Automating Software Test Case Creation with Generative AI

Shirisha Gongati

***Abstract*—Manual test-case creation is one of the most time-consuming, repetitive, and error-prone tasks in software quality assurance. Large Language Models (LLMs) such as GPT-4o and Llama3.1 have recently demonstrated strong capabilities in natural-language reasoning and structured output generation. This work presents an AI-assisted test-case generation framework capable of producing complete, standardized manual test cases from natural-language requirements. The system features both cloud-based and offline LLM support, a simple graphical user interface (GUI) for non-technical users, structured output formatting, and robust error-handling mechanisms. Evaluation across 15 functional requirements shows that the system reduces test-case creation time by approximately 75% while maintaining high accuracy and consistency. The results suggest that LLM-based assistance can be a practical and impactful addition to modern QA workflows.**

***Index Terms*—Software Testing, Test Case Generation, Large Language Models, Generative AI, Automation, Requirement Analysis.**

I. Introduction and Motivation

Software testing plays a crucial role in ensuring the reliability, safety, and user satisfaction of software systems. Within the software testing lifecycle, the design of manual test cases is a particularly costly activity. Test engineers must carefully read natural-language requirements, infer testable conditions, decide on input combinations, and describe step-by-step procedures together with expected outputs. This work is not only time-consuming, but also highly repetitive and susceptible to human error.

In industrial practice, test teams often operate under strict time and budget constraints. For rapidly evolving systems, such as web and mobile applications, requirements change frequently and test suites must be continuously updated. As a result, teams struggle to maintain comprehensive coverage, especially for negative and edge-case scenarios. Traditional automation frameworks such as Selenium, Cypress, or Appium still require manually authored test cases before scripts can be implemented.

Recent progress in Large Language Models (LLMs) has opened up new possibilities for automating parts of this process. Models like GPT-4o and Llama3 can interpret free-form text, reason about implied behavior, and generate structured content. This naturally motivates the question: *Can LLMs be used to automatically generate high-quality manual test cases from natural-language requirements?*

This project investigates that question by building an LLM-driven test-case generation assistant. The goal is to provide a practical tool that reduces repetitive work for testers, improves consistency in documentation, and forms a foundation for subsequent automation.

S. Gongati is with the Department of Computer Science, Ohio University. Email: sgxxx@ohio.edu.

II. Related Work and Literature Review

AI and NLP have been applied to software testing in several ways. Zhang et al. introduced TestPilot [1], which uses LLMs to generate test cases from natural-language descriptions and showed promising results in terms of coverage and diversity. Chen et al. presented LLM4Test [2], a systematic study of the effectiveness of LLM-generated test cases across multiple models and domains, highlighting both strengths and common failure modes.

Ferrari et al. explored the use of NLP techniques to support requirement analysis [3]. Their work focused on extracting domain concepts, behavioral constraints, and ambiguities from textual requirements, demonstrating that language technologies can significantly aid early-phase software engineering tasks.

Industrial tools such as TestSigma AI and Katalon TestOps AI also offer AI-assisted test design. TestSigma allows users to express tests in near-natural language which are then converted into executable scripts, whereas Katalon uses AI to analyze existing automated suites and recommend improvements. Microsoft TestGen focuses on automatic unit-test generation at the code level.

Despite these advances, several gaps remain. Many tools:

- are tightly coupled to a vendor-specific automation stack,
- emphasize automated scripts rather than reusable manual test cases,
- do not support offline or on-premise execution, and
- provide limited control over the underlying AI prompts and output format.

The system proposed in this paper directly addresses these limitations by focusing on flexible, template-based manual test-case generation, with both cloud and local LLM backends and a lightweight GUI designed specifically for QA engineers.

III. Foundational Work

Before implementing the complete system, several foundational investigations were performed.

A. Requirement Understanding Experiments

Initial experiments evaluated how different LLMs interpret typical software requirements. Both short and long requirements were tested, including login flows, account lockout

rules, and shopping cart behavior. Multiple prompt styles were tried, ranging from open-ended queries to highly constrained instructions specifying the desired output sections. These experiments indicated that:

- structure in the prompt strongly improves structure in the output, and
- explicitly asking for "preconditions", "steps", and "expected results" produces more consistent test cases.

