



FIT3164 Data Science Project 2

Final Written Report

AI-Driven Web Testing Automation

Prepared by

Team MDS05

Chia Xin Yue | 33942811 | cchi0095@student.monash.edu

Lim Yue Syuen | 34003568 | ylim0101@student.monash.edu

Zoe Yow Cui Yi | 33214476 | zyow0001@student.monash.edu

Supervised by

Mr. Yip Wai Hoong (NXP)

Ms. Kamalahshunee K Velautham

Dr. Lan Boon Leong

Date: Thursday, 6 November 2025, 11:55 PM

Word Count: 3317

Table of Contents

Authorship Statement.....	3
1 Introduction.....	4
1.1 Project Background.....	4
1.2 Project Objectives.....	4
1.2.1 Overview.....	4
1.2.2 Request classification.....	4
1.2.3 Test case execution flow.....	4
1.2.4 Test script generation flow.....	4
1.3 Report Overview.....	5
2 Project background summary.....	5
3 Outcomes.....	6
3.1 Results, Achievements and Delivered Product.....	6
3.2 Alignment with Requirements Traceability Matrix (RTM).....	7
3.3 Requirements Met and Unmet.....	7
3.4 Limitations, Potential Improvements and Future Work.....	7
4 Methodology.....	8
4.1 Project Design and Rationale.....	8
4.1.1 Overall System Workflow.....	9
4.1.2 Overall System Architecture.....	10
4.2 Implementation Process.....	10
4.2.1 Implementation Steps.....	10
4.2.2 Software Tools Used and Justification.....	11
5 Software and Project Critique.....	13
5.1 Overall Project Execution.....	13
5.2 Comparison with Initial Project Proposal.....	13
5.2.1 What changed, and why.....	14
5.3 Factors Behind Project Success.....	14
5.4 Reflection on Unsuccessful Aspects.....	14
6 Conclusion.....	15
7 References.....	15
8 Appendix.....	17
Appendix A.....	17
Appendix B.....	19
Appendix C.....	20
Appendix D.....	21

Authorship Statement

Chia Xin Yue wrote the draft and finalised the project outcomes and methodology part of the report. She also drafted the Requirements Traceability Matrix (RTM) in Appendix A, and added the chatbot interface screenshots in Appendix B.

Lim Yue Syuen drafted and finalised the introduction, the flow chart in the methodology, software and project critique, and conclusion. She also finalised the Requirements Traceability Matrix (RTM) in Appendix A and added the end product screenshot for the Selenium Python script generation in Appendix C.

Zoe Yow Cui Yi wrote the draft and finalised the introduction's report content summary, project background summary, and the references. She also added the sample code in Appendix D and finalised the report style and presentation.

Chia Xin Yue, Lim Yue Syuen, and Zoe Yow Cui Yi finalised the report's content.

Google Doc Link:

https://docs.google.com/document/d/1RbJfzCU3ICb1IYP9Gi_u1k5lrSYp1MAQyLVg4ZORSI0/edit?usp=sharing

1 Introduction

This project proposes a chatbot-driven test automation framework for executing test cases on NXP Semiconductors' CamStar webpage. Led by Project Manager Zoe Yow, with Yue Syuen as Technical Lead and Xin Yue as Quality Assurer, the team will work closely with external supervisor Mr. Yip Wai Hoong from NXP. Internal supervision will be provided by Ms. Kamalahshunee K. Velautham (School of IT) and Dr. Lan Boon Leong (School of Engineering) from Monash University.

1.1 Project Background

Currently, test executions on the CamStar webpage are carried out manually. While straightforward, this approach is time-consuming, error-prone, and lacks scalability. With increasing digitization in semiconductor manufacturing and a growing need for rapid validation cycles, there is a strong business case for transitioning toward intelligent test automation.

1.2 Project Objectives

1.2.1 Overview

A production-ready chatbot web application designed for the NXP CamStar webpage. Users type plain-English requests. The bot classifies each request to either execute existing test cases or generate new Python Selenium test scripts. It confirms plans, collects required variables, runs tests, streams progress, captures screenshots and logs, and stores outputs. Reports and scripts are saved in a database-backed storage layer. The UI shows the saved file path for view or download.

1.2.2 Request classification

An intent classifier routes each prompt to one of two paths: Execute Test Case or Generate Test Script. Low-confidence cases are returned to the user for clarification before any action.

1.2.3 Test case execution flow

The bot retrieves the closest predefined test case using semantic search. It displays the matched case, required variables, and planned steps for confirmation. It then validates user-provided variables and starts execution. Progress streams in real time and screenshots of the execution steps are captured. When the run finishes, the system builds a structured report with the screenshots and a summary, saves it to storage, and returns the file path for viewing or downloading.

1.2.4 Test script generation flow

The bot accepts an input file (Selenium IDE generated Python file) and optional instructions. It extracts requirements, then generates a Python Selenium script that conforms to the execution system. The script is then saved to storage, and the UI displays the saved file path for view or download.

