ISSN:1738-6764, eISSN:2093-7504

Towards Adaptive Test Automation: JSON DSLs and LLM Agents for End-to-End Testing

Dong Kwan Kim [1,*]

[1] Mokpo National Maritime University; dongkwan@mmu.ac.kr
* Correspondence

https://doi.org/10.5392/IJoC.2026.22.1.077

Manuscript Received 25 November 2025; Received 24 December 2025; Accepted 16 January 2026

Abstract: *End-to-end (E2E) testing of web user interfaces is a crucial but resource-intensive endeavor, often complicated by intricate user workflows, fragile scripts, and significant maintenance costs. Traditional UI testing frameworks, such as Playwright, offer precise control but demand substantial manual effort and are sensitive to changes in the user interface. To address these challenges, this paper introduces an approach that utilizes Large Language Models (LLMs) for the automated generation and execution of E2E tests. We present two complementary paradigms: (1) a JSON-based domain-specific language (DSL) that facilitates declarative test specification and deterministic execution through the Playwright Model Context Protocol (MCP), and (2) an agent-based testing framework in which an LLM dynamically plans and executes actions based on high-level objectives, progress state, and evolving UI snapshots. As a case study, we employ the Vendure e-commerce framework, which offers a representative testing environment featuring functionalities like product search, shopping cart management, payment workflows, and administrative tasks. The JSON-based DSL approach is evaluated for its effectiveness in enhancing test script readability, reusability, and maintainability. In parallel, the agent-based model showcases adaptability and self-healing capabilities. Experimental results indicate that LLM-driven test automation alleviates the burden of manual script creation, improves test coverage across user and administrative scenarios, and provides resilience against changes in application interfaces. By integrating structured JSON-based methods with adaptive agent-based reasoning, this work lays the groundwork for more robust, flexible, and scalable AI-driven test automation frameworks.*

Keywords: End-to-End Test; Automated Testing; Large Language Models; Model Context Protocol; Agents

1. Introduction

Web applications present substantial difficulties for end-to-end (E2E) testing, as they incorporate rich interaction flows, continually evolving user interfaces, and multiple dependencies on external services. These properties frequently result in brittle test scripts that require significant time and effort to construct and maintain, especially in rapidly changing environments where minor UI modifications can disrupt existing test cases. Despite these limitations, E2E testing remains a critical practice for verifying end-user functionality and ensuring the dependable operation of deployed systems. Advances in Large Language Models (LLMs) have opened new pathways for automating diverse software engineering activities, including code synthesis, defect identification, requirements understanding, and test generation. Their generative capabilities enable the transformation of high-level intentions or natural-language descriptions into executable test procedures. Although established automation frameworks such as Playwright [1] provide robust browser-level control, they continue to rely on manually authored scripts. LLM-based techniques offer a compelling opportunity to overcome these constraints by decreasing human involvement, improving resilience to UI changes, and expanding the scope of test coverage.

In this paper, an LLM-enhanced testing framework is explored that integrates GPT-4o with the Playwright Model Context Protocol (MCP) [2, 3] to automatically generate and execute test scripts. The proposed approach introduces two complementary paradigms:

1. A JSON-based domain-specific language (DSL) for deterministic and declarative test specification, ensuring machine executability and reproducibility.
2. An Agent-based testing model, where the LLM dynamically plans and executes actions step-by-step using contextual information such as goals, progress state, and evolving UI snapshots.

To evaluate the applicability of the proposed approach, a case study is conducted using Vendure [4], an open-source e-commerce framework that offers representative complexity for real-world testing scenarios. In this case study, the test cases cover both end-user workflows (e.g., login, product search, shopping cart management, and checkout) and administrative functions (e.g., product registration, customer management, and order operations).

The main contributions of this paper are as follows:

- The proposed approach integrates an LLM-driven method for automatic E2E test script generation with MCP and Playwright.
- A domain-specific language is designed to connect human-readable descriptions with executable JSON-based test scripts.
- The proposed framework is extended with an Agent-based testing paradigm, introducing adaptability and self-healing behavior in dynamic UI environments.
- The proposed approach is validated through a case study on Vendure, covering both user and administrative functionalities in a realistic e-commerce setting.

