International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Autonomous Web Application Testing Using Agentic AI

Pruthviraj Pramod Bhoite
Department of Computer Engineering
SVPM's College of Engineering
Baramati, India

Vaibhav Madhav Biradar
Department of Computer Engineering
SVPM's College of Engineering
Baramati, India

Karan Ganesh Jaybhay
Department of Computer Engineering
SVPM's College of Engineering
Baramati, India

Yash Shital Phadtare
Department of Computer Engineering
SVPM's College of Engineering
Baramati, India

Prof. Priyanka N. Kate-Deshmukh
Department of Computer Engineering
SVPM's College of Engineering
Baramati, India

***Abstract*—Web applications change so often that manual testing cannot keep up, and scripted automation breaks whenever the user interface changes. Large language models (LLMs) and the agent systems built on them offer a different approach: software agents that observe an application, decide what to do, act through a browser, and judge the result. This paper reviews how such agents are applied to web application testing. We organize existing work around five stages of the testing lifecycle (requirement analysis, test generation, execution, oracle and defect detection, and reporting), compare single-agent and multi-agent architectures, and examine how agentic testers are evaluated. We collect the challenges that keep recurring: hallucination, non-determinism, cost, weak evaluation practice, and safety. The reviewed work automates isolated stages, and a system that handles the whole lifecycle with little manual scripting has not yet been demonstrated. We therefore propose a five-agent workflow, built on LangGraph, Playwright, a locally hosted open-source model, and PostgreSQL, for end-to-end autonomous testing, and we outline a protocol for evaluating it. The paper presents a design and a literature-based analysis only; it reports no experimental results of its own.**

***Keywords*—agentic AI, large language models, multi-agent systems, web application testing, test automation, Playwright**

I. INTRODUCTION

Modern web applications change all the time. Teams release small updates several times a week, front-end frameworks regenerate the page structure, and one redesign can alter dozens of screens. Testing exists to protect quality, so it has to keep pace with these changes. Manual testing is repetitive and expensive, and the effort grows with every release. Scripted automation removes the repetition, but each script builds in assumptions about element locators, page layout, and the order of steps. When those assumptions stop holding, the script fails even if the application still works. This fragility is a well-documented reason why manual testing is still common [1], [2], [3].

Large language models (LLMs) change what is possible. They can read natural language, interpret page content, and produce structured plans. When an LLM is placed in a loop where it perceives its environment, chooses an action, and observes the result, the combination is called an agent [4], and the resulting approach is often called agentic artificial intelligence (AI). In testing, an agent can be given a goal such as "verify that a user can add an item to the cart and check out" and try it in a real browser without a pre-written script. Because the test is stated as an intent and not as a fixed list of locators, it has a better chance of surviving a change in the interface.

This paper asks four questions. How are LLM-based and agentic techniques used across the testing lifecycle? Which architectures are used to build them? How well do they work, and how is that measured? And what is still unsolved? Our contributions are: (1) an organization of the literature on agentic web testing by lifecycle stage; (2) a comparison of single-agent and multi-agent designs; (3) a discussion of evaluation practice with a consolidated list of challenges; and (4) a statement of the research gap, together with a design for an end-to-end multi-agent testing platform and a protocol for evaluating it.

The rest of the paper is organized as follows. Section II reviews the background and related work. Section III describes the survey method, the lifecycle taxonomy, the proposed system, and the planned evaluation protocol. Section IV presents the findings on lifecycle stages, architectures, and evaluation practice. Section V discusses the challenges, the research gap, and open problems, and Section VI concludes.

II. LITERATURE REVIEW

A. Web Application Testing

Software testing is the activity of running a program in order to find defects [5]. For web applications, testing is usually divided into functional testing (does each feature behave as specified), user-interface (UI) testing (does the page render and respond correctly), and non-functional testing, such as performance, security, accessibility, and cross-browser compatibility. Tests are run either by people who follow written test cases or by scripts driven through a browser-automation library such as Playwright or Selenium. A script replays a fixed sequence of steps reliably, but it has no understanding of what the page means.

B. Why Scripts Become Brittle

A scripted test usually finds elements through identifiers, Cascading Style Sheets (CSS) selectors, or XPath expressions. These locators depend on the page structure and not on what the user is trying to do. A renamed class, a moved button, or a new wrapper element can break a locator. The test then fails for reasons unrelated to product quality, and someone has to repair it [2]. Over time, this maintenance work can outweigh the benefit of automation. In practice, the repetitive manual effort stays, the automation is brittle, and the tools cannot reason about or recover from unexpected situations.