B. Test Case Structure Definition

A standardized format was defined to make AI-generated test cases directly usable by testers and compatible with common test management tools:

- **Test Case ID** – a label or identifier,
- **Description** – high-level intent of the test,
- **Preconditions** – system or data state required before execution,
- **Test Steps** – ordered actions to be performed,
- **Expected Results** – observable outcomes for each step or for the scenario as a whole,
- **Priority** – optional indication of business criticality.

All prompts and output-processing routines enforce this structure.

C. Model Selection

GPT-4o, GPT-3.5, and Llama3.1 were compared qualitatively. GPT-4o produced the most detailed and precise test cases, especially for complex multi-step workflows. Llama3.1, deployed locally via the Ollama runtime, generated slightly shorter outputs but offered strong privacy guarantees and zero per-call cost. Based on these observations, the final system supports both models and allows the user to choose the backend.

IV. Similar Tools

Several tools share overlapping goals with this work.

A. TestSigma AI

TestSigma allows testers to write test scenarios in a semi-natural language which are then converted into executable automated scripts. While powerful, the tool primarily targets script generation for its own execution platform and does not focus on reusable manual documentation.

B. Katalon TestOps AI

Katalon TestOps provides dashboards and AI-based analytics to identify gaps and redundancies in automated suites. It is less concerned with initial test design and does not generate requirement-level manual test cases.

C. Microsoft TestGen and General LLM Assistants

Microsoft TestGen automatically generates unit tests from source code. General assistants like ChatGPT can be prompted to write test ideas, but their responses are ad hoc and not bound to a particular template or workflow.

The proposed system is intentionally lightweight and framework-agnostic. It aims to produce high-quality manual test cases that can be consumed by any QA team, regardless of automation stack, and to do so with both cloud-based and local models depending on organizational constraints.

V. Design and Architecture

The high-level system architecture is shown in Fig. 1. It consists of five main modules: a GUI input layer, a prompt engineering module, an LLM backend, an output formatter, and an error-handling and logging component.

A. GUI Input Layer

The front-end is implemented in a Jupyter Notebook using `ipywidgets`. Testers enter natural-language requirements into a text box, select the desired model (GPT-4o or Llama3.1), and click a button to trigger generation. Results are displayed in a scrollable output area that supports copying into external tools.

B. Prompt Engineering Module

The prompt engineering module is responsible for converting the raw requirement text into a carefully crafted instruction for the LLM. It embeds the requirement inside a template that clearly specifies:

- the role of the model as a senior QA engineer,
- the required test-case structure,
- expectations for positive, negative, and edge scenarios when applicable, and
- formatting details, such as bullet lists for steps.

C. LLM Backend

The core reasoning is handled by the LLM backend. For online use, GPT-4o is accessed via the OpenAI API. For offline or privacy-sensitive scenarios, Llama3.1 is invoked through the Ollama local runtime. Both backends are wrapped by a unified interface so that the rest of the system is agnostic to which model is selected.

D. Output Formatter

Responses from the LLM often contain additional explanation or commentary. The output formatter cleans this text, extracts the sections corresponding to the predefined template, and presents the final test case in a compact, readable format.

E. Error Handling and Logging

The system includes checks for empty input, exception handling around network calls, and a basic logging mechanism for diagnostics. If the cloud model is unavailable, the system can fall back to the local model.

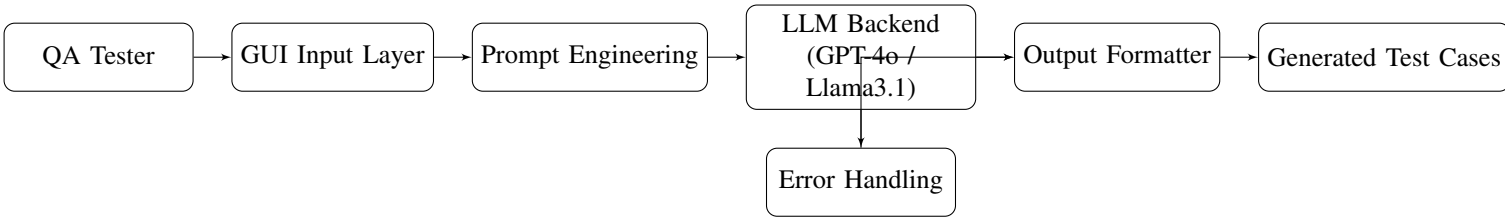

Fig. 1. System architecture of the proposed LLM-based test-case generation assistant.

Fig. 2. Graphical user interface of the test-case generator showing requirement input and test-case generation in progress.