The remainder of this paper is structured as follows. Section II reviews related work. Section III presents the design and implementation details of the proposed LLM-based test automation framework. Section IV presents experimental results on Vendure. Section V discusses key findings and limitations, and Section VI concludes with final remarks and future directions.

2. Related Work

End-to-end testing research has mostly focused on traditional automation tools like Selenium, Cypress, and Playwright. Existing comparative studies show that Playwright outperforms Cypress and Selenium in synchronization handling, execution stability, and resource efficiency [5]. Further analyses confirm Playwright's superior reliability in complex GUI workflows, along with advantages in cross-browser consistency and reduced flakiness [6, 7]. Despite these strengths, script-based automation still remains brittle when UI structures evolve or dynamic behaviors deviate from expected flows.

To address these limitations, prior work has explored higher-level abstractions through domain-specific languages, enabling declarative specification of user interactions and reducing the burden of imperative scripting. DSL-based approaches—including JSON and YAML representations—improve readability, modularity, and reusability. Additionally, research findings highlight their value in structuring tool integration and execution models, such as the Model Context Protocol (MCP) [8, 9]. Nevertheless, many DSL approaches remain rigid and fail to adapt automatically when UI layouts or workflows change.

Recent advancements in Large Language Models have reshaped the testing landscape. Recent work shows that LLM-based systems can automate reasoning-intensive tasks, while state-of-the-art analyses highlight their expanding role in software testing [10, 11]. LLMs have been shown to generate executable web-form test cases directly from UI descriptions [12] and to support broader applications such as exploratory testing and regression analysis. Additional work examines LLM-guided unit test generation [13], prompt-driven agent construction, and the use of LLMs for automated programming tasks [14] including repair and synthesis. Practical evaluations further document the adoption of LLMs in real-world testing environments [15].

Building on these foundations, recent studies examine autonomous LLM-based software agents capable of planning, executing, and validating actions over extended workflows [16, 17]. Complementary research investigates versioning strategies, maintenance practices, and lifecycle considerations for agent-based software systems [18, 19]. These works highlight the potential of LLM-driven testing but also reveal challenges related to predictability, tool safety, and integration with deterministic automation engines.

In contrast to these individual efforts across automation tools, the presented work introduces a unified framework that integrates LLM planning with Playwright via MCP, provides a JSON-based DSL to bridge natural-language descriptions with executable scripts, and incorporates an adaptive agent capable of recovery under evolving UI conditions. A case study on the Vendure platform demonstrates the practicality and robustness of this hybrid approach.

3. Proposed Approaches

Traditional Playwright testing requires developers to manually write and maintain test scripts in TypeScript or JavaScript. While effective in many scenarios, this approach remains inherently brittle, as even minor modifications to the user interface or workflow can easily cause test failures, and it further imposes a substantial programming burden on software testers who must continuously update and refine the underlying scripts to maintain reliability. To mitigate these limitations, this study proposes LLM-based testing, which leverages large language models to make test automation more adaptive, expressive, and maintainable. Within the category of LLM-based testing, two complementary approaches are introduced. The first is JSON-based testing with Playwright-MCP [20, 21], where high-level test scenarios are expressed in a structured JSON format. This approach simplifies test authoring, improves readability, and provides a deterministic bridge between human-readable instructions and machine-executable scripts. The second is Agent-based testing, where the tester specifies only high-level goals, and an LLM-driven agent—equipped with planning, action execution, memory, and verification—autonomously generates and executes the necessary test steps. Addressing the inherent limitations of traditional Playwright-based testing, these LLM-driven approaches seek to improve tester productivity, lower maintenance overhead, and introduce self-healing capabilities for automated end-to-end workflows.

3.1 JSON-Based Testing

Figure 1 illustrates the standard end-to-end testing workflow in Playwright, beginning with the developer manually authoring test scripts that describe the intended user interactions with the web application. The scripts are executed in an instrumented browser environment that replicates real user behavior, allowing the system to observe how the UI responds to each action. During execution, assertions continuously evaluate whether the resulting interface states satisfy predefined expectations, and any deviations are recorded as observable defects. Playwright also collects detailed execution traces, which provide developers with actionable insights for diagnosing the root causes of failures. Using these diagnostic artifacts, the developer refines the application logic or updates the test scripts before re-running the test suite to verify correctness. The workflow concludes once all assertions succeed and the system confirms that the application has passed the end-to-end test, reflecting a deterministic, script-driven methodology that requires continuous maintenance as the UI evolves.