C. Large Language Models and Agents

LLMs trained on large collections of text and code can suggest test ideas, write test code, and describe observed behavior in plain language. A review of 102 studies on LLMs in software testing found that test-case preparation and program repair were the most common uses [6]. In a practitioner questionnaire with 83 responses, 48% of respondents said they already used LLMs in testing activities [7]. An agent goes further than a single prompt. It wraps the model in a loop with perception (a page snapshot or screenshot), memory, planning, and tools [4]. The ReAct pattern alternates reasoning steps with actions, so the model can keep track of a plan, adapt when something unexpected happens, and use feedback from the environment [8].

D. Multi-Agent Systems

A multi-agent system splits a task among specialized agents, for example a planner, an executor, a verifier, and a reporter, which share information through a common state. Agent-based software testing is older than LLMs; a mapping study reports research activity since 1999 [9]. An early framework for web-based systems already used cooperating agents for the test runtime, script generation, execution, and a dashboard [10]. Orchestration frameworks such as LangGraph now describe these systems as a graph of nodes and transitions, which makes control flow, retries, and branching explicit.

E. Related Surveys and Positioning

Three earlier works are close to ours. Wang et al. review LLM-based techniques across software testing as a whole [6]. Wang et al. also present a general framework for LLM-based autonomous agents, which is not specific to testing [4]. Feldt et al. propose a taxonomy of testing agents by level of autonomy, ranging from an assistant that answers questions to an agent that drives the testing process itself [11]. Our review narrows the scope to web applications, organizes the work by lifecycle stage, and adds a comparison of architectures and evaluation practice.

F. Representative Studies

TABLE I lists the studies discussed in this paper and what each contributes. Two patterns stand out. First, each study isolates one stage: surveys map the territory, one benchmark targets execution, one prototype targets exploration and usability, and others cover generation or reporting. Second, evaluation is hard to compare, because each study uses its own applications, test sets, and measures.

TABLE I. REPRESENTATIVE STUDIES

Work	Focus	Main contribution	Limitation for end-to-end testing
Wang et al. [6]	LLMs in software testing	Survey of 102 studies by testing task	Predates mature agent frameworks
Feldt et al. [11]	Concept	Autonomy-level taxonomy of testing agents	Vision paper; limited evaluation
Chevrot et al. [1]	Test execution	Benchmark of 113 test cases; two open-source test agents	Executes given tests; does not generate them
Ye et al. [12]	Exploratory web testing	WebProber; 120-site case study	Usability focus; prototype stage
Sherifi et al. [13]	Generation to reporting	Agent framework for unit tests, execution, reports	Unit testing in Python and Java, not web UI
Almutawa et al. [14]	Web microservices	Multi-agent test design with LLM and non-LLM agents	Scope limited to microservice testing
Haldar et al. [15]	Education	Generative AI for preparatory testing artifacts	Not an autonomous system
Fuad and Sakib [16]	Web exploration	Tester behavior guides automated web testing	Addresses exploration, not the full lifecycle
Singh et al. [17]	Agent tool use	Better tool selection and tool-call formation	Not testing-specific
Wang et al. [4]	General agents	Unified framework for LLM agents	Not testing-specific

Read together, these studies show that the building blocks exist. Models can draft test cases, agents can drive a browser and give verdicts, exploratory agents can surface usability problems, and tool-use research addresses how agents call browsers and other tools. What the studies do not show is a single pipeline in which these blocks are connected, so that the output of one stage becomes the input of the next without a person rewriting it. This observation motivates the lifecycle view adopted in the next section.

G. Synthesis and Positioning

Placing the studies side by side shows where the proposed work sits. The execution benchmark [1] and the exploratory prototype [12] show that agents can operate real web interfaces, but neither generates its own test plan from the application. The generation-to-reporting framework [13] covers more of the lifecycle, but for unit tests and not for browser-level behavior. Studies of generative AI tools [15] and of tester-guided exploration [16] address single steps, and the agent-tooling work [17] supplies a component rather than a testing approach. No reviewed study connects exploration, generation, execution, defect analysis, and reporting under one orchestrated workflow with stored evidence. Our design takes that connection as its goal, and it reuses the findings of these studies where they apply: review of generated tests before execution, step-level evidence for verdicts, and careful tool selection for browser actions.

III. METHODOLOGY

A. Survey Approach

This is a narrative survey, not a systematic review. We found candidate studies through keyword searches on arXiv and the general web, using combinations of the terms “LLM”, “agent”, “multi-agent”, “web testing”, “test generation”, “graphical user interface (GUI) testing”, and “autonomous testing”. We also followed the reference lists of the most relevant papers, in particular those of Chevrot et al. and Sherifi et al., and added a few foundational works that we already knew. A source was kept if it applies language models or agents to web or GUI testing, or if it provides foundations for doing so (agent surveys, reasoning-and-acting methods, web-agent benchmarks, and earlier agent-based testing). A source was excluded if it described no identifiable method or evaluation. The 20 sources cited are mostly from 2022 to 2025, with a few earlier foundational works. The search was not exhaustive, and we do not claim complete coverage. Because the field moves quickly and many relevant reports are preprints, we treat preprint findings as provisional.

B. Lifecycle Taxonomy

We organize the literature along five stages, shown in Fig. 1. Each stage can be handled by a specialized agent. TABLE II summarizes what an agent does at each stage and the typical limitation.

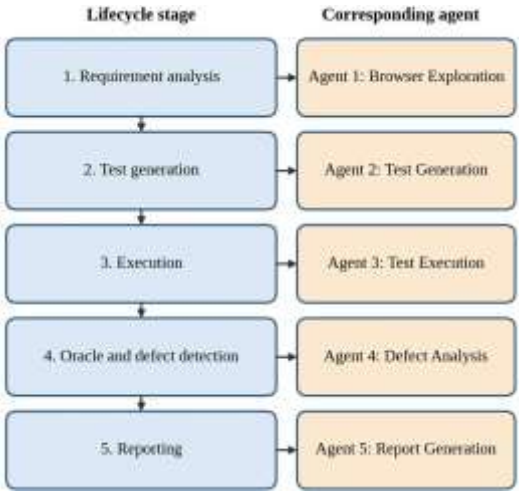


Fig. 1. Testing lifecycle stages and the corresponding agent of the proposed system.

TABLE II. LIFECYCLE STAGES AND THE ROLE OF AGENTS

Stage	What the agent does	Typical limitation
Requirement analysis	Reads specifications, user stories, or the live application to identify features and flows	Ambiguous or incomplete requirements lead to incomplete coverage
Test generation	Produces test cases, data, and scenarios from requirements or exploration	Generated cases may be redundant, incorrect, or untestable
Execution	Operates the browser to carry out steps and recover from unexpected states	Slow, costly, and non-deterministic compared with scripts
Oracle and defect detection	Decides whether observed behavior is correct and flags defects	False positives and missed defects; weak root-cause analysis
Reporting	Summarizes results, evidence, and likely causes	Reports can state unsupported explanations with confidence

C. Proposed System

1) Overview

Motivated by the gap found in this review, we are designing a multi-agent platform that moves on its own from exploring the target application to planning, executing, and reporting tests, without manual scripting in between. Five specialized agents correspond one-to-one to the lifecycle stages of Fig. 1: a Browser Exploration Agent (Agent 1), a Test Generation Agent (Agent 2), a Test Execution Agent (Agent 3), a Defect Analysis Agent (Agent 4), and a Report Generation Agent (Agent 5). A LangGraph orchestrator routes tasks between the agents and keeps the shared state, and a PostgreSQL store provides persistent memory across runs. This paper presents the design only; it does not report experimental results.

2) System Workflow

Fig. 2 shows the end-to-end workflow. The tester supplies a target URL and a test objective and scope. The API layer, a FastAPI endpoint, validates the input and passes it to the agent orchestrator, which routes tasks and holds the workflow state. Agent 1 uses browser automation to inspect the pages and interface elements of the target application. Using this application context, Agent 2 creates test scenarios and steps, and Agent 3 carries them out in the browser through Playwright. A result-validation step then compares the observed behavior with the expected outcome. A passing result goes to Agent 5, which compiles the summary, evidence, and status. A failing result goes to Agent 4, which classifies the failure, attaches the evidence, and estimates the likely cause. The final output is a test report with the passed and failed cases, the detected defects, and the validated results. Generated tests, execution logs, defect records, and reports are saved in the persistent memory. The stored failure evidence and execution outcomes feed back to the orchestrator, so the next planning cycle can use them.

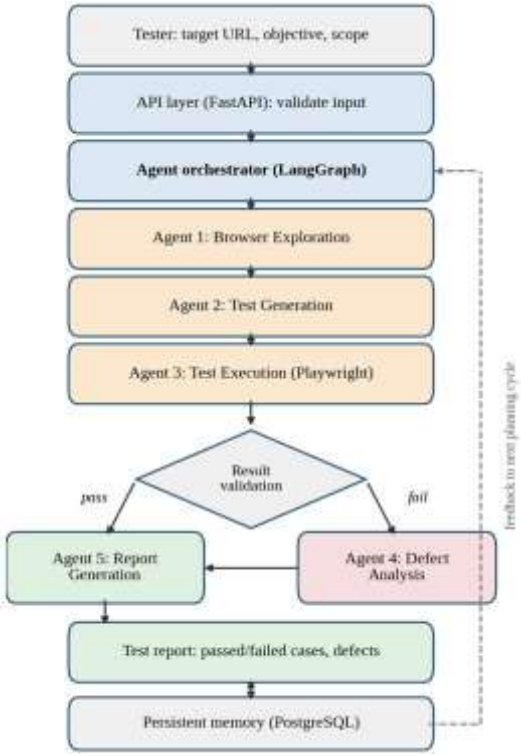


Fig. 2. System workflow of the proposed autonomous web application testing platform: five agents, result validation, persistent memory, and a feedback loop.