1.3 Report Overview

As outlined in the introduction, this report presents the development and delivery of our chatbot-driven test automation framework for NXP's CamStar webpage with its background and objectives. Building on that foundation, the remainder of the report is structured as follows:

- 1. Project background summary:** Summarises the project rationale and key literature.
- 2. Project Outcomes:** Discusses the results and achievements of the project.
- 3. Methodology:** Outlines the development process of the project, focusing on the final design rationale and how it was implemented. It includes an explanation of design choices, supported by diagrams, and details the software tools used throughout development, along with how they contributed to building the final product.
- 4. Software and Project Critique:** Reflects on the overall execution of the project by comparing the final outcome with the initial proposal. It discusses changes in approach, reasons for any deviations, and evaluates both successful and less effective aspects of the project, including management, software tools, and communication
- 5. Conclusion:** Summarises the entire project report.

2 Project background summary

As industrial systems grow more complex and software release cycles accelerate through agile methodologies, traditional manual testing approaches have become inefficient and error-prone. At NXP, testers must manually execute repetitive validation steps on the Camstar web platform, leading to time-consuming and inconsistent outcomes. This project addresses these limitations by developing a chatbot-driven web testing automation framework that uses natural language interaction, semantic retrieval, and browser-based automation to execute or generate test cases automatically.

Despite clear advancements, several key gaps persist across existing research and tools:

- **Semantic Search:** Existing embedding-based methods focus on static data retrieval and do not address dynamic, real-time use cases within chatbot-driven testing workflows.
- **Test Automation:** Reinforcement learning and GenAI approaches offer adaptive test selection, but lack integration with natural language interfaces that can make testing more accessible to non-technical users.

- **AI-driven Chatbot Systems:** While Large Language Model (LLM) powered chatbots are advancing in productivity applications, they have not been applied to the domain of test automation or designed to handle semantic retrieval of test cases followed by automated execution.

No prior system unifies these streams into a cohesive, chatbot-driven semantic test automation architecture suitable for industrial software environments like NXP’s Camstar web platform. By integrating technologies such as Anthropic Claude for language interpretation, PGVector for semantic search, and Selenium for automation, the project bridges the gap between human requests and automated testing execution, enhancing accessibility, accuracy, and efficiency in industrial testing environments.

3 Outcomes

3.1 Results, Achievements and Delivered Product

The project successfully delivered a fully functional AI-driven automated web testing system integrating a chatbot interface, Selenium-based test execution and reporting, and Claude AI-assisted script generation.

The final product features a Streamlit chatbot interface (Appendix B) consisting of two sections:

- Left panel: Provides a “Save as JSON” function for chat export and detailed usage instructions.
- Main panel: Serves as the primary conversational area, allowing users to (i) execute existing test cases or (ii) generate new test scripts.

For test execution, the chatbot retrieves relevant test cases, verifies user input e.g. lot numbers, and executes them via Selenium. A Word-based report containing screenshots of Selenium web interactions, pass/fail statuses, and error messages is generated automatically and displayed through a file path.

For test script generation, Claude AI analyses user prompts and produces optimized Selenium Python scripts according to internal coding standards. These are displayed and stored in the repository.

This architecture was chosen to ensure modularity, maintainability, and user accessibility within NXP’s secure environment. Streamlit was selected for its lightweight deployability, while Claude AI was chosen for its high-quality code generation and context handling.

Overall, all three targeted test cases executed successfully with complete logging and reporting, achieving the intended goal of simplifying complex test automation through natural language.

3.2 Alignment with Requirements Traceability Matrix (RTM)

All functional requirements outlined in the RTM were met and verified during testing and demonstration.

Please refer to Appendix A for the complete RTM; a summary is provided below.

Requirement Category	Description	Status
Automated Test Execution and Logging	Execute and log 3 core test scenarios	Completed
Chatbot Interface	Enable user interaction, progress updates, and result display	Completed
Generative AI for Script Generation	Integrate Claude AI to optimise Selenium scripts	Completed
Prompt Engineering	Refined prompts for test retrieval and test script optimisation	Completed

3.3 Requirements Met and Unmet

As mentioned, all functional requirements in the RTM were successfully met and validated through successful testing and demonstration.

For non-functional requirements, project documentation is maintained in NXP's Confluence workspace and kept at the latest version, so this requirement (R5.1) is met. Version control with GitLab is not required because all code and files are managed directly on a remote server. The original requirement was meant for projects developed and versioned on local laptops, which does not apply to this setup.

3.4 Limitations, Potential Improvements and Future Work

While the project achieved full alignment with planned objectives, several limitations and areas for improvement were identified

Limitation	Description	Suggested Improvement/Future Work
Prompt length	The current prompt is long	Future iterations should focus on reducing prompt length while maintaining functionality. Instead of embedding full test-case details directly into the prompt, the system could utilise database queries to retrieve test cases dynamically. This would make prompt generation more efficient, scalable, and maintainable.

Chatbot history	Chat history is only within the session	To support multiple users and persistent memory across sessions, future work could involve storing conversation history in a PostgreSQL database with PGVector for semantic retrieval or leveraging cloud storage solutions such as Amazon Web Services (AWS). This would improve both user experience and long-term traceability.
Cloud integration	Users must manually copy file paths to access generated reports	At present, users must manually copy file paths to access generated reports. A valuable enhancement would be to automate report uploads and retrieval through NXP's internal cloud systems or SharePoint, enabling seamless access and centralised documentation.
Test Execution Reliability	Occasional click failures due to dynamic web elements or load times	Implement adaptive waiting mechanisms or AI-based visual element detection would enhance the robustness and reliability of the automated testing process.

4 Methodology

4.1 Project Design and Rationale

The project's design aimed to improve the traditional test automation process which was time-consuming and error-prone by enabling users to generate and execute test scripts through natural language.

4.1.1 Overall System Workflow

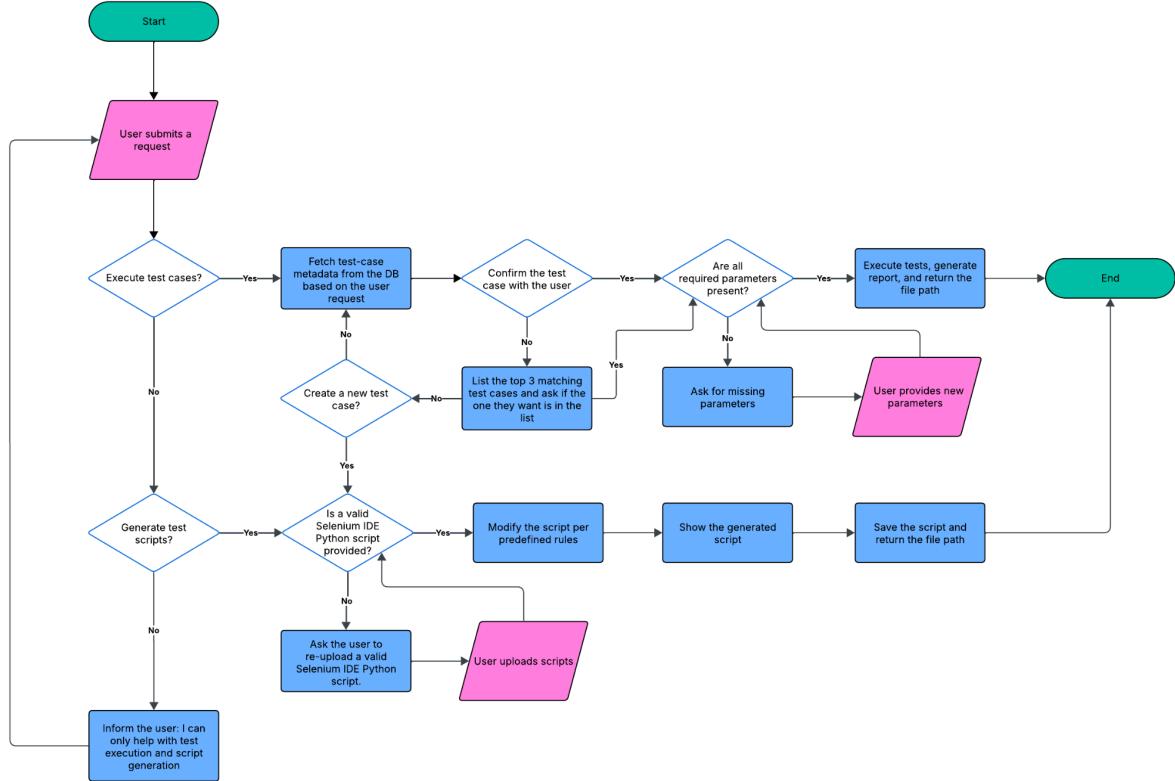


Figure 1: Full flow chart of the chatbot system

The overall user flow, illustrated above, demonstrates how a user's request is processed end-to-end within the system.

When a user submits a request via the chatbot, the system first determines whether the intent is to execute existing test cases or generate a new test script.

- For **test execution**, the chatbot uses the LLM (Anthropic Claude) to retrieve relevant test-case metadata from the PGVector database, confirm it with the user, and verify that all required parameters (such as the selection ID) are provided. Missing inputs are requested interactively before the system proceeds to execute the tests through Selenium. Upon completion, it generates a Word report containing execution screenshots and pass/fail status, saves it to the database, and returns the file path to the user.
- For **script generation**, the chatbot invokes the LLM (Anthropic Claude) to rewrite the user-uploaded Python script exported from Selenium IDE, following predefined rules in the system prompt. The rewritten script streams to the interface as it is generated so the user can review it before downloading. The final script is stored in the database, and the file path is shown to the user.