VI. Development Workflow

The overall development workflow is summarized in Fig. 3 and described below.

A. Environment Setup

The project was implemented in Python using Jupyter Notebook. Dependencies included the OpenAI SDK, the Ollama client, `requests` for HTTP calls, and `ipywidgets` for the GUI. The environment was configured so that either backend model could be enabled with minimal changes.

B. Model Integration

Wrapper functions were implemented to send prompts to GPT-4o or Llama3.1 and return their responses. These functions encapsulate JSON payload construction, parameter selection (such as temperature and maximum tokens), and error handling for timeouts or unexpected responses.

C. Interface Implementation

The GUI was built iteratively. Early prototypes simply printed raw model output; later versions added model selection controls, improved layout, and structured display of test cases. Particular care was taken to avoid blocking the notebook kernel during long-running calls.

D. Prompt Refinement and Iteration

Prompt templates were refined based on manual inspection of generated test cases. Several iterations were necessary to consistently obtain well-structured sections and to minimize hallucinated requirements not present in the original text.

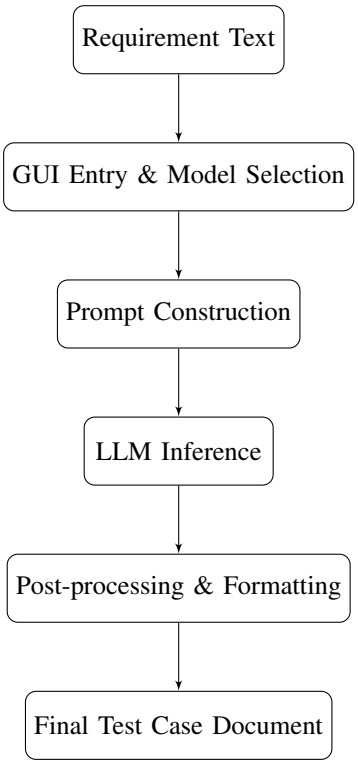

Fig. 3. End-to-end workflow from requirement input to final test-case document.

VII. Testing and Evaluation

To evaluate the system, 15 representative functional requirements were collected from typical web-application scenarios, including:

- user login and logout,
- password reset via email,
- account lockout after multiple failed attempts,
- adding, updating, and removing items from a shopping cart, and
- payment processing with both successful and failed transactions.

Each requirement was submitted to both GPT-4o and Llama3.1 through the system interface. Generated test cases were evaluated along the following dimensions:

- **Accuracy** – whether the test case behavior matched the requirement.
- **Completeness** – presence of preconditions, steps, and expected results.
- **Clarity** – readability and unambiguity of the steps.

- **Structure** – adherence to the defined template.

Overall, GPT-4o produced the most detailed test cases, often including secondary negative paths that were only implicit in the original requirement. Llama3.1 generated more concise cases but still met the quality criteria for most scenarios. Manual timing measurements indicated that preparing a new test case using the assistant (including human review) took roughly one quarter of the time compared to writing the same case from scratch, corresponding to an approximate 75% reduction in authoring effort.

VIII. Conclusions and Future Work

This paper presented an LLM-based assistant for generating structured manual test cases from natural-language requirements. By combining prompt engineering, a standardized test-case template, dual cloud/local model support, and an intuitive GUI, the system significantly reduces repetitive work for QA engineers while improving consistency and coverage.

Future work includes exporting test cases directly to Excel or CSV formats, integrating with tools such as Jira or TestRail, automatically generating families of negative and edge-case tests, and deploying the assistant as a standalone web or desktop application for broader adoption.

References

[1] J. Zhang, S. Wang, and M. Harman, "TestPilot: Automated Test Case Generation Using Large Language Models," 2023.
[2] T. Chen, S. Jiang, and Y. Zhou, "LLM4Test: Can Large Language Models Generate Effective Test Cases?," 2023.
[3] A. Ferrari, S. Gnesi, and G. Tolomei, "Using NLP to Support Requirement Analysis in Software Engineering," 2018.
[4] H. Hemmati, A. Arcuri, and L. Briand, "Achieving Scalable Model-Based Testing Through Test Case Diversity," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 657–674, 2013.
[5] R. Pandita, X. Xiao, W. Weimer, and S. M. Kim, "Using Test Oracles Generated from Natural Language to Detect Regression Faults," in *Proc. IEEE/ACM Int. Conf. on Software Engineering (ICSE)*, 2016.