3) Architecture

Fig. 3 shows the intended architecture. A web interface connects to a FastAPI service layer, which passes the tester's requests to a LangGraph orchestrator. The orchestrator controls the workflow and holds the shared execution state, and the five agents of Fig. 2 run under it. Ollama serves an open-source LLM for planning, generation, and analysis. Playwright provides browser interaction, test execution, and screenshot capture. PostgreSQL stores projects, test cases, executions, defects, reports, and testing memory. Hosting the model locally keeps page content on the tester's own machine, which addresses the privacy concern raised in Section V-A.

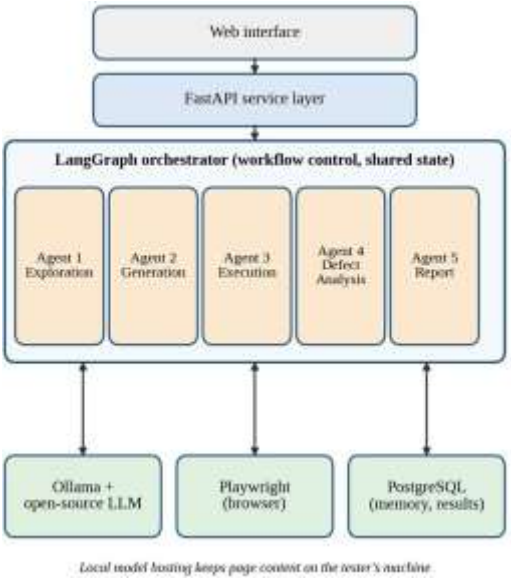


Fig. 3. Proposed multi-agent architecture for autonomous web application testing.

4) Execution Loop and Decision Rules

Each agent follows the loop in Fig. 4. It observes the current page, the test goal, and the execution history. It then plans the next action and its expected result, executes the action through Playwright, verifies the observed behavior against the expected outcome, and updates the stored evidence, results, and memory. Each test ends with one of the four outcomes in TABLE III. Failed steps are retried within configured limits, and unresolved cases are flagged for human review instead of being forced into a verdict.

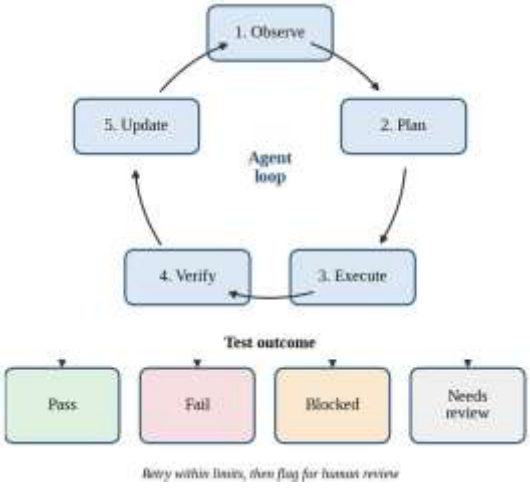


Fig. 4. Agent execution loop and test outcomes.

TABLE III. DECISION RULES FOR TEST OUTCOMES

Outcome	Rule
Pass	Expected result confirmed
Fail	Mismatch supported by evidence
Blocked	A prerequisite prevents execution
Needs review	Evidence is insufficient or conflicting

5) Data Model

TABLE IV shows the proposed PostgreSQL data model. Screenshots and traces are stored as files, and the database keeps only their paths.

TABLE IV. PROPOSED DATA MODEL

Entity	Key fields	Relationship
Projects	project_id, target_url	One project has many test cases
TestCases	case_id, project_id, steps, expected_result	Each case belongs to a project
Executions	execution_id, case_id, status, started_at	A case can have many executions
Defects	defect_id, execution_id, summary, evidence_path	An execution can produce defects
Reports	report_id, project_id, generated_at, report_path	A project can have many reports
TestingMemory	memory_id, project_id, context, observation	A project keeps reusable observations

6) Requirements

The prototype needs a development computer with a multicore processor to run the backend and the browser. Memory and disk requirements depend on the chosen local LLM, and extra storage is needed for model files, screenshots, traces, and reports. The software stack is Python, FastAPI, LangGraph, and PostgreSQL, together with Playwright and its browser binaries, Ollama, and a compatible LLM. A target application URL, a test account, and access to the application complete the setup. Hardware capacity and execution speed still have to be validated with the chosen model.