This branching flow ensures robustness and usability. The chatbot handles invalid inputs, prevents redundant actions like repeated logins, and maintains execution continuity even when partial errors occur. This design ensures smooth automation aligned with real-world testing workflows at NXP.

4.1.2 Overall System Architecture

The system was designed and implemented as an AI-driven automated testing framework that integrates a chatbot interface, Generative AI-based script generation, and Selenium-powered test execution and reporting. The key design goal was to simplify test automation for business analysts by enabling them to trigger tests or generate scripts through natural language, without the need to write code.

This modular design was chosen for maintainability, scalability, and fault tolerance, allowing each component to operate independently while supporting seamless end-to-end automation. The system follows a modular architecture consisting of four main components:

1. **Chatbot Interface (Frontend):** Built with Streamlit and Python to serve as the main user interaction point. It handles user requests (e.g. “Test lot on hold”), confirms test cases, gathers required parameters, and displays real-time updates and results. This design lowered the barrier to entry for testers without coding backgrounds and streamlined interaction with the automation engine.
2. **Generative AI Module (Claude AI):** Integrated through API calls to interpret user intent and generate Selenium-compatible Python scripts. This automates the initial script creation process, reducing manual coding effort.
3. **Selenium Automation Engine (Backend):** Executes validated scripts directly on the Camstar web application. Selenium WebDriver handles browser interactions, screenshots, and exception recovery to ensure uninterrupted execution.
4. **Reporting and Logging Module:** Automatically saves execution results, screenshots, and pass/fail comparisons into a formatted Word report using the python-docx library. This helps solve the problem of manual documentation of test results which was tedious and inconsistent.
5. **PGVector Database Integration:** Using PostgreSQL with PGVector extension allows semantic storage and retrieval of test cases based on meaning rather than exact keywords.

4.2 Implementation Process

4.2.1 Implementation Steps

The implementation process followed an iterative approach consisting of several key stages:

1. Requirement Analysis and Planning:

System requirements were defined collaboratively with NXP supervisors and documented in a Requirement Traceability Matrix (RTM).

2. Architecture and Workflow Design:

The system flow (Figure 1) was developed to visualise user interactions from request submission to report generation. This ensured logical handling of both test execution and script generation paths.

3. Chatbot Development:

Implemented using Streamlit's event-driven model, enabling users to submit test or script requests. Input validation and branching logic were added to guide users interactively.

4. Claude AI Integration for Script Generation:

Used API requests to send structured prompts and parameters e.g. test case type, variable names to Claude. The model returned Python scripts that were refined by the system to meet internal coding standards.

5. Selenium Test Execution and Logging:

Implemented automated browser operations using Selenium. Test execution results, screenshots, and timestamps were collected and stored in a Word document automatically.

6. Testing, Validation, and Final Integration:

Conducted validation using three key NXP test scenarios: Print Run Card, Lot Trigger Immediate Hold, and Lot Release. Each successful execution confirmed full alignment with the requirements listed in the RTM.

4.2.2 Software Tools Used and Justification

Software Libraries	Justification
Selenium	<ul style="list-style-type: none">• Open-source tool for automating web page (CamStar) testing which makes it easier to generate test case scripts.• Chosen because it supports Python (which we are familiar with) while existing automation testing software tools, like Katalon Studio, does not support Python (Wikipedia contributors, 2025a).• Has extensive browser compatibility, extending from major browsers e.g. Chrome, Firefox and Edge (which we will be mainly using), allowing for cross-browser testing and any user to conduct testing without considering what browser they use (Automation, 2023).
Pytesseract	Used to validate outputs of certain test cases by extracting text from PDF files through Optical Character Recognition (OCR).
Python Imaging	Used to support Pytesseract by opening and handling image objects

Library (PIL)	extracted from PDF files for OCR.
Pathlib	Used to create and manage file directories required for saving and organizing test-related documents and images.
Pdf2image	Used to convert PDF pages into image formats, so that the OCR or image comparisons (OpenCV) can be applied.
OpenCV	Used to match PDF images with predefined templates by identifying exact pixel locations, helpful in validating test output layout and structure.
Requests	Facilitates in sending HTTP requests in Python, used for interacting with external APIs or services, like executing Selenium during test case execution.
Boto3	Used as the AWS Software Development Kit for Python to: <ul style="list-style-type: none"> • Create a session and bedrock-runtime client. • Invoke the Anthropic Claude model with response streaming to handle user input, emit JSON variables, and return rewritten test scripts. • Boto3 is transport only. The LLM does the understanding and rewriting.
Langchain	Acts as the middleware to the chatbot interface with the PostgreSQL PGVector database, allowing semantic search and retrieval of relevant test cases based on vector similarity (Ws, 2025).
Flask	Used to connect frontend (Streamlit chatbot) and backend (Selenium and LLM), handling requests and responses.
Sys (System-specific parameters and functions)	Enables passing a list of command line arguments to a Python script, particularly useful when executing Python scripts in the background (Subprocess).
Subprocess	Allows test case execution Python scripts to run asynchronously in the background without blocking the main program flow.
Pythondocx	Used to auto-generate formatted Word documents summarizing test case results for end users.
Pywin32	Supports the creation of file objects within Word documents and converts them into PDFs, finalizing output documents for user viewing.
Streamlit	<ul style="list-style-type: none"> • Open-source framework to rapidly build and share machine learning and data science web apps. • Used for building the UI design of the chatbot. • Lack of entry requirements for web development skills such as HTML, CSS, JavaScript etc. This is extremely important for the

	team because we have limited knowledge and experience with advanced web development tools.
Dotenv	Used to load environment variables for the Streamlit app with <code>load_dotenv</code> , keeping secrets out of code and enabling local <code>.env</code> configuration
Selenium IDE	<ul style="list-style-type: none"> Utilized to efficiently record user interactions on the Camstar platform, enabling automated generation of Python-based test scripts. This approach simplifies the process for users by reducing manual coding effort and ensuring accuracy in capturing workflows. The generated scripts can then be uploaded into the chatbot for further refinement, streamlining test development and enhancing overall productivity.

5 Software and Project Critique

5.1 Overall Project Execution

The project was executed effectively, achieving full alignment with the original objectives outlined in the proposal. The team successfully delivered an AI-driven test automation framework that automates web testing through natural language prompts. The design, implementation, and integration stages were completed within schedule, and collaboration with NXP ensured that the system was aligned with industrial requirements. Internal communication was managed through structured weekly meetings and agile iteration cycles, enabling rapid troubleshooting and version control across contributors.

Despite the overall success, some integration and debugging phases took longer than expected due to the fact that we were initially unfamiliar with Streamlit's rerun and state model. Early code coupled UI rendering with execution logic, which produced unintended refreshes and inconsistent control flow. We refactored to use explicit session state and phase-based orchestration, which stabilized the UI and integration points.

5.2 Comparison with Initial Project Proposal

The proposal targeted a chatbot that executes predefined Selenium tests and can generate new ones from natural-language prompts. The final design kept the same goal but changed how users express test intent and how scripts enter the system.

5.2.1 What changed, and why

1. **Structured test context in JSON:** We store test metadata as JSON rather than only a name and path. This improved retrieval fidelity and parameter validation during runs.
2. **Composite scenario - “Life of Lot”:** We introduced a chained test that runs all three core scripts: Print Run Card, Lot Trigger Immediate Hold, and Lot Release, in one flow to mirror real operations.
3. **From free-text steps to Selenium IDE uploads:** Users now record steps in Selenium IDE, export a Python script, upload it, then let LLM rewrite it with rule-based prompts to conform to our execution environment. This reduced ambiguity and raised success rates for execution.
4. **Prompt-engineered code transformation:** We added strict output schemas and prompt rules so the LLM produces compatible Selenium code and predictable plans.

These deviations were deliberate. JSON context tightened the contract between the chatbot and backend. The “Life of Lot” composite provides an end-to-end scenario that combines existing cases into a single run, saving time for users. Early trials showed that natural-language descriptions of step-by-step actions led to inconsistent plans and brittle execution. Recording with Selenium IDE preserves exact selectors and timings, while LLM rewriting enforces our conventions.

5.3 Factors Behind Project Success

Several aspects contributed to the project’s success:

1. **Effective communication:** Weekly syncs and documentation through shared repositories enable consistent progress.
2. **Tool synergy:** The combination of Selenium, Claude AI, PostgreSQL, and Streamlit ensured a cohesive system capable of end-to-end test automation.
3. **Agile Methodology:** Incremental sprints allowed frequent validation with supervisors, ensuring that requirements were met continuously.

5.4 Reflection on Unsuccessful Aspects

No major unsuccessful aspects were identified. All planned objectives were achieved, and the final implementation met the initial proposal. Challenges such as Streamlit state handling, model accuracy, and integration complexity were resolved through refactoring, tighter prompt design, and tool updates. These were typical of iterative development rather than failures. The delivered system is stable, fully functional, and demonstration-ready at production quality, reflecting effective teamwork, sound planning, and timely adaptation to technical constraints.

6 Conclusion

This report represented an AI-driven web-testing framework for NXP's Camstar system that lets users trigger test executions via a chatbot, classify intents, run predefined Selenium cases, and generate compatible scripts from Selenium IDE uploads. The system streams progress, captures screenshots, and writes structured reports, with test context stored as JSON for accurate retrieval. A composite "Life of Lot" scenario chains the three core test functions: Print Run Card, Lot Trigger Immediate Hold, and Lot Release to mirror real operations. Outcomes align with the updated RTM and the project aims to reduce manual effort while improving reliability for industrial use.

Methodologically, the build combined Streamlit, Flask, Selenium, PGVector, and an LLM with prompt-engineered contracts. Execution quality improved after moving from a local Llama model to Claude. The critique surfaced key limitations including session memory constraints, fragile Selenium scripts, prompt length overhead, and cloud delivery gaps. To address these, the next steps include implementing persistent chat history, reinforcing Selenium waits for dynamic web elements, adopting schema-first compact prompts, and enabling direct cloud-based report delivery. These actions extend the delivered system and keep it aligned with enterprise deployment needs.

7 References

Automation, D. (2023, September 20). Selenium vs. Other QA Tools: A Comparative Analysis. Medium.

<https://medium.com/@devopsautomation93/selenium-vs-other-qa-tools-a-comparative-analysis-201bc1d66a9>

Bug bytes web. (n.d.). <https://bugbytes.io/posts/vector-databases-pgvector-and-langchain/>

Langchain-Ai. (n.d.). GitHub - langchain-ai/rag-from-scratch. GitHub.

<https://github.com/langchain-ai/rag-from-scratch>

Pankaj. (2024, November 19). Building a Local RAG Agent with LLaMA3 and LangChain.

Medium.

https://medium.com/@pankaj_pandey/building-a-local-rag-agent-with-llama3-and-langchain-6f041655eb83

PostgreSQL Extensions: Turning PostgreSQL Into a Vector Database With pgvector. (2025, January 10). Timescale.

<https://www.timescale.com/learn/postgresql-extensions-pgvector>

Sewrathan, A., Gichohi, S., Arye, M., & Ishimwe, H. (2025, April 23). PostgreSQL as a Vector Database: A PGVector tutorial. Timescale Blog.

<https://www.timescale.com/blog/postgresql-as-a-vector-database-using-pgvector>

Test case generation, automation and execution using GENAI. (n.d.). Specialist Software Testing and Quality Engineering Consultancy | Prolifics Testing.

<https://www.prolifics-testing.com/news/genai-test-automation-concept>

Vileikis, L. (2025, April 24). A Beginner's Guide to Vector Search Using pgvector. DbVisualizer.

<https://www.dbvis.com/thetable/a-beginners-guide-to-vector-search-using-pgvector/>

Wikipedia contributors. (2025a, April 18). Katalon Studio. Wikipedia.

https://en.wikipedia.org/wiki/Katalon_Studio

Wikipedia contributors. (2025b, May 5). PostgreSQL. Wikipedia.

<https://en.wikipedia.org/wiki/PostgreSQL>

Wikipedia contributors. (2025c, May 8). PostgreSQL. Wikipedia.

<https://en.wikipedia.org/wiki/PostgreSQL>

Ws. (2025, May 6). Building a local AI chatbot using Streamlit, LangChain, and Ollama. Medium.

<https://medium.com/@Shamimw/building-a-local-ai-chatbot-using-streamlit-langchain-and-ollama-484b82083988>

8 Appendix

Appendix A

Requirements Traceability Matrix (RTM)

Requirements		Description	Types(s) (FR/NFR)	Source	Status
Automated Test Execution & Logging	1.1	Implement 3 core test scenarios/cases: Print Run Card, Lot Trigger Immediate Hold and Lot Release	FR	Project Supervisor	Completed
	1.2	Save automated test execution results, including screenshots for actual vs. expected comparisons, pass/fail status and error details, in Word document	FR	Project Supervisor	Completed
	1.3	Support combined execution of multiple test cases	FR	Project Supervisor	Completed
	1.4	Prevent redundant action e.g. relogin to CamStar webpage during automated test execution	FR	Project Supervisor	Completed
	1.5	If a module fails, continue executing next test case without crashing	FR	Project Supervisor	Completed
Chatbot Interface	2.1	A chatbot interface for users to request test case(s) and confirm execution.	FR	Project Supervisor	Completed
	2.2	Accept user input via the chatbot for test case variables e.g. selection ID and feed into test scripts	FR	Project Supervisor	Completed
	2.3	Send live test execution progress updates to the chatbot	FR	Project Supervisor	Completed
	2.4	Provide the final report link or location within the chatbot interface	FR	Project Supervisor	Completed
	2.5	Provide the generated script link or location within the chatbot interface	FR	Project Supervisor	Completed
Generative AI for Script Generation	3.1	Generate initial test scripts using Selenium IDE (i.e. record basic flows)	FR	Project Supervisor	Completed
	3.2	Recognise key inputs e.g. selection ID as variables	FR	Project Supervisor	Completed
	3.3	Use an LLM to fine-tune and optimize the Selenium IDE-generated code	FR	Project Supervisor	Completed
	3.4	Save optimized scripts into the repository and provide the file path within the chatbot interface	FR	Project Supervisor	Completed

	3.5	Remember past interactions	FR	Project Supervisor	Completed
Prompt Engineering	4.1	Write 3 core test plans	FR	Project Supervisor	Completed
	4.2	Refine prompt for accurate test case retrieval and request for required test case variables e.g. selection ID	FR	Project Supervisor	Completed
	4.3	Generate test scripts via Selenium IDE	FR	Project Supervisor	Completed
	4.4	Use AI Forge to remove login steps, add screenshots and retrieve variables	FR	Project Supervisor	Completed
Documentation	5.1	Maintain clear documentation using Confluence	NFR	Project Supervisor	Completed
	5.2	Version Control using GitLab	NFR	Project Supervisor	Not Applicable

Appendix B

Save Chat History

Save as json

How to Use the Chatbot

Test Case Execution:

1. Start a Test:

- Type a short description of the **test case** you want to run.
- The chatbot will suggest a matching test.
- If the suggestion is correct, reply **yes** or **y**.
- If it's not correct, simply re-enter your description.
- You can run **one test** or **combine several tests** in a single run.

2. Provide Lot Information

- The chatbot will ask for any **required details** (e.g., Selection Id (Lot Number)).
- Enter the information exactly as shown on Camstar.
- Choose `debug=True` if you want screenshots of every step.
- Choose `debug=False` if you only want final result screenshot.
- Make sure the details are all correct before submitting.

3. Run the Test

- Once you confirm, the chatbot will automatically run the selected test(s) on Camstar.
- You don't need to manually click anything in Camstar.

4. Monitor Progress

Rerun

CamStar Portal Auto Testing

How can I help you today?

test lot on hold then release

Does this look like the test cases you had in mind?

Test Case 1: Lot Trigger Immediate Hold
Description 1: This test is designed to verify the system's ability to immediately place a lot on hold, ensuring that lot status is updated accordingly.
Parameters 1: selection_id, test_comment, debug

Test Case 2: Lot Release
Description 2: This test is to verify the functionality that release a lot, ensuring that the status is updated accordingly.
Parameters 2: selection_id, test_comment, debug

If this fits, tell me and we'll sort out any remaining details. If not, describe what's different or ask to see other options.

yes

To run these together, I still need: selection_id, test_comment, debug
You can share them in plain language or key=value form.

KLPC14CCR000, TEST_123, true

The test case ['Lot Trigger Immediate Hold', 'Lot Release'] will be executed with parameters:
{'selection_id': 'KLPC14CCR000', 'test_comment': 'TEST_123', 'debug': 'true'}

Executing...

Report generated successfully!

Copy and paste the file path to your file explorer:

\\MYKUL01CMMSDWDV\\OutputDocuments\\NnCtFZeX.docx

Appendix C

Save Chat History

Save as json

How to Use the Chatbot

Test Case Execution:

1. Start a Test:

- Type a short description of the test case you want to run.
- The chatbot will suggest a matching test.
- If the suggestion is correct, reply yes or y.
- If it's not correct, simply re-enter your description.
- You can run one test or combine several tests in a single run.

2. Provide Lot Information

- The chatbot will ask for any required details (e.g., Selection Id (Lot Number)).
- Enter the information exactly as shown on Camstar.
- Choose debug=True if you want screenshots of every step.
- Choose debug=False if you only want final result screenshot.
- Make sure the details are all correct before submitting.

3. Run the Test

- Once you confirm, the chatbot will automatically run the selected test(s) on Camstar.
- You don't need to manually click anything in Camstar.

4. Monitor Progress

- You'll see live updates in the chat while the test is running.
- Pass = Test ran successfully.
- Fail = Something went wrong during the test.
- You can watch which test is running and its status in real time.

5. View & Download Report

- When the test finishes, a Word report will be generated automatically.
- The report include:
 - Test case name
 - Pass/Fail status
 - Screenshots (based on your debug setting)
- Copy the URL provided in the chat and paste it into your browser or file explorer to view or

CamStar Portal Auto Testing

How can I help you today?

Uploaded file: test_lotprinting.py

```
import time
import traceback
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.keys import Keys

class TestLotprinting():

    def test_lotprinting(web_obj, session_id, lot_id, debug=True):
        success = False
        screenshot_lst = []
        driver = web_obj.driver

        try:
            adhoc_link = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#tabContainerContent"))
            screenshot_lst.append(web_obj.element_action(adhoc_link, "click", debug))
            printing_link = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#print"))
            screenshot_lst.append(web_obj.element_action(printing_link, "click", debug))

            web_obj.driver.switch_to_frame_by_number(1)

            imbt_button = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#imbt"))
            screenshot_lst.append(web_obj.element_action(imbt_button, "click", debug))

            jstree_leaf = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#jstree"))
            screenshot_lst.append(web_obj.element_action(jstree_leaf, "click", debug))

            td_element = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#td"))
            screenshot_lst.append(web_obj.element_action(td_element, "click", debug))

            parameter_input = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#parameter"))
            screenshot_lst.append(web_obj.element_action(parameter_input, "send_keys", "test"))

            parameter_input_enter = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#parameterEnter"))
            screenshot_lst.append(web_obj.element_action(parameter_input_enter, "enter"))

            print_action_button = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#printAction"))
            screenshot_lst.append(web_obj.element_action(print_action_button, "click", debug))

            web_obj.driver.switch_to_default_content()

            close_tab_button = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#closeTab"))
            screenshot_lst.append(web_obj.element_action(close_tab_button, "click", debug))

            success = True
            return success, screenshot_lst
        except Exception:
            time.sleep(1)
            screenshot_lst.append(web_obj.driver_save_screenshot("lot_printing_success.png"))
            try:
                close_tab_button = WebDriverWait(driver, 10).until(EC.element_to_be_clickable(By.CSS_SELECTOR, "#tabContainerContent"))
                close_tab_button.click()
                traceback_string = traceback.format_exc()
                return success, screenshot_lst, traceback_string
            except Exception:
                traceback_string = traceback.format_exc()
                return success, screenshot_lst, traceback_string

```

Script generated successfully!

Copy the file path below and paste in your file explorer:

\\MYKUL01CMMSDWDV\GeneratedScripts\20251106_192008_test_lotprinting.py

Appendix D

User Interface Sample Code

```
"""
CamStar Portal Auto Testing: Streamlit chat app for test planning,
execution, and script generation.

- Loads prompts and guides from disk via environment variables (.env).
- Uses AWS Bedrock via boto3 to stream LLM responses (Anthropic Claude).
- Integrates with a vector DB service for context retrieval.
- Executes tests via a remote HTTP service and streams execution logs.
- Generates and returns Word reports and rewritten scripts.

"""

from dotenv import load_dotenv

load_dotenv()

# --- Helpers ---
def render_history() -> None:
    """
    Render chat history from Streamlit session state.

    Side Effects:
        Writes chat messages to the Streamlit UI.
    """
    for m in ss.ui_messages:
        with st.chat_message(m["role"]):
            st.markdown(m["content"])

def download_generated_script(message):
    """
    Extract test script and parameters from a test case payload.

    Supports two shapes:
    1) Flat: {"test_script": <str>, "parameters": {...}}
    2) Plan: {"plan": [{"test_script": <str>, "parameters": {...}}, ...]}

    Args:
        test_case: Test case payload returned by the LLM.

    Returns:
        A tuple of (test_script, params). The script is a string or a
        list of strings.
        The params is the associated parameter dict.

    Raises:
        KeyError: If required keys are missing in both shapes.
    """
    pattern = r"```\python\s*(.*?)```"
```

```

match = re.findall(pattern, message, re.DOTALL)
file_names_lst = [f.name for f in ss.attach_files]
dir_list = []
final_msg = None

if match:
    for i, script in enumerate(match):
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        script_file_name = f"{timestamp}_{file_names_lst[i]}"
        filename = os.path.join(os.getenv("response_share_path"),
script_file_name)
        dir_list.append(filename)
        file_dir = os.path.join(os.getenv("response_local_path"),
script_file_name)
        file = open(file_dir, "w")
        file.write(script)
        file.close()

if len(dir_list) > 0:
    plural = "s" if len(dir_list) > 1 else ""
    final_msg = (
        "Script generated successfully! \n"
        f"Copy the file path{plural} below and paste in your file
explorer:\n"
    )
    for dir in dir_list:
        final_msg += f"```\n{dir}\n```"
return final_msg

# --- Main Chat Area ---
st.title("CamStar Portal Auto Testing")
render_history()

if ss.initial_prompt:
    add_messages("How can I help you today?", only_ui=True,
check_repeat=True)
    ss.initial_prompt = False

if prompt := st.chat_input("Say something", accept_file="multiple"):
    if prompt.text and not prompt.files:
        add_messages(prompt.text, is_user=True, check_repeat=True)
    if prompt.files:
        ss.attach_files = prompt.files
        plural = "s" if len(ss.attach_files) > 1 else ""
        file_names_msg = f"Uploaded file{plural}: {", ".join(f.name for f
in ss.attach_files)}"
        file_names_lst = [f.name for f in ss.attach_files]
        content = [f.read().decode("utf-8") for f in ss.attach_files]
        user_file_input = [f"Uploaded file: {file_names_lst[i]}\n\n{f}"
for i, f in enumerate(content)]
        add_messages(file_names_msg + " \n" + prompt.text, is_user=True,
only_ui=True, check_repeat=True)
        add_messages(f"{'\n\n'.join(user_file_input)}\n\n{prompt.text}",


```

```
    is_user=True, only_llm=True, check_repeat=True)

with st.spinner("Thinking..."):
    reply = chat_model(ss.llm_messages)
    add_messages(reply, show_after_rerun=True)
    if ss.attach_files:
        script_generation_msg = download_generated_script(reply)
        if script_generation_msg:
            add_messages(script_generation_msg, only_ui=True)
st.rerun()
```