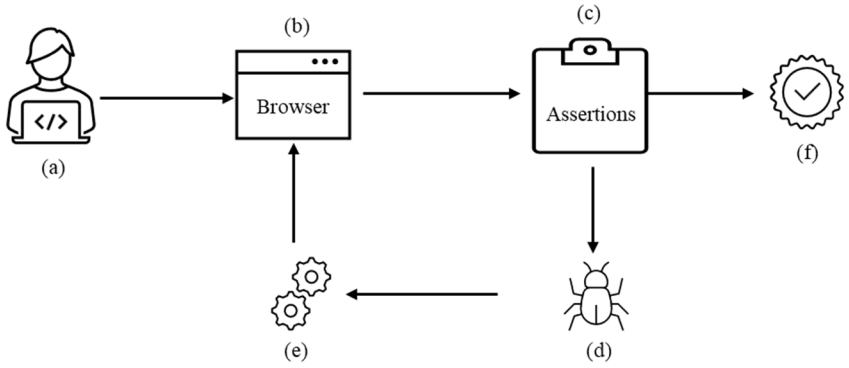


Figure 1. Overview of End-to-End Testing Workflow in Playwright: (a) The developer writes the test scripts. (b) The test scripts are executed in the web browser. (c) Assertions check whether the UI behaves as intended after interactions. (d) Traces capture execution details to facilitate debugging. (e) The developer fixes the identified errors and reruns the tests (f) The web page successfully passes the test.

JSON-based testing with Playwright-MCP is a method of representing test scenarios as structured JSON objects rather than imperative code. Each action—such as navigation, form filling, clicking, or waiting—is expressed in JSON with explicit parameters. The JSON script is then executed via the MCP bridge, which maps the instructions to Playwright commands that interact with the browser. This approach simplifies test authoring, improves readability, and ensures that scenarios are both human-accessible and machine-executable. By bridging high-level descriptions with deterministic automation, JSON-based testing enhances maintainability and provides a foundation for integration with AI-driven test generation.

Table 1 presents an example of how a high-level end-to-end test scenario, expressed in a human-readable DSL, is systematically transformed into its equivalent JSON-based MCP test script. This comparison underscores the capability of the Playwright-MCP framework to bridge the gap between intuitive test specification and machine-executable test automation.

Table 1. DSL Example and its corresponding MCP Test Script

```
#Go to the user login page
navigate to "http://localhost:8002/sign-in"
#Sign-in with userID and password
fill textbox with css selector "#email" as
"email@mmu.ac.kr"
fill textbox with css selector "#password" as "passwd"
click "Sign In" button
wait until URL contains "/account"
#Add a product to the shopping cart
navigate to "http://localhost:8002/products/spiky-cactus"
click element with text "Add to Cart"
wait until element with text "Add to Cart" is visible
#Checkout
click element with aria-label "Open cart tray"
click "Checkout" button
wait until URL contains "/checkout"
```

(a) End-to-End Test Scenario in DSL

```
[{ "type": "navigate", "url": "http://localhost:8002/sign-in" },
  { "type": "fill", "selector": { "css": "#email" }, "value":
"email@mmu.ac.kr" },
  { "type": "fill", "selector": { "css": "#password" }, "value":
"passwd" },
  { "type": "click", "selector": { "role": "button", "name": "Sign In" } },
  { "type": "waitForURLContains", "urlPart": "/account" },
  { "type": "navigate", "url": "http://localhost:8002/products/spiky-
cactus" },
   { "type": "click", "selector": { "text": "Add to Cart" } },
   { "type": "waitForSelector", "selector": { "text": "Add to
Cart" } },
  { "type": "click", "selector": { "ariaLabel": "Open cart
tray" } },
  { "type": "click", "selector": { "role": "button", "name":
"Checkout" } },
{ "type": "waitForURLContains", "urlPart": "/checkout" }]
```

(b) MCP Test Script

The DSL is formally defined using Extended Backus–Naur Form (EBNF), which provides a rigorous grammar specification. The DSL abstracts common user interactions with web applications, such as navigation, form input, element clicking, and waiting for specific conditions to be met on the page. At the highest level, a program is defined as a sequence of one or more commands, with each command representing a discrete testing action. Declarative in nature, the DSL allows test scenarios to be described in a way that emphasizes intent rather than implementation details, thereby improving readability, automation feasibility, and potential integration with test generation systems. The DSL grammar supports several categories of commands. The navigate command directs the browser to a specified URL, where the destination is provided as a quoted string. The fill command exists in two variants: one locates a textbox by its accessible role or label, while the other uses a CSS selector. Both forms accept a string value to be entered into the field, thereby supporting both semantic and technical element identification. For user interactions, the click command is represented in three distinct forms, enabling testers to specify actions based on different element attributes. In addition, the DSL includes commands for waiting, such as verifying whether the current URL contains a specific substring or whether a DOM element with particular text becomes visible.

On the left-hand column of Table 1 (a), the DSL-based test case illustrates a typical e-commerce scenario: navigating to a login page, entering credentials, verifying account access, adding a product to the cart, and completing the checkout process. Each step is described in the DSL with a high degree of readability, making the script accessible even to stakeholders without extensive programming expertise. The right-hand column (b) demonstrates how these DSL commands are compiled into structured JSON objects for execution. Each instruction is translated into a machine-readable JSON entry with explicit attributes such as “type”, “selector”, “url”, and “value”. For example, the DSL command “fill textbox with css selector ‘#email’ as ‘email@mmu.ac.kr’” is transformed into a JSON object that specifies the action type (“fill”), the selection method (“css”), and the input value. Similarly, navigational, interactive, and waiting commands are mapped to JSON structures that eliminate ambiguity and ensure deterministic execution. The JSON representation facilitates interoperability within CI/CD pipelines, ensures consistency across test executions, and allows extensions with metadata for logging or recovery. The DSL serves as a human-friendly abstraction optimized for test authoring, while the JSON script serves as a canonical, machine-readable format for automated execution.

3.2 LLM Agent-Based Testing

LLM Agent-based Testing represents a paradigm shift from traditional scripted approaches to autonomous, goal-driven test automation. Instead of writing imperative code or structured JSON scripts, the tester specifies only a high-level testing goal (e.g., “verify checkout flow succeeds”). The LLM-powered agent then plans, executes, and validates the testing process autonomously through an integrated architecture.

The agent typically consists of four core components:

- Planner – decomposes the high-level goal into a sequence of actionable steps.
- Actions – generate concrete commands (e.g., navigation, click, input) that can be executed by external tools.
- Tools – such as Playwright or MCP, which interact directly with the web application under test.
- Memory – stores past interactions, test history, and contextual hints, supporting self-healing in case of UI changes.
- Verifier/Observer – evaluates the outcomes of actions by checking assertions, validating DOM states, or confirming success criteria.

Figure 2 illustrates the data flow in an LLM Agent-based web testing system using a Planner-Actor-Executor architecture. It starts with two inputs: Goal Description (the test objective) and Browser Page (current web page state). The PLANNER builds a prompt by combining the goal, page snapshot from TOOLS, and progress from MEMORY, and sends it to the LLM API to generate an Action Plan (e.g., navigate, click, wait). MEMORY tracks the test progress with components like Added Set (items added to the cart), Last Action Sig (for loop detection), and Progress State (completed vs. remaining tasks). The TOOLS component captures the page state through Snapshot and provides Playwright Commands for browser automation. The ACTIONS component routes the action plan through Action Router and executes it via Execute Action, which translates it into Playwright commands. VERIFIER checks the outcomes through Outcome Check and updates MEMORY. The data flows cyclically: Goal and Page inputs are captured, combined with Progress to build the prompt, which is then processed by the LLM API to generate the action plan. The plan is executed via Playwright commands, verified, and the memory is updated accordingly. This iterative process enables autonomous, adaptive testing that responds to changing page states without relying on pre-scripted test flows.

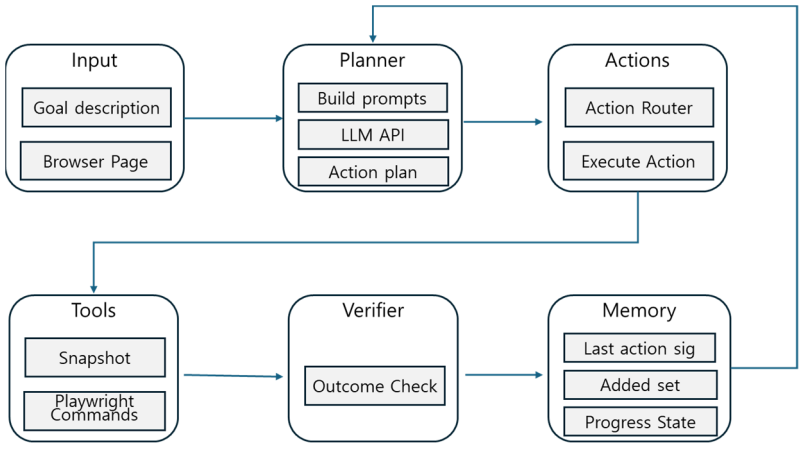


Figure 2. LLM Agent Architecture

Table 2 describes the pseudo code for LLM Agent-Based Web Testing. An LLM-driven Playwright agent automates the entire web application testing process: signing in, adding products to the cart, and proceeding to checkout. The core of this approach is the prompt, which acts as an instruction set, guiding the LLM while allowing flexibility. The prompt first defines the goal, such as adding products to the cart and reaching checkout. It tracks progress, specifying completed and remaining tasks, and includes a snapshot of the environment, such as the page URL, title, clickable elements, and indicators of page type (e.g., product or checkout page). The prompt also enforces a structured response format, requiring the LLM to output a JSON object specifying one of the allowed actions (e.g., navigate, click, waitForSelector) with necessary arguments and a justification. Rules are embedded to ensure the agent only performs actions related to product, cart, and checkout tasks, and navigates to the first remaining product when multiple options are available.

Table 2. Pseudo Code for LLM Agent-based Web Testing: mcp-login-checkout-agent.js

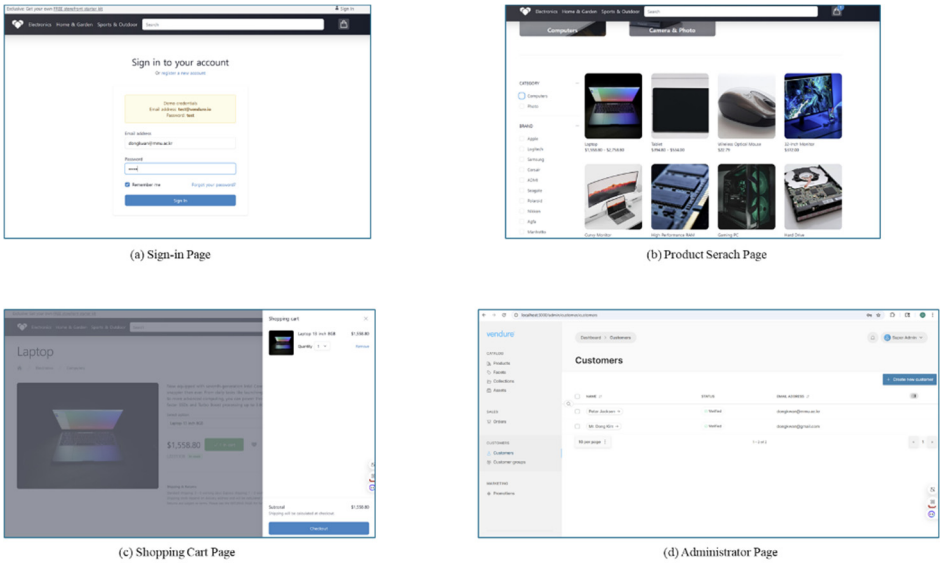
```
COMPONENTS:
  PLANNER → LLM decides next step (JSON action)
  ACTIONS → {navigate, click, wait, complete, fail}
  TOOLS   → Playwright commands
  MEMORY  → track added items, last action, visited URLs
  VERIFIER→ check outcomes (cart updated, URL reached)

MAIN:
  launch browser, sign_in_if_needed()
  for step in 1..MAX_STEPS:
    snapshot = capture_state(page)
    if product page and not added: TOOLS.addToCart(); MEMORY.update()
    else if all products added: TOOLS.openCart(); TOOLS.checkout(); break
    else:
      plan = PLANNER(goal, progress, snapshot)
      if loop(plan, MEMORY.last_action): TOOLS.goto(next remaining)
      else: execute(plan, TOOLS, VERIFIER); MEMORY.update(plan)
    if reached("/checkout" or "/sign-in"): break
  close browser
```

This approach allows the LLM to make decisions dynamically based on the evolving UI state, with Playwright handling the low-level browser interactions. By replacing pre-scripted test flows with adaptive decision-making, the agent can continue test execution even when minor interface changes occur, providing a more robust and flexible testing solution. Prompt engineering significantly affects the flexibility, accuracy, and reliability of LLM Agent-based testing. Well-engineered prompts enhance task comprehension and reduce misinterpretation, while structured instructions and explicit context increase task completion accuracy and consistency. Well-structured prompts reduce output variability and enhance reproducibility, supporting adaptable updates to evolving test scenarios while also enabling more efficient error recovery and debugging. Moreover, optimized prompt templates enable scalability to new domains with minimal retraining and reduce human effort in post-editing. In contrast, weakly engineered prompts lead to unstable outputs, inconsistent results, and higher manual debugging costs. They lack the context and structure necessary to guide the model's reasoning, causing output variability. Incomplete prompts also hinder reproducibility and test coverage. However, prompt engineering offers a scalable alternative to fine-tuning, providing adaptability without modifying model parameters. High-quality prompts are thus essential to enhancing the performance stability and effectiveness of LLM-driven test automation systems, offering both flexibility and control across dynamic testing environments.

4. Experimental Results

4.1 Subject Application

To evaluate the effectiveness of the proposed approach, a case study is conducted on Vendure, an open-source headless e-commerce platform that provides representative complexity for automated testing. Vendure incorporates the core functionalities of modern e-commerce systems, such as product catalog browsing, search, user login, shopping cart management, and checkout processes. In addition, its open-source nature provides complete experimental control over both the frontend and backend, ensuring clarity and reproducibility of expected behaviors. The system under evaluation consists of the Vendure client and the Vendure server. The Vendure client is a browser-based storefront developed with contemporary frontend technologies such as Angular or React. It enables end users to perform typical online shopping tasks, including navigating product categories, accessing detailed product descriptions, placing items into the shopping cart, and proceeding through the checkout process. Figure 3 highlights representative user-facing pages: the Login Page supports authentication, the Product Search and Listing Page displays available items with "Add to Cart" functionality, the Product Detail Page offers extended product information with purchasing options, and the Cart and Checkout Page allows customers to review selections and finalize orders.



(a) Sign-in Page (b) Product Serach Page

(c) Shopping Cart Page (d) Administrator Page

Figure 3. Main Web Pages of the Vendure Application

Along with the client pages, the Vendure server provides the backend infrastructure. It exposes a rich GraphQL API that supports catalog management, order processing, user account administration, promotions, and inventory control. The backend also offers plugin-based extensibility, enabling developers to integrate payment gateways, shipping solutions, or additional administrative tools. As part of the case study, the Administration Page—used by system administrators to manage customer records and operational data—is additionally evaluated. The Vendure client and server constitute a realistic, interactive e-commerce environment, making it a robust testbed for validating the proposed LLM-based test automation framework. The combination of a dynamic and feature-rich user interface on the client side with a structured, extensible backend API provides a practical platform for designing, executing, and assessing context-aware automated test scripts.

4.2 Test Scenarios and Interaction Types

Table 3 summarizes a set of representative test scenarios designed to evaluate both user-side and administrative functionalities of the Vendure-based e-commerce application. The scenarios cover typical end-to-end workflows that capture the essential interactions of an online shopping system, including login, product search, cart management, checkout, and administrative tasks such as product and customer management.

Table 3. Test Scenarios for User and Admin Modes in the Vendure E-Commerce Application

Mode	Test Scenario	Description
User	**TS#1:** Login/Product/Checkout	The user logs into their account, adds products to the cart, and completes the checkout process.
	TS#2: NoLogin/Product/Checkout	The user, without logging in, adds products to the cart and attempts to complete the checkout process as a guest.
	TS#3: Login/Product/Checkout/ Payment	The user logs in, selects items, proceeds to checkout, and completes the order by submitting payment details.
	TS#4: Login/Product/Cart/Checkout	In the shopping cart, users can modify item quantities or cancel/remove products.
	TS#5: ProductSearchByKeywords	Users can find products by typing search queries.
	TS#6: ProductSearchByCategoryImages	Users can find products by clicking category images.
Admin	**TS#7:** AdminLogin/Product/Registration	The administrator logs into the admin panel and adds a new product to the catalog, including name, price, and inventory details.

	TS#8: AdminLogin/Customer/Registration	The administrator logs in and registers a new customer manually through the admin interface.
	TS#9: AdminLogin/Product/Update/Delete	The administrator can update and remove product details.
	TS#10: AdminLogin/Customer/Update/Delete	The administrator can update and remove customer information.
	TS#11: AdminLogin/InfoUpdate	The administrator modifies their profile details.
	TS#12: AdminLogin/Order/Search/Delete	The administrator can find and remove customer orders.

The following user-mode flows are considered.

- TS#1 (Login/Product/Checkout) simulates a standard user activity in which the customer logs into their account, selects products, adds them to the cart, and completes the checkout process.
- TS#2 (NoLogin/Product/Checkout) evaluates system behavior when a guest user attempts checkout without authentication, testing how unauthorized access is handled (e.g., redirection to the login page).
- TS#3 (Login/Product/Checkout/Payment) extends the flow to include payment submission, covering the complete purchase cycle from login to final order confirmation.
- TS#4 (Login/Product/Cart/Checkout) examines cart-level interactions, allowing users to update item quantities or remove products before checkout.
- TS#5 (ProductSearchByKeywords) and TS#6 (ProductSearchByCategoryImages) capture search functionalities, testing the system's ability to retrieve products via keyword input or category-based navigation.

For admin mode, the following scenarios validate management tasks within the administration interface.

- TS#7 (AdminLogin/Product/Registration) assesses how an administrator logs in and registers a new product, specifying attributes such as name, price, and inventory.
- TS#8 (AdminLogin/Customer/Registration) covers manual customer registration via the admin panel.
- TS#9 (AdminLogin/Product/Update/Delete) evaluates administrative operations for modifying or removing product details.
- TS#10 (AdminLogin/Customer/Update/Delete) extends this to customer management, verifying the ability to update or delete customer records.
- TS#11 (AdminLogin/InfoUpdate) represents profile management where administrators can update their own account information.
- TS#12 (AdminLogin/Order/Search/Delete) tests the system's capability to allow administrators to search for and delete customer orders.

These scenarios provide a comprehensive evaluation framework for assessing both end-user functionality and administrative control in the Vendure e-commerce system. They highlight the system's robustness in handling core e-commerce workflows as well as its flexibility in supporting administrative management tasks, making it a suitable environment for validating the proposed LLM-based automated testing approach.

The distribution of interaction types in Table 4 across twelve test scenarios (TS#1 to TS#12) reveals key insights into the complexity and behavior of automated tests. Common actions like navigation and clicks are prevalent, highlighting their importance in user and administrative workflows. Notably, every test scenario involves at least one navigation action, indicating that multi-page transitions are fundamental to web testing. Scenarios such as TS#1 and TS#2, with five navigation steps each, are the most complex, featuring stages like authentication and checkout, often involving session-dependent flows. Similarly, click actions are frequent, ranging from five to eleven occurrences per scenario, emphasizing the interactive nature of workflows. Form-based interactions—especially those involving data entry into input fields—significantly contribute to the overall complexity of test scenarios. TS#8, corresponding to customer registration, records the highest number of fill actions, while administrative workflows in TS#5 also feature numerous form submissions. In comparison, user-centric flows like TS#3 and TS#4 show moderate form interaction, focusing on login and product registration inputs.

Table 4. Distribution of Interaction Types Across Test Scenarios

Scenario	Interaction Type							
	navigate	fill	click	waitForURLContains	waitForSelector	select Option	evaluate	press
TS#1	5	2	7	2	4	-	-	-
TS#2	5	-	6	2	4	-	-	-
TS#3	2	5	5	2	1	-	-	-
TS#4	3	2	6	2	3	1	1	-
TS#5	4	5	1	4	3	-	-	3
TS#6	3	2	3	3	3	-	-	-
TS#7	1	4	5	-	-	-	-	-
TS#8	1	8	5	-	-	-	-	-
TS#9	1	3	10	1	4	-	-	2
TS#10	1	3	10	1	3	-	-	-
TS#11	1	5	6	-	-	-	-	4
TS#12	1	2	11	-	-	-	-	-

Synchronization mechanisms, such as *waitForURLContains* and *waitForSelector*, are crucial for ensuring reliable test execution in dynamic web environments. Scenarios like TS#1 and TS#2 employ these steps after key actions like login or checkout, ensuring that pages loaded correctly before proceeding with further steps. This reflects the need for explicit synchronization in real-world testing to handle asynchronous content rendering. In terms of overall complexity, TS#1 emerges as the most interaction-rich, simulating a full checkout process with over sixteen interactions, while TS#2 is notable for its heavy reliance on *waitForSelector* actions, indicating a need for handling dynamic content. TS#5, the most form-intensive scenario, involves numerous fill actions and keyboard inputs. Conversely, TS#9, TS#10, and TS#12 are more focused on clicks, showcasing the diversity of interaction patterns. The test suite also incorporates specialized actions such as *selectOption* and *evaluate*, which increase coverage by addressing edge cases that simpler tests often overlook. These specialized actions reflect realistic user interactions, such as selecting from dropdown menus or submitting forms via keyboard.

Overall, the analysis highlights that effective end-to-end testing requires a balance of navigation, form entry, synchronization, and specialized actions. The presence of synchronization mechanisms across scenarios underscores the importance of accounting for asynchronous UI behaviors. Additionally, the differences between user and admin workflows guide the design of adaptive test generation models like Playwright-MCP, which should tailor interactions based on the specific needs of each scenario type, ensuring robust and comprehensive test coverage.

4.3 Quantitative Experimental Results

This study presents a quantitative evaluation of LLM agent–based web testing in comparison with traditional Playwright script–based testing, conducted on the Vendure e-commerce application using 12 representative test scenarios. The LLM agent was implemented using GPT-4o via the OpenAI API, and all test scenarios were executed on the actual Vendure application without simulation or mock environments to reflect realistic testing conditions. Each of the 12 test scenarios was executed five times for both the LLM agent and traditional scripts to reduce execution variance and improve statistical reliability, resulting in a total of 120 test executions (60 LLM agent runs and 60 traditional script runs). During each execution, success or failure status, execution time, error messages, failure causes, and adaptation behavior in UI change scenarios were systematically recorded to enable a comprehensive quantitative comparison between the two testing approaches.

To provide quantitative evidence, the proposed LLM agent is quantitatively compared with traditional script-based testing in terms of success rate, productivity metrics, and self-healing capability. Success Rate is evaluated using two complementary metrics: Test Pass Rate under Stable UI conditions and Test Pass Rate under UI Changed conditions, which together measure both baseline correctness and robustness to UI variations. Productivity Metrics is quantified by Average Authoring Time per scenario, capturing the time and effort required to create and maintain test cases. Self-healing Capability is measured by the Recovery Success Rate, which reflects the ability of the testing approach to recover and continue execution after UI changes without

manual intervention. The results in Table 5 demonstrate that the proposed LLM agent consistently outperforms traditional scripts across all quantitative performance metrics. Regarding Success Rate, the LLM agent achieves a higher pass rate under both stable and changed UI conditions, with a particularly large performance gap observed when UI changes occur. In terms of Productivity Metrics, the dramatic reduction in authoring time quantitatively confirms that the agent significantly reduces manual effort in test development and maintenance. Finally, the substantially higher Recovery Success Rate provides direct quantitative evidence of the agent's self-healing capability, showing that it can autonomously adapt to UI changes where traditional scripts fail.

Table 5. Quantitative Performance Comparison between Traditional Script-Based Testing and LLM Agent-Based Testing

Metric	Traditional Playwright	LLM Agent	Interpretation
Test Pass Rate (Stable UI)	72.5%	85.0%	Success Rate (baseline correctness)
Test Pass Rate (UI Changed)	16.3%	82.5%	Success Rate under UI variation (robustness)
Average Authoring Time (per test scenario)	15.0 minutes	2.0 minutes	Productivity gain
Recovