

# Agentic AI-Based Automated Test Case Generation and Execution

Srushti Haware<sup>1</sup>, Sakshi Hiremath<sup>2</sup>, Sharayu Sanap<sup>3</sup>, Janhavi Inamdar<sup>4</sup>, Prof. Shweta Shah<sup>5</sup>

*Dept. of Computer Engineering, Pune Institute of Computer Technology, Pune, India*

\*\*\*

**Abstract** - Software systems today are growing faster and getting more complex than traditional testing workflows can handle. Converting requirement documents and UI designs into executable test cases still demands heavy manual effort, and the resulting coverage is often incomplete. While LLMs have opened new possibilities for automating parts of this work, most systems built on them treat test generation as a one-shot problem—fire a prompt, get scenarios, done. There is no coordination across stages, no real handling of multimodal inputs, and no mechanism for the system to respond when a generated test actually fails. This paper describes an agentic AI framework built to address these gaps. The system chains four specialized agents into a pipeline: one handles multimodal input processing, one generates Gherkin-format test scenarios, one converts those scenarios into runnable Playwright/Pytest scripts, and a human-in-the-loop validation layer sits between generation and execution to catch problems early. When tests fail or reviewers reject scenarios, the system regenerates rather than moving on. Implemented using a hybrid local-cloud inference setup and evaluated on an employee shift management application, results showed meaningful gains in test coverage and a high proportion of directly executable scripts, with low manual effort overall.

**Key Words:** Agentic AI, Automated Software Testing, Test Case Generation, Large Language Models, Multimodal Learning, Gherkin, Test Automation, Playwright, Human-in-the-Loop, Software Testing Framework

## 1. INTRODUCTION

Testing is one of those phases in software development that everyone agrees is critical, yet it consistently receives less attention than it deserves—partly because it is tedious, and partly because automating it well is genuinely hard. As systems grow larger and release cycles compress, manual testing becomes a bottleneck. Rule-based automation helps, but it requires significant upfront effort and tends to break when requirements change. The result is incomplete coverage, late-stage bugs, and expensive rework.

Early AI approaches to this problem applied machine learning to generate test cases or predict defect-prone code [1]–[4]. These methods worked reasonably well on structured inputs but struggled with unstructured requirements or dynamic interfaces. The arrival of Large Language Models changed what was possible—LLMs can read natural language specifications and generate plausible test scenarios without task-specific training [5]. Liu *et al.*, for instance, showed that GPT-based models could perform zero-shot GUI testing on mobile apps, handling interactions that previously required human intuition [6]. Unit test

generation has also seen promising results from LLM-based approaches [7].

The problem is that using an LLM as a standalone step does not actually solve testing automation—it just moves the bottleneck. Without any coordination between stages, generated outputs pile up redundancies, contradict each other, and often cannot run without manual cleanup. Combining a requirements PDF with UI screenshots in the same analysis pass is rarely supported. And if a test fails during execution, there is nothing to close that loop: the failure sits in a report and a human has to diagnose it.

Agentic AI addresses these gaps by structuring the work across multiple coordinated agents, each with a well-defined role [8], [9]. Rather than one LLM doing everything, the agents divide responsibilities—one parses inputs, another reasons about what scenarios are needed, a third writes the scripts. This decomposition makes it much easier to introduce feedback at each handoff [10]–[12], and in the testing domain specifically, it allows the pipeline to validate, execute, and refine generated tests rather than just producing them.

The contributions of this work are as follows:

- A multi-agent pipeline for automated software testing, covering input processing, scenario reasoning, script generation, and execution within a single coordinated workflow.
- A multimodal input processing approach that jointly handles requirement documents and UI screenshots rather than treating them separately.
- A traceable transformation path from natural language requirements through Gherkin scenarios to executable Python scripts, with identifiers linking each artifact back to its origin.
- A human-in-the-loop validation layer with a feedback mechanism that triggers regeneration of rejected or failed test cases.
- An evaluation on a real application comparing results against manual testing and single-step LLM generation baselines.

## 2. LITERATURE SURVEY

Research on AI-assisted software testing has a reasonably long history at this point. Early work applied supervised learning to fault prediction and test case prioritization. Results were promising, but these methods depended heavily on structured and labeled input data [1]–[4]—which meant they struggled the moment requirements were expressed informally or UIs changed without warning.

LLMs shifted the conversation significantly. Wang *et al.* provided a broad survey of LLM-based testing techniques, showing that models pretrained on code and natural language can interpret specifications and produce test cases without fine-tuning [5]. Liu *et al.*'s zero-shot mobile GUI testing work was particularly illustrative—GPT-3 could simulate interactions on apps it had never seen during training [6]. Unit test generation with LLMs has also shown consistent gains [7]. But the pattern across all this work is the same: LLMs handle understanding and generation well. Orchestration is where they fall short—managing multi-step flows, recovering from failures, and keeping up with changing requirements.

Agentic AI has emerged as the architectural response. The premise is straightforward: decompose complex tasks into components handled by specialized agents, then coordinate them [8], [9]. In testing, this has shown up in scriptless automation frameworks [11], [12] and in test-driven development pipelines [10]. LLM-driven unit test generation using agentic setups is also gaining traction [7]. The gaps in current work, though, are real. Handling both text and images as inputs at the same time is largely unsupported—most frameworks pick one or the other. Feedback based on actual execution outcomes is even rarer; a test fails, a report gets written, and that is the end of it. Scalability, traceability across stages, and robustness in dynamic environments also remain open problems [13]–[16]. These are the gaps this work sets out to address.

### 3. SYSTEM DESIGN AND ARCHITECTURE

The system is built as a multi-agent, modular pipeline where each agent is responsible for exactly one stage of the workflow. Fig -1 shows the overall structure.

The pipeline begins with requirement documents (PDF or DOCX) and UI screenshots as inputs. The Analysis Agent processes both together and encodes the extracted information in a JSON format that captures features, workflow sequences, constraints, and anticipated failure modes. Keeping everything in a standardized intermediate format means each downstream agent can be developed and debugged independently.

The Test Generation Agent takes this JSON and produces BDD-style Gherkin scenarios. Generation is constrained so each scenario covers exactly one logical outcome, avoiding the combinatorial explosion that happens when an LLM is given free rein over conditions. Scenarios are also tagged as *NEW*, *UPDATED*, or *UNCHANGED*, allowing the system to handle evolving requirements incrementally without discarding previously validated work.

Before any script is written, generated scenarios go through a HITL validation layer. The reviewer can accept a scenario or send it back. The generator rarely produces outright wrong scenarios—the failures tend to be subtle, the kind that would pass a quick reading but break at execution time. The review step catches those.

The Script Generation Agent converts accepted Gherkin scenarios into Python test scripts using Playwright and Pytest. Each scenario maps to a uniquely named test function, keeping requirements and executable tests linked. Scripts cover both frontend interactions (via Playwright) and backend state checks (via API calls).

The Execution Agent runs the generated scripts and routes failure information back into the pipeline. A failed test does not just produce a report—it triggers regeneration. A state management component handles versioning with hash-based change detection, so reruns only regenerate what has actually changed.

### 4. METHODOLOGY

#### A. Analysis Agent

The Analysis Agent sits at the entry point of the pipeline. It takes multimodal inputs—PDF or DOCX files for textual requirements, and screenshots for visual context—and converts them into a structured JSON representation. Text goes through a cloud-based LLM (Groq API); screenshots go through a locally hosted vision model via Ollama. This split is deliberate: cloud inference gives stronger language understanding, while running vision locally means screenshots from proprietary UIs never leave the machine. The output JSON captures features, technical rules, workflow sequences, and failure scenarios. Forcing the analysis into this format surfaces things that requirements documents leave implicit—what should happen at a boundary value, what the system state looks like after an error that no one bothered to document.

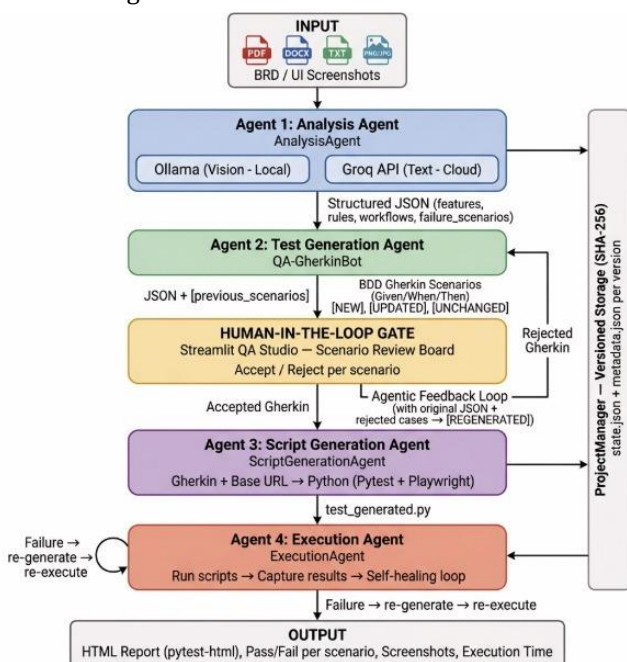


Fig -1: Agentic AI-based automated test generation and execution pipeline.

## B. Test Generation Agent

The Test Generation Agent reads the structured JSON and produces Gherkin scenarios. The prompt includes constraints arrived at through iteration: one outcome per scenario, no cross-product combinations of conditions, and explicit coverage of negative and edge cases alongside the functional ones. Without those constraints, early versions produced technically valid scenarios that were almost entirely happy-path.

Version-aware generation works by including previously accepted scenarios in the prompt context. The agent labels each new output as new, updated, or unchanged, letting the reviewer focus on what has actually shifted rather than re-reviewing everything.

## C. Human-in-the-Loop Validation

The HITL layer is a Streamlit interface. Each scenario is shown alongside the script it would produce—so the reviewer sees not just what the test checks, but how it plans to check it. Accepted scenarios move to script generation. Rejected ones go back to the Test Generation Agent; if the reviewer left a comment explaining why, that comment is included in the regeneration prompt.

Showing the script alongside the scenario was a design decision made after early testing showed reviewers missing execution-level issues that looked fine in Gherkin. The two-panel view solved that.

## D. Script Generation Agent

The Script Generation Agent converts validated Gherkin scenarios into executable Python test scripts. Each Given/When/Then step maps to Playwright actions and assertions. The prompt enforces consistent patterns for navigation, form interaction, and response validation, while still leaving room for scenario-specific behavior.

After generation, a post-processing step scrubs the output—stripping markdown artifacts, fixing indentation, and catching syntax issues. Scripts that fail this check are flagged before they ever reach the executor.

## E. Execution and Feedback Mechanism

The Execution Agent calls Pytest via subprocess, capturing stdout, stderr, and the structured JSON report from the pytest-html plugin. For each test function it tracks pass/fail, error traces, and execution time.

When tests fail, the error traces get parsed and summarized before being sent back to the Script Generation Agent as additional context. Failures tended to fall into two buckets: timing issues (element not ready when the script tried to interact with it) and logic errors in the generated script itself. The feedback mechanism handled both—for timing issues, the agent would add explicit waits, while logic errors sometimes needed a full regeneration of the scenario.

## F. State Management and Versioning

Every pipeline run produces a versioned snapshot: the input hash, the scenario JSON, the accepted subset, and

execution results. On subsequent runs, the system hashes inputs at the feature level and compares against the previous snapshot to identify what needs regeneration. For large requirement sets these matters—re-running the full pipeline from scratch every time would be prohibitively slow.

## 5. IMPLEMENTATION DETAILS

The framework is implemented in Python. Agents are independent modules sharing a data directory, which makes it easy to restart the pipeline from any stage without rerunning what already worked.

### A. Pipeline Coordination

A central coordinator manages stage transitions and handles errors between agents. Data is passed via JSON files on disk rather than in-memory between agents. The practical reason: at any point during development or debugging, the intermediate file can be opened to see exactly what one agent handed to the next. It also made it straightforward to re-run a single stage in isolation.

### B. Multimodal Processing

Text-based requirement documents go to the Groq API for structured extraction. Screenshots run through Ollama with a vision-capable model locally. One issue encountered early: screenshots from different UI frameworks—React, Angular, plain HTML—can look very different even when they represent the same interaction. The model would sometimes misidentify elements or miss interactive components entirely. This was addressed by including explicit prompts about element identification, rather than letting the model infer semantics from visual appearance alone.

### C. Technology Stack

Text-based LLM inference runs through the Groq API. Vision inference runs locally via Ollama. Playwright handles browser automation; Pytest is the test runner; pytest-html generates structured execution reports. All model parameters, API endpoints, and timeout values live in a single YAML configuration file—swapping models during experimentation was just a matter of editing one line.

### D. Script Generation Details

The script generation prompt includes the Gherkin scenario, the application's base URL, and a set of Playwright conventions standardized over several iterations—for example, always calling `page.wait_for_selector` before touching a dynamic element, and always checking API responses with `page.expect_response`. Getting these conventions right took time. Early scripts passed local tests but broke in CI because they assumed elements would be ready faster than they actually were in a headless environment.

Test function names are derived deterministically from the scenario title, so traceability holds even when scripts are regenerated.

### E. Execution Agent

The Execution Agent runs Pytest as an external subprocess rather than invoking it programmatically. Subprocess invocation gives cleaner exit codes and avoids state leaking between test runs—something that caused intermittent, hard-to-diagnose failures when the programmatic approach was tried early on.

After each run, results are parsed into a summary (totals, pass/fail counts, per-test details) and written to the versioned state directory. The Streamlit interface also reflects these results so reviewers can see execution outcomes without leaving the validation UI.

### F. Human-in-the-Loop Interface

The Streamlit interface shows each scenario as a card: Gherkin on the left, the corresponding script on the right. Reviewers can accept, reject, or defer. Rejecting a scenario opens a comment field; the comment travels with the scenario back into regeneration. Once decisions are submitted, rejected scenarios queue immediately for regeneration while accepted ones move to the script generation stage.

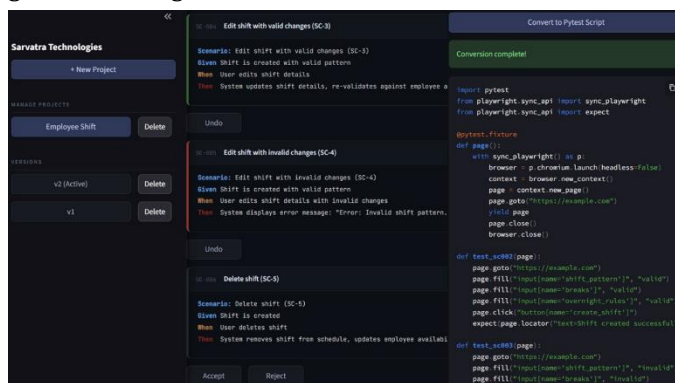


Fig -2: Human-in-the-loop interface showing scenario validation and corresponding generated test scripts.

## 6. RESULTS AND DISCUSSION

### A. Experimental Setup

The framework was evaluated on an employee shift management application—a system with non-trivial business logic around role-based access, schedule validation, and notification triggers. Input consisted of a requirements document (DOCX) and eight UI screenshots covering the main workflows. The system ran on a standard development machine using Groq API for text inference and LLaVA locally for vision.

### B. Test Case Generation Analysis

The Analysis Agent extracted structured representations from all inputs, including several constraints that were only implicit in the screenshots—validation rules for shift overlaps and role permissions that the text document left underspecified. The Test Generation Agent then produced scenarios spanning four categories: functional, negative, boundary, and edge cases.

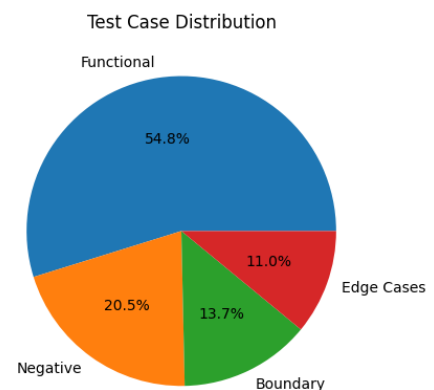


Fig -3: Distribution of generated test cases across different categories.

As Fig -3 shows, functional scenarios made up roughly 54.8% of the total—expected, given the structure of the prompt. What was more notable was the proportion of negative (20.5%) and boundary (13.7%) cases generated without those categories being explicitly requested. They emerged from the constraints the Analysis Agent had encoded. Edge cases came in at 11%. These categories tend to get skipped or underrepresented in manual planning, so having them appear organically was a meaningful result.

### C. Human-in-the-Loop Validation Results

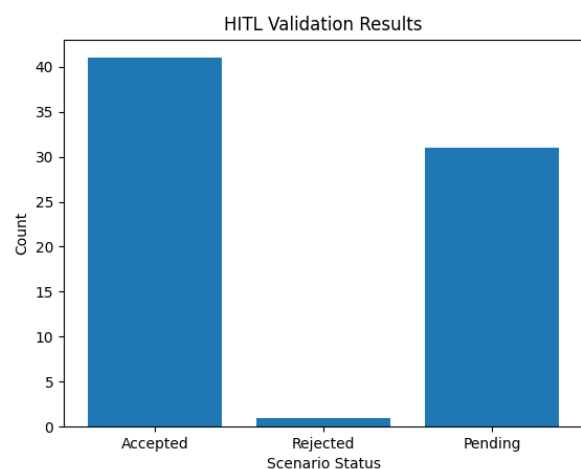


Fig -4: Validation outcomes of generated test scenarios using the HITL interface.

Fig -4 shows how scenarios were categorized during review. Most were accepted outright. Rejections clustered around edge-case scenarios where the agent had filled in gaps with assumptions that did not match how the application actually behaved—shifts crossing midnight, for instance, or permission checks on recently deactivated accounts. The side-by-side script view helped reviewers catch these quickly; without it, the Gherkin alone often looked reasonable. Scenarios that were rejected with reviewer comments almost always passed on the second generation.

### D. Script Generation Results

All accepted Gherkin scenarios were successfully converted to executable Python scripts. Each test function carried a unique identifier traceable back to its source scenario. Scripts covered both frontend interactions and backend API validation rather than just simulating UI clicks.

**Table -1:** Evaluation of Generated Test Scripts

Metric	Observation	Inference
Test Case Diversity	High	Covers multiple categories
Script Generation Success	Consistently high	Reliable automation output
Manual Intervention Required	Low	Efficient automation
Traceability	Maintained	End-to-end mapping ensured

Table -1 summarizes these results. The post-processing step caught minor syntax issues in roughly 12% of scripts before they reached the executor—mostly markdown code fences left in the output, and occasional indentation problems in nested assertions.

### E. Execution Results

Most scripts ran successfully on the first attempt. The failures that did occur split into two categories: timing-sensitive scenarios where Playwright tried to interact with elements before the page had fully loaded, and a handful of API validation steps where the expected response format had quietly changed. The feedback loop caught both types and triggered regeneration with the relevant error context attached to the prompt. Timing failures were resolved by the agent adding explicit waits; the API failures required regenerating the assertion logic.

The HTML reports produced by pytest-html were genuinely helpful during this phase. Having full error traces in line with the test description meant failures could be diagnosed without re-running the framework interactively.

### F. Comparative Analysis

**Table -2:** Comparison with Existing Approaches

Approach	Automation Level	Test Coverage	Feedback Loop
Manual Testing	Low	Limited	No
LLM-based Generation	Medium	Moderate	No
Proposed Framework	High	Comprehensive	Yes

Table -2 puts the system in context against the two most obvious alternatives. Manual testing is precise but slow, and scales poorly as requirements grow. Single-step LLM generation speeds things up but produces outputs that still need significant human cleanup and offer no recovery path when tests fail. The proposed framework is the only one of the three that treats execution outcomes as inputs to the generation process rather than just final results.

### G. Discussion

The core bet of this work—that structuring test generation across specialized agents rather than running a single LLM would produce better results—held up across the evaluation. The framework handled multimodal inputs in cases where the screenshots contained UI state that the requirements document had not described in text. The HITL layer intercepted failures that would have been invisible to automated checks. And the feedback loop, while not fully autonomous, reduced the manual debugging effort substantially.

That said, real limits were encountered. The quality of the structured JSON representation depends on how clearly the input requirements are written—vague or contradictory requirements produce vague JSON, and those problems compound as the data moves through the pipeline. The feedback loop also cannot resolve infrastructure issues on its own; when failures were caused by environment-specific configuration (proxy settings, certificate issues in CI), the agent flagged them correctly but could not fix them. Scale testing was also not performed—large requirement sets would put pressure on LLM context windows in ways not yet characterized.

Future work should prioritize two areas: making the execution feedback more autonomous, and improving the vision model's robustness across a wider range of UI frameworks and interaction patterns.

### 7. CONCLUSION AND FUTURE WORK

This paper presented an agentic AI framework for automated test case generation and execution. The system chains four specialized agents—covering input analysis, scenario generation, script production, and execution feedback—into a pipeline that takes multimodal inputs and

produces runnable Pytest/Playwright scripts with minimal manual effort. A HITL validation layer and closed-loop feedback mechanism allow iterative improvement without restarting from scratch.

Evaluation on a real shift management application showed strong test coverage, a high rate of script execution success, and meaningful reduction in manual effort compared to both manual testing and single-step LLM generation. The agentic design proved particularly valuable for the cases that single-step generation handles poorly: combined text-and-image inputs, incremental requirement updates, and recovery from execution failures.

Three directions for future work stand out. First, making the execution feedback more autonomous—right now environment-specific failures are caught but not automatically fixed. Second, extending the vision model's capability to handle more complex UI patterns; degraded performance was observed on applications with heavy dynamic rendering and context-dependent element states. Third, wiring the framework into CI pipelines so that test regeneration triggers automatically on requirement or code changes, removing the need for manual pipeline invocation.

## REFERENCES

- [1] M. U. Shafique and M. A. Khan, "A survey on the impact of AI in software testing," *Journal of Software Engineering Research and Development*, vol. 10, no. 1, 2022.
- [2] Y. Zhang and X. Zhao, "AI-based testing techniques: A systematic review and future directions," *ACM Computing Surveys*, vol. 54, no. 3, 2021.
- [3] S. Dey and A. Gupta, "Enhancing software testing with artificial intelligence: A review," *Software Quality Journal*, vol. 28, no. 4, 2020.
- [4] Y. Liu and J. Wang, "Automated test case generation using deep learning techniques," *Journal of Systems and Software*, vol. 203, 2023.
- [5] S. Wang, M. Zhou, L. Zhao et al., "Software testing with large language models: Survey, landscape, and vision," *arXiv preprint arXiv:2307.07221*, 2023.
- [6] X. Liu, T. Wang, Y. Chen, J. Zhang, and Y. Li, "Chatting with GPT-3 for zero-shot human-like mobile automated GUI testing," *arXiv preprint arXiv:2305.09434*, 2023.
- [7] "LLM-driven unit test case generation using agentic AI," *IRO Journals*, 2024.
- [8] "Agentic AI: Autonomous intelligence for complex goals—A comprehensive survey," *IEEE*, 2024.
- [9] "The rise of agentic AI: A review of definitions, frameworks, architectures, applications, evaluation metrics, and challenges," *Future Internet*, vol. 17, no. 9, 2024.
- [10] S. Haware et al., "Retail resilience engine: An agentic AI framework for building reliable retail systems with test-driven development approach," in *IEEE Conference*, 2024.
- [11] "Agentic AI-based test automation: A strategic leap forward for enterprises," *ResearchGate*, 2024.
- [12] "AI agentic scriptless automation in software testing," *International Journal of Computer Trends and Technology*, vol. 72, no. 9, 2023.
- [13] P. Gokhale and S. Gokhale, "AI in software testing: A comprehensive review," *Journal of Computer Languages, Systems & Structures*, vol. 62, 2021.
- [14] H. Kaur and S. Singh, "The role of AI in software testing: Current trends and future directions," *International Journal of Software Engineering and Applications*, vol. 13, no. 1, 2022.
- [15] T. Menzies and M. Pezze, "AI for software engineering: A roadmap," *IEEE Software*, vol. 37, no. 5, 2020.
- [16] J. Rojas and J. A. Pino, "The future of software testing: AI and machine learning," *Software Testing, Verification and Reliability*, vol. 31, no. 8, 2021.