7) Agent Responsibilities

Each agent has a defined input and output, so that the stages can be tested separately. Agent 1 takes the target URL and scope and returns a map of the application: the pages reached, the forms and controls found on them, and the navigation links between them. Agent 2 takes that map and the tester's objective and returns test cases, each with ordered steps and an expected result, which are stored as TestCases records. Agent 3 takes a stored case, performs its steps in the browser through Playwright, and records the status, a screenshot, and a trace for each execution. The validation step compares the recorded behavior with the expected result and assigns one of the four outcomes in TABLE III. Agent 4 receives only the executions that did not pass, classifies the mismatch, links the evidence, and writes a short statement of the likely cause, which is stored as a Defects record. Agent 5 compiles the passing and failing results, the defects, and their evidence into a report. Because every hand-off goes through stored records, a failed run can be inspected and resumed at the stage where it stopped.

8) Human Oversight and Safety Controls

The design keeps a person in control of the points where an error would be costly. Generated test cases are shown to the tester for approval before they are executed, which follows the finding that generated artifacts need review [15]. Execution is limited to a target environment supplied by the tester, using a test account and disposable data, because an autonomous agent can otherwise submit forms or delete records on a live system. Retries are capped, and a case that remains unresolved is marked as needing review and not forced into a pass or fail verdict. Finally, the trace stored for each execution allows a reviewer to see what the agent did and why it reached its verdict.

D. Planned Evaluation Protocol

The proposed system has not yet been evaluated, so this paper reports no results for it. This subsection states how we intend to evaluate it, using the measures discussed in Section IV-C. First, we plan to use controlled, offline web applications with known passing and failing test cases, in the manner of the benchmark of Chevrot et al. [1], so that each verdict can be compared with a human reference. Second, we plan to report verdict agreement, sensitivity, specificity, and step alignment, together with cost and time measured as model calls, tokens, and duration. Third, to examine adaptation to change, we plan to apply interface modifications, such as renamed identifiers or moved controls, and to compare the repairs that a scripted baseline needs with the cases the agents handle without intervention. Fourth, each test will be run several times with the model version, prompts, and sampling settings fixed and reported, so that repeatability can be measured. A scripted Playwright suite and a single-agent variant of the same system will serve as baselines, so that any benefit of role separation can be isolated from the benefit of using an LLM at all.

IV. RESULTS

This section presents the findings of the literature analysis. Because the paper is a review and the

proposed system is not yet implemented, the results are those reported by the surveyed studies and not measurements of our own.

A. Findings by Lifecycle Stage

1) Requirement analysis and exploration

Before testing begins, someone has to establish what the application is supposed to do. Agents can read written requirements, but they can also explore a running application by following links and interacting with controls to build a picture of its features. Exploration suits applications with little documentation. However, an agent that infers the intended behavior from the interface alone may mistake a defect for a feature. Earlier work on tester-behavior-guided exploration points in the same direction [16].

2) Test generation

Test generation is the most studied use of LLMs in testing [6]. Models turn requirements into test cases, written in natural language or as executable code, and they can propose edge cases that a tester might overlook. The quality depends strongly on the prompt and the context provided. In a case study with generative AI tools, the tools were useful for drafting preparatory testing artifacts, but the output needed review before use [15]. Generated tests still need validation, because a test that looks plausible can assert the wrong behavior.

3) Execution

In agentic execution, the test is described as a goal or as natural-language steps, and the agent works out how to carry them out in the browser. This removes the dependence on fixed locators. Chevrot et al. studied this setting systematically. They built a benchmark of three offline web applications with 113 test cases (62 passing and 51 failing), and they implemented two open-source autonomous test agents that execute steps, check assertions, and give verdicts [1]. Their work raises a key question: can an agent follow a human-written test and reach the same verdict a human would?

4) Oracle and defect detection

An oracle is the means of deciding whether a behavior is correct. Scripted tests rely on explicit assertions. Agents can also judge the page as a whole, for instance by looking at a screenshot and deciding whether it looks right. This helps with visual and usability problems that assertions rarely capture, but it also allows wrong judgments. Ye et al. report WebProber, a prototype that explores a site starting from its Uniform Resource Locator (URL), simulates clicking, typing, and scrolling, and writes a readable report. On 120 academic personal websites it found 29 usability issues, many of which traditional tools missed [12].

5) Reporting and analysis

Turning execution traces into a useful report is a good fit for language models. Sherifi et al. describe an agent-oriented framework that generates unit tests, runs them, and produces a report with results, rationales, and coverage. It was evaluated on Python and Java projects and targets unit testing rather than web testing [13]. Root-cause analysis remains difficult: an agent describes what failed much more reliably than it explains why.

B. Architectural Findings

1) Single-agent designs

A single-agent design uses one model in a perceive-reason-act loop. It is simple to build and easy to debug. Its weakness is that a single context has to carry the planning, the execution details, and the judgment together, so long tests can exhaust the context or drift from the goal. Even the baseline agent of Chevrot et al. encodes profile, memory, planning, and action in one prompt, while their stronger agent separates an orchestrator, an actor, and an assertor that share memory [1]. This suggests that useful testing agents tend toward role separation.

2) Perception: structure versus vision

Agents perceive a page in two broad ways. Text-based perception uses the accessibility tree or the Document Object Model (DOM). It is precise but can be large and noisy. Vision-based perception uses screenshots, often with markers on interactive elements. It matches what a person sees, but it is sensitive to resolution and layout. Systems for general web tasks, such as SeeAct [18], ground the model's decisions in screenshots, and benchmarks such as Mind2Web [19] and WebArena [20] provide environments for comparing approaches. Combining both channels is a common compromise.

3) Multi-agent designs

Multi-agent designs assign each lifecycle role to a specialized agent. A typical arrangement has an analyzer that studies the requirements or explores the application, a planner that produces test scenarios, an executor that drives the browser, a verifier that judges outcomes, and a reporter that compiles the results. The agents pass shared state

between them, and explicit orchestration allows retries, human checkpoints, and early stopping. For example, Almutawa et al. combine two LLM-powered agents (a test engineer and an HTML interpreter) with a non-LLM code executor to test web microservices [14]. The cost of this approach is complexity: more components, more prompts to maintain, more points of failure, and higher model usage. TABLE V contrasts the two designs.

TABLE V. SINGLE-AGENT AND MULTI-AGENT DESIGNS

Aspect	Single agent	Multi-agent
Structure	One model in one loop	Specialized agents with shared state
Context load	Planning, action, and judgment share one context	Each agent holds only its own context
Debugging	Simple to trace	Needs tracing across agents
Control flow	Implicit in the prompt	Explicit graph with retries and checkpoints
Cost	Lower model usage	Higher model usage and more prompts
Best suited to	Short, focused tests	End-to-end lifecycle automation

4) Tools and grounding

Agents act through tools. For web testing the essential tool is browser automation, and Playwright is a common choice. Other tools may include a database for test artifacts, a code interpreter, and an application programming interface (API) client. How an agent chooses among tools and forms valid tool calls is a research topic in itself [17], and a wrong choice is a source of error.

C. Evaluation Practice and Reported Results

1) What is measured

The most direct measure is agreement with a human tester: given a test whose correct verdict is known, does the agent reach the same verdict? Chevrot et al. treat this as a binary classification problem and report accuracy, specificity, and sensitivity. They also define step-alignment metrics, because an agent can give the right verdict for the wrong reason, which they call a type III error. Their true accuracy counts only verdicts that were reached for the right reason [1]. For exploratory approaches, the question changes to what the agent finds, for example the number of confirmed issues [12]. TABLE VI summarizes common measures.

TABLE VI. COMMON EVALUATION MEASURES

Measure	What it tells us	Caveat
Verdict agreement	Whether the agent reaches the human verdict on known tests	Depends on the quality of labeled tests
Sensitivity and specificity	Share of failing tests caught and of passing tests accepted	Trade-off between missed defects and false alarms
Step alignment	Whether a correct failure was found at the right step	Needs a recorded failing step
Defect detection	Confirmed defects found; precision and recall	Needs ground truth or manual confirmation
Coverage	Pages, features, or code reached	High coverage does not mean strong checking
Cost and time	Model calls, tokens, duration	Varies with model and provider
Maintenance effort	Repairs needed after a UI change	Hard to measure fairly

2) Test subjects and results

Controlled and uncontrolled test subjects serve different purposes. Offline copies of applications give reproducible conditions, as in the benchmark of Chevrot et al., whose three applications come from WebArena [1]. Live sites are more realistic but change over time, as in the in-the-wild study of Ye et al. [12]. In the controlled benchmark, the more advanced agent reached a true accuracy of 0.61, against 0.40 for the baseline, when both used the same model. The results changed little across three commercial models, but the agents tended to flag passing tests as failures, and specificity was never above 0.57 [1]. A qualitative analysis grouped the remaining errors into action capacity, action versatility, interface observability, assertion verifiability, and test-case conformity. General web-agent benchmarks measure task completion and not defect finding [20]. A benchmark that combines realistic applications, seeded defects, and changing interfaces is therefore still missing.

3) Threats to validity

Several issues limit how far the reported results can be trusted. Model versions change, so a result obtained today may not be repeatable next year. Results are also sensitive to prompts and sampling settings; Chevrot et al. fixed the temperature at zero to improve determinism [1]. Models may have seen popular public applications during training. Human testers also disagree with each other, so the reference verdict is not always certain. Good practice is to fix and report the model version and settings, run each test several times, publish the prompts, and prefer applications that the model is unlikely to have seen.

D. Comparison with Existing Approaches

TABLE VII compares the proposed design with current practice. Manual testing needs significant effort in every cycle. Scripted automation depends on predefined scripts that need constant upkeep. Existing AI-assisted tools help testers but do not make decisions on their own. The proposed system aims to make those decisions itself, while keeping a person in control of approval and review. The entries for the proposed system are design intentions and remain to be evaluated.

TABLE VII. COMPARISON OF TESTING APPROACHES

Approach	Human effort	Adaptation to UI change	Autonomy
Manual testing	High in every cycle	Human adjusts	None
Scripted automation	Medium; scripts need upkeep	Low; scripts break	Low
AI-assisted tools	Medium; tester decides	Partial	Assists only
Proposed system	Low; review and approval	Intended; to be evaluated	High, with oversight

E. Summary of Findings

Five findings follow from the analysis. First, LLMs are already useful for drafting test cases, but the drafts need review. Second, agents can execute natural-language tests without fixed locators, but their verdicts agree with human verdicts only part of the time, and they tend to over-report failures. Third, role separation, in the form of orchestrator, actor, and assertor, improved results over a single prompt in the one controlled comparison we found [1], which supports a multi-agent design. Fourth, exploratory agents can find usability problems that conventional tools miss, but the evidence comes from a small prototype study [12]. Fifth, evaluation practice is inconsistent, so results from different studies cannot be compared directly.

V. DISCUSSION

A. Challenges

Hallucination and incorrect actions. Models can invent elements, misread a page, or describe behavior that never happened. This produces both false defects and missed ones. Chevrot et al. found cases where an agent wrongly validated an assertion [1].

Non-determinism and reproducibility. The same instruction can produce different action sequences in different runs. A test that cannot be reproduced is hard to trust and hard to debug.

Cost and latency. Every step involves a model call and often a screenshot, so an agent run is slower and more expensive than replaying a script. Agents are better suited to exploration and triage than to frequent full-suite regression testing.

Oracle reliability. Deciding whether behavior is correct is harder than performing actions. An agent that judges its own result may agree with an application that is wrong. The risk is reduced by a separate verifier and, where the requirements are precise, by explicit assertions.

Evaluation practice. Metrics, datasets, and baselines differ across studies, as discussed in Section IV-C.

Safety and side effects. An autonomous agent on a live application can submit forms, delete data, or trigger payments. Testing should therefore use isolated environments, disposable data, and limited permissions.

Security and privacy. Sending page content to an external model service may expose confidential data. Locally hosted open models reduce this risk, but at the cost of capability and extra infrastructure effort.

Human trust. Teams adopt tools that they can inspect. Agents that keep a trace of their reasoning, actions, and evidence are easier to audit than agents that return only a verdict.

B. Research Gap

The surveyed work shows progress on separate stages but not on their integration. Tools automate individual tasks rather than the whole process, and many still need manual intervention between stages. Root-cause analysis and reporting remain largely manual, and systems rarely adapt automatically when the application changes. Multi-agent integration for testing is still limited, partly because it is complex to implement. A platform that goes from requirement analysis through planning and execution to reporting, without manual scripting in between, has not yet been convincingly demonstrated.

C. Open Problems

Shared benchmarks with seeded defects, changing interfaces, and ground-truth verdicts are needed so that studies can be compared. *Self-healing and adaptation* should let agents distinguish an intentional change from a regression. *Reliable oracles* should combine explicit assertions, visual comparison, and independent verifier agents. *Root-cause analysis* should link a failure to logs, network traces, and code changes. *Cost-aware*

testing should use agents for exploration and triage and scripts where determinism matters. *Human-in-the-loop design* should define where a person approves plans and reviews uncertain verdicts. *Safe execution* requires sandboxed environments and permission models.

D. Benefits, Limitations, and Scope of the Proposed Design

The expected benefits are less manual effort, test generation without hand-written scripts, adjustment to application behavior and detected changes, structured defect analysis and reports, and reuse of the platform across web applications. The expected limitations are the complexity of initial development, incorrect or incomplete AI-generated test cases, execution cost, and wrongly labeled or missed defects. These match the challenges in Section V-A and will guide the evaluation. The design is technically feasible with Python, FastAPI, Playwright, and LangGraph. It is economically feasible because the tools are open source, and operationally feasible because it reduces repetitive effort. The current scope covers web exploration, test generation, UI and functional testing, defect detection, validation, and test reports. API, performance, and security testing are planned extensions.

E. Implications for Practitioners

The findings suggest a cautious way to adopt agentic testing today. Agents are best used where scripts are weakest: exploring a new or poorly documented application, drafting an initial set of test cases for human review, and triaging failures with evidence attached. Scripts remain the better choice for stable, frequently repeated regression checks, because they are fast, cheap, and deterministic. Teams should record the model version, prompts, and settings with every run, keep a person responsible for approving plans and for resolving cases marked as needing review, and run agents only in isolated environments. A hybrid arrangement, in which agents explore and generate and scripts replay what has proved stable, uses the strengths of both.

VI.CONCLUSION

Agentic AI turns web testing from the replay of fixed scripts into goal-driven interaction with an application. The literature shows that language models help generate tests, that agents can execute natural-language test cases and explore applications for usability problems, and that a lifecycle view is a productive way to organize the field. It also shows that the results are preliminary, that evaluation is inconsistent, and that no existing work convincingly covers the whole lifecycle. Hallucination, non-determinism, cost, oracle reliability, and safety must be addressed before such systems can be trusted in production. In our view, multi-agent pipelines with explicit orchestration, shared benchmarks, and human checkpoints are the most promising route.

This work has limitations. The review is narrative and not exhaustive, many of the cited findings come from preprints, and the proposed platform has been designed but not yet implemented or evaluated. Our next step is to build the prototype and to run the evaluation protocol of Section III-D, so that the claims made here can be tested against measured results.

ACKNOWLEDGMENT

The authors thank the Department of Computer Engineering, SVPM's College of Engineering, Baramati, for its support during this work.

REFERENCES

[1] A. Chevrot, A. Vernotte, J.-R. Falleri, X. Blanc, and B. Legeard, "Are autonomous web agents good testers?," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 206–228, 2025.

[2] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?," in *Proc. IEEE Int. Conf. Software Testing, Verification and Validation (ICST)*, 2016, pp. 180–190.

[3] M. Leotta, B. García, F. Ricca, and J. Whitehead, "Challenges of end-to-end testing with Selenium WebDriver and how to face them: A survey," in *Proc. IEEE Conf. Software Testing, Verification and Validation (ICST)*, 2023, pp. 339–350.

[4] L. Wang et al., "A survey on large language model based autonomous agents," *Front. Comput. Sci.*, vol. 18, no. 6, p. 186345, Dec. 2024.

[5] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: Wiley, 2012.

[6] J. Wang et al., "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, Apr. 2024.

[7] R. Santos, I. Santos, C. Magalhaes, and R. de Souza Santos, "Are we testing or being tested? Exploring the practical applications of large language models in software testing," in *Proc. IEEE Conf. Software Testing, Verification and Validation (ICST)*, 2024, pp. 353–360.

[8] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.

[9] P. P. Kumaresen, M. Frasheri, and E. P. Enoiu, "Agent-based software testing: A definition and systematic mapping study," in *Proc. IEEE 20th Int. Conf. Software Quality, Reliability and Security Companion (QRS-C)*, 2020, pp. 24–31.

[10] S. Paydar and M. Kahani, "An agent-based framework for automated testing of web-based systems," *J. Softw. Eng. Appl.*, vol. 4, pp. 86–94, 2011.

[11] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Towards autonomous testing agents via conversational large language models," in *Proc. 38th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2023, pp. 1688–1693.

[12] N. Ye, X. Yu, R. Xu, T. Peng, and Z. Yu, "AI agents for web testing: A case study in the wild," arXiv:2509.05197, 2025.

[13] B. Sherifi, K. Slhoub, and F. Nembhard, "The potential of LLMs in automating software testing: From generation to reporting," arXiv:2501.00217, 2024.

[14] M. Almutawa, Q. Ghabrah, and M. Canini, "Towards LLM-assisted system testing for microservices," in *Proc. IEEE 44th Int. Conf. Distributed Computing Systems Workshops (ICDCSW)*, 2024, pp. 29–34.

[15] S. Haldar, M. Pierce, and L. F. Capretz, "Exploring the integration of generative AI tools in software testing education: A case study on ChatGPT and Copilot for preparatory testing artifacts in postgraduate learning," *IEEE Access*, vol. 13, pp. 46070–46090, 2025.

[16] M. M. N. Fuad and K. Sakib, "eBAT: An efficient automated web application testing approach based on tester's behavior," in *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*, 2022, pp. 482–486.

[17] M. Singh, K. Rao, G. Coviello, and S. T. Chakradhar, "TacTool: Tactical tool usage in agentic AI systems," in *Proc. IEEE ICA*, 2025, pp. 32–35.

[18] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su, "GPT-4V(ision) is a generalist web agent, if grounded," in *Proc. 41st Int. Conf. Machine Learning (ICML)*, 2024, vol. 235, pp. 61349–61385.

[19] X. Deng et al., "Mind2Web: Towards a generalist agent for the web," arXiv:2306.06070, 2023.

[20] S. Zhou et al., "WebArena: A realistic web environment for building autonomous agents," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2024.