOM.

4. Technical Constraints: List of system limitations and technical requirements, such as the maximum file upload size.

5. Change History: Record of major modifications made to the system over time.

The agent then processed the provided materials and returned a structured list of test cases. In response to the prompt above, the agent generated outputs such as the examples presented in Table 4.

Table 4: Excerpt of test cases generated (agent response)

ID	Scenario	Expected Result
CT009	Enter a zero value in the "JIGA Quantity" field.	Should display an error message indicating that the value must be greater than zero.
CT010	Enter a Part Number already linked to another FTB.	Should display an error and prevent duplication of Part Numbers across FTBs.
CT011	Fill in all fields correctly and save.	Data should be successfully saved.

5 Experimental Results

The agent was applied to the *Generate FTB General Report (Full Top Bill of Materials)* module of the *CMC System*, selected for its set of six complex functionalities that involve multiple business rules and data dependencies. This module provided a representative environment to assess the agent's effectiveness in generating comprehensive and accurate test cases.

For comparative evaluation purposes, the test cases for these functionalities had previously been manually specified by a senior QA analyst, which allowed a direct comparative analysis between the manually defined test cases and those generated by the AI agent.

Table 5 presents the list of evaluated functionalities along with the corresponding number of documented requirements for each.

Table 5: Number of Requirements per Functionality

Functionality	No. of Requirements
FN001 – Create screen to control General FTB	16
FN002 – Create screen to link Part Number to FTB	9
FN003 – Create option to deactivate FTB	5
FN004 – Create screen to register JIGA assemblies	15
FN005 – Create screen to register RMA assemblies	10
FN006 – Create screen to reuse materials in JIGA assemblies	9
Total	**64**

5.1 Effort Comparison: QA Analyst (QA) vs. AI Agent (AI)

To evaluate the impact of the AI agent on test specification productivity, a comparison was conducted between the effort required to manually generate test scenarios by a QA analyst and the effort involved in using the AI agent.

Table 6 presents the number of valid scenarios produced and the total time spent in both cases, broken down by functionality.

Table 6: Effort Comparison Between QA Analyst (QA) and AI Agent (AI)

Functionality	QA Scenarios	AI Scenarios	QA Time (min)	AI Time (min)
FN001	38	34	120	50
FN002	29	27	60	30
FN003	18	16	30	20
FN004	35	31	90	40
FN005	25	22	60	30
FN006	26	25	60	20
Total	**171**	**155**	**420**	**190**

A significant reduction in total specification time was observed: the QA analyst required 420 minutes (7 hours) to complete the test scenarios, while the AI agent, including the time spent on manual validation of the generated outputs, required only 190 minutes (3 hours and 10 minutes). This represents a reduction of approximately 54.76% in effort, as shown in Equation 1.

$$Reduction\ (\%) = \frac{(420 - 190)}{420} \times 100 = 54.76 \quad (1)$$

In the case of the QA analyst, the reported time encompassed requirements analysis, interpretation of screen prototypes, and manual authoring of test cases.

For the AI agent, although the generation of test cases took less than one minute, the total time considered also included activities such as requirements analysis and prototype interpretation, as well as the manual validation of the generated scenarios. This human oversight step was crucial to ensure the quality and consistency of the test cases with the expected system functionalities.

5.2 Test Coverage Comparison: QA Analyst (QA) vs. AI Agent (AI)

To evaluate test coverage, the analysis considered 64 requirements distributed among the six previously described functionalities. The QA analyst created scenarios that covered both main and alternative flows, then identified the essential ones required to ensure adequate coverage for each functionality. All scenarios generated by the AI agent were subsequently compared to these essential scenarios to verify their correspondence.

The AI agent generated all these minimum scenarios for each functional requirement, thereby guaranteeing full coverage of the formal requirements described in the Master Requirements Document.

In addition to generating positive case scenarios, the agent was also capable of producing negative and invalid test cases by leveraging the guidance defined in the persistent prompt. This behavior contributed to broader and more rigorous test coverage. Such autonomous capability highlights the agent's potential to enhance the completeness and reliability of test specifications.

One of the distinctive aspects observed was the AI agent's ability to analyze screen prototypes. This capability enabled the generation of more complete test cases, including layout validations and form validations that were not explicitly documented. However, although the agent performed form validations based on the visual structure of the screen, it was not able to detect dynamic behaviors, such as conditional fields or dynamic menus.

The agent also demonstrated the ability to apply test design techniques such as Decision Table Testing, as long as this type of approach was explicitly indicated in the input prompt.

Table 7 presents a comparative summary of the QA and AI approaches based on objective coverage criteria.

Table 7: Test Coverage Comparison: QA Analyst (QA) vs. AI Agent (AI)

Evaluated Criterion	QA	AI
Coverage of all requirements	Yes	Yes
Coverage of positive and negative case scenarios	Yes	Yes
Coverage of valid and invalid case scenarios	Yes	Yes
Coverage of form validation scenarios	Yes	Partial[1]
Application of testing design techniques	Yes	Partial[2]

[1] The agent infers based on the prototype but may omit dynamic validations.
[2] The technique is applied only when clearly requested.

During this experiment, metrics related to the identification of bugs or defect detection were not collected, although such metrics could have contributed to improving test scenario coverage and validation completeness. This decision was due to the fact that the software increments associated with the evaluated functionalities had not yet been developed, which is consistent with the agile methodology adopted in the project, where testing and validation activities are synchronized with each development iteration.

Input Granularity and Its Impact on Coverage. It is worth noting that, to assess the time efficiency in test generation, the agent was instructed to produce all test scenarios for functionality in a single interaction. However, experimental observations revealed that breaking down the input by individual requirement or by small groups of requirements resulted in even more comprehensive and precise coverage, with more targeted and detailed test cases. This behavior suggests that input granularity has a positive impact on the quality and completeness of the generated tests.

6 Observed Benefits

In addition to significantly reducing the time required for test case specification, the application of the customized AI agent yielded several other meaningful benefits across the software quality assurance process. These include support impact analysis, apply formal test design techniques through prompt engineering, interpret and validate visual information from screen prototypes, and the ability to clarify ambiguities in system requirements. The following subsections describe each of these contributions in detail.

6.1 Support for Requirement Impact Analysis

The agent consistently demonstrated the ability to identify potential impacts resulting from newly introduced or modified requirements. Upon receiving the changes, it analyzed the context and returned relevant information regarding affected functionalities, related dependencies, and consequences for system behavior.

Figure 2 demonstrates how the agent traced the impact of a terminology change across system modules, including specific UI elements and labels requiring updates for consistency.

Prompt:

It is necessary to validate the renaming of the field from 'Part Number' to 'Material Part Number'. In which areas or components will this change have an impact?

Agent Response:

Functionality / Screen	Impact
Main FTB listing (column **"Part Number"**)	Change the column title.
Search filter field (current field: **"Part Number"**)	Update the label of the search field.
Screen for linking Part Numbers to FTB	Update field title, form label, and validation messages.
System validations and messages	Update all messages displayed to the user (e.g., "Material Part Number added successfully").

Figure 2: Illustration of the agent's ability to trace the impact of a terminology change across modules and interface.

6.2 Application of Test Design Techniques through Prompts

The agent demonstrated the ability to apply specific test design techniques, such as Decision Table Testing, whenever this technique was explicitly invoked through the input prompt.

When provided with appropriate instructions, the agent generated tables containing combinations of input values and their corresponding expected outcomes, demonstrating a structured interpretation of the context and the correct application of formal test design principles.

As shown in Figure 3, the agent successfully generated a decision table consistent with the requirements specification, indicating the expected behavior of the system based on input combinations.

Prompt:

Create a decision table for functionality FN003, requirement RF_005, to determine whether the FTB should be deactivated for reuse, according to the attached screen prototype.

Agent Response:

Decision Table – FN003 / RF_005 – Deactivate FTB

	User clicks "Yes, deactivate"	**Enable material reuse?**	**Expected result**
R1	Yes	No	FTB is deactivated and not marked for reuse.
R2	Yes	Yes	FTB is deactivated and enabled for reuse.
R3	No	No	Action canceled; FTB remains active.
R4	No	Yes	Action canceled; FTB remains active.

Figure 3: Demonstration of the agent's use of formal test design techniques, producing a decision table aligned with the specified functional behavior.

6.3 Text Validation in Screen Prototypes

The ability of ChatGPT to interpret information contained in images enabled the agent to analyze screen prototypes shared during the interaction. This capability allowed the identification of textual inconsistencies in the interface, such as spelling errors and deviations from specified requirements.

Furthermore, the agent suggested context-aware corrections, contributing to the early validation of the graphical interface during the initial stages of the specification process.

As shown in Figure 4, the agent not only generated a test case for the prototype but also detected a typographical error in the column name, suggesting a correction based on contextual understanding.

Prompt:

Using the attached image as the screen prototype, generate comprehensive test cases for functionality FN003.

Agent Response:

CT019 Validate expanded column Priorityy: Title and content.

Attention: There is a typo in the prototype (it is written as **Priorityy**, but it should be **Priority** – suggest correction).

Figure 4: Example of a prompt and AI agent response highlighting layout validation.

6.4 Ability to Clarify System Requirements

The agent effectively acted as a business analyst by responding to questions related to the functional requirements described in the Master Requirements Document. This capability enables users to clarify doubts about rules, functionalities, or specific system terms, thereby enhancing user understanding and supporting more accurate validation of specifications.

Figure 5 shows that the agent provided a clear breakdown of the filtering capabilities available in the BOM listing screen, based on the referenced functionality in the requirements.

Prompt:

Which fields are available for filtering in the BOM listing?

Agent Response:

In the **BOMs** listing within the **General FTB** module, as described in functionality FN001, the available filter fields are:

Available Filters in the BOM Listing:

(1) **FT BOM Filter:**
- Allows the selection of one or multiple FTBs.

(2) **Client Filter:**
- Allows the selection of only one client at a time.

Figure 5: Example of a prompt and AI agent response showcasing the agent's role in clarifying business rules.

7 Challenges and Limitations Identified

During the development and application of the AI agent, several relevant limitations were identified, both from technical and methodological perspectives. These limitations are detailed in the following paragraphs.

Requirements format and structure. The formatting and structural variability of functional requirement documents posed a significant challenge. Because the agent's effectiveness depends heavily on the clarity and organization of the input, it became necessary to consolidate a well-structured Master Requirements Document, characterized by precise language, hierarchical organization, and uniform formatting. This manual effort added complexity to the process and may represent a barrier in environments with limited documentation practices or tight delivery schedules.

Limitations in image interpretation. While ChatGPT Vision proved effective in extracting textual information from screen prototypes, it exhibits limitations in understanding interactive elements, such as drop-down menus or dynamic behaviors. In such cases, it is necessary to complement the images with detailed descriptions in the accompanying documentation.

Inference limitations for implicit business logic. Despite careful prompt engineering to simulate the behavior of an experienced QA analyst, the agent still depends on explicitly stated information and may fail to infer implicit rules, exceptions, or undocumented nuances.

Access limitations to ChatGPT customization. The creation of customized agents (Custom GPTs) is a feature available only to users of paid versions of ChatGPT, such as ChatGPT Plus. This requirement may pose an access barrier for institutions or professionals without a subscription, limiting the adoption of the proposed approach in resource-constrained contexts.

Domain-specific limitations. The results obtained are tied to a specific context—a component assembly control system—with carefully prepared documentation and evaluation conducted by authors with deep domain knowledge. Therefore, caution is advised when generalizing these results to other areas, domains, or teams with different levels of technical maturity.

Language-specific behavior. All interactions with the agent, including prompts, input documents, and validations, were conducted in Brazilian Portuguese. This choice was aligned with the real-world context of the system under evaluation but also introduces a relevant limitation: as large language models behave differently across languages, the findings presented here may not be directly generalizable to other linguistic contexts. Furthermore, semantic nuances inherent to Portuguese may have influenced the interpretation of instructions and the generation of test cases. These factors should be carefully considered when replicating or adapting this approach to different domains or languages.

Data sensitivity constraints. Throughout the experiment, particular attention was given to the safeguarding of confidential information related to the system and its users. All documents used in the agent's configuration and interactions were preprocessed and anonymized to avoid exposing sensitive data. This necessity introduced an extra step in the workflow and highlighted ethical and security considerations that must be addressed when applying AI agents in real-world software engineering contexts.

8 Lessons Learned

The practical application of an AI agent based on LLMs for automated test case generation yielded several valuable insights that can inform future implementations of similar solutions. These lessons emerged from both the technical challenges faced and the methodological decisions taken throughout the experiment. The key takeaways are detailed below.

Significance of Well-Structured Input Data. A key insight from the experiment was that the quality of the agent's responses heavily depends on the organization, clarity, and completeness of the provided documents. The creation of the Master Requirements Document, with standardized language, hierarchical structure, and consistent formatting, proved essential for the model's effective performance.

Accuracy Depends on Explicit Instructions. The experiment demonstrated that the model performs more effectively when given clear and specific instructions. Generic commands tend to produce generic responses; in contrast, detailed prompts, such as those specifying a particular technique (e.g., Decision Table Testing) or validation criteria, resulted in more appropriate and targeted outputs.

Technical Constraints Require Workaround Strategies. Platform limitations, such as token limits and the lack of persistent memory, required the adoption of strategies like re-uploading the Master Requirements Document at the beginning of each new session to ensure the agent had full access to the necessary information for generating test cases. Although these constraints did not hinder the agent's use, they require user awareness and thoughtful planning during interactions.

Curation and Validation Remain Indispensable. Although the agent significantly reduced the effort involved in authoring test cases, the process still required active involvement from the QA analyst to review the generated scenarios and ensure alignment with the business rules. This highlights that human oversight remains essential to guarantee the quality and reliability of the outcomes.

Iterative collaboration enhances results. Throughout the experiment, it was observed that interactions conducted in multiple stages, including progressive prompt adjustments, clarification of doubts, and the division of requirements into smaller blocks, resulted in more complete and accurate test scenarios. This behavior suggests that iterative collaboration between the user and the agent supports deeper context exploration and increases the effectiveness of the responses generated.

9 Conclusion and Future Work

This experience report demonstrated the feasibility of using a customized AI agent, built on ChatGPT, to automate the specification of test cases in a real-world system. The results showed significant gains in terms of time efficiency, coverage, and support for validation activities, with the potential to transform traditional practices in software quality assurance processes.

Despite the benefits identified, the study also revealed challenges that pave the way for future investigations. In this regard, key opportunities for further exploration are discussed in the following paragraphs.

Enhancing Visual Interpretation Capabilities with ChatGPT Vision. Although the agent was able to interpret interface prototypes, the use of ChatGPT Vision capabilities still presents opportunities for improvement. Future work may explore ways to integrate visual and textual requirement elements to increase the completeness and accuracy of the generated tests, especially in cases involving dynamic forms or conditional validations.

Automating the Generation of Functional Test Scripts. Based on the scenarios generated by the agent, a natural extension would be the automatic creation of test scripts for tools such as Cypress, Playwright, or Selenium. This approach would enable a more precise evaluation of the level of detail in the generated test cases and their practical applicability within automated testing pipelines.

Reducing Dependence on Human Validation. Despite the productivity gains, manual validation of the generated test cases is still necessary to ensure quality and compliance with the requirements. Future work may explore strategies to reduce this dependency, such as implementing self-assessment mechanisms within the agent, assigning confidence levels to the generated scenarios, and enabling continuous learning from corrections made by analysts. These improvements would enhance the agent's autonomy, reduce the cognitive load on users, and accelerate the validation process.

Evaluation Across Diverse Application Domains. The present evaluation was conducted within a system specific to the component assembly domain. As future work, it is recommended to replicate the approach in other domains (such as e-commerce, finance, or education), investigating the adaptability of the agent's configuration and the effectiveness of test generation in different contexts.

These directions offer significant opportunities to advance the use of LLM-based agents in software testing processes, fostering the ongoing evolution of validation, verification, and quality assurance strategies from an AI-oriented perspective.

ACKNOWLEDGMENTS

The authors gratefully acknowledge OpenAI for providing ChatGPT, which served as a supportive tool in the generation and revision of portions of this manuscript. We also express our sincere gratitude to Viviane Bernardo for her valuable contributions during the manuscript revision process. Additionally, we acknowledge the support provided by FPFtech, whose institutional backing was fundamental to the development of this work.

REFERENCES

[1] Mohammad Baqar and Rajat Khanda. 2025. The Future of Software Testing: AI-Powered Test Case Generation and Validation. In *Proceedings of the Computing Conference: Intelligent Computing*. Springer, 276–300. doi:10.1007/978-3-031-92605-1_18

[2] John Berryman and Albert Ziegler. 2025. *Prompt Engineering for LLMs: The Art and Science of Building Large Language Model–Based Applications*. O'Reilly Media, Inc., Sebastopol, CA, USA. Print edition.

[3] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. System Test Case Design from Requirements Specifications: Insights and Challenges of Using ChatGPT. *arXiv preprint* arXiv:2412.03693 (2024). arXiv:2412.03693 [cs.SE] Published on arXiv Dec 4, 2024.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. *arXiv preprint arXiv:2005.14165* (2020). doi:10.48550/arXiv.2005.14165

[5] ChatGPT Brasil. 2024. *ChatGPT Vision*. https://chatgpt.com.br/chatgpt-vision/ Informational web page; accessed 2025-06-30.

[6] Sabit Ekin. 2024. *Prompt Engineering for ChatGPT: A Quick Guide to Techniques, Tips, and Best Practices*. doi:10.36227/techrxiv.22683919.v2 Preprint (TechRxiv); generated by OpenAI's ChatGPT and edited by Sabit Ekin. Accessed 2025-06-24.

[7] Navid Bin Hasan, Md Ashraful Islam, Junaed Younus Khan, Sanjida Senjik, and Anindya Iqbal. 2025. Automatic High-Level Test Case Generation Using Large Language Models. *arXiv preprint* arXiv:2503.17998 (2025). arXiv:2503.17998 [cs.SE] Published on arXiv Mar 23, 2025.

[8] Caio Jordan de Lima Maia. 2023. Teste de Aceitação Gerado por Inteligência Artificial: Um Estudo Exploratório sobre Seu Potencial de Uso por Analistas de Teste. (2023). https://repositorio.ufpb.br/jspui/handle/123456789/31509 Undergraduate thesis, Universidade Federal da Paraíba, João Pessoa, Brazil. In Portuguese.

[9] Bernard Marr. 2024. *Generative AI in Practice: 100+ Amazing Ways Generative Artificial Intelligence Is Changing Business and Society*. John Wiley & Sons, Inc., Hoboken, NJ, USA. Print edition.

[10] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. *arXiv preprint* abs/2402.06196 (2024). arXiv:2402.06196 [cs.CL] Published on arXiv Feb 5, 2024.

[11] OpenAI. 2023. *Introducing GPTs*. https://openai.com/index/introducing-gpts/ OpenAI blog post; accessed 2025-06-27.

[12] OpenAI. 2024. *ChatGPT*. https://openai.com/index/chatgpt/ OpenAI product page; accessed 2025-06-20.

[13] OpenAI. 2024. *Memory and New Controls for ChatGPT*. https://openai.com/blog/memory-and-new-controls-for-chatgpt Blog post; accessed 2025-07-14.

[14] OpenAI. 2024. *Thinking with Images*. https://openai.com/index/thinking-with-images/ Informational web page; accessed 2025-06-30.

[15] OpenAI. 2024. *What Are Tokens and How to Count Them?* https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them OpenAI Help Center article; accessed 2025-07-15.

[16] OpenAI. 2025. *Prompt Engineering*. https://platform.openai.com/docs/guides/text#prompt-engineering OpenAI API documentation; accessed 2025-07-15.

[17] OpenAI. 2025. Prompt engineering best practices for ChatGPT. https://help.openai.com/en/articles/10032626-prompt-engineering-best-practices-for-chatgpt Accessed: 2025-07-12.

[18] OpenAI. 2025. *Tokens*. https://platform.openai.com/docs/concepts/tokens OpenAI API Documentation; accessed 2025-06-25.

[19] Harshad Vijay Pandhare. 2024. From Test Case Design to Test Data Generation: How AI Is Redefining QA Processes. *International Journal of Engineering and Computer Science* 13, 12 (2024), 26737–26757. doi:10.18535/ijecs.v13i12.4956

[20] Saksorn Ruangtanusak, Pittawat Taveekitworachai, and Kunat Pipatanakul. 2025. Talk Less, Call Right: Enhancing Role-Play LLM Agents with Automatic Prompt Optimization and Role Prompting. *arXiv preprint arXiv:2509.00482* (2025). doi:10.48550/arXiv.2509.00482

[21] Ian Sommerville. 2011. *Engenharia de Software* (9 ed.). Pearson Prentice Hall, São Paulo, Brazil. Technical review by Kechi Hirama; translated from the original *Software Engineering*.

[22] Arailym Talasbek. 2023. Artificial AI in Test Automation: Software Testing Opportunities with OpenAI Technology — ChatGPT. *Journal of Emerging Technologies and Computing* 62, 1 (2023), 5–14. doi:10.47344/sdubnts.v62i1.912

[23] Gaolei Yi, Zizhao Chen, Zhenyu Chen, W. Eric Wong, and Nicholas Chau. 2023. Exploring the Capability of ChatGPT in Test Generation. In *Proceedings of the 23rd IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 72–80. doi:10.1109/QRS-C60940.2023.00013

[24] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. *arXiv preprint* abs/2303.18223 (2023). arXiv:2303.18223 [cs.CL] Published on arXiv Mar 31, 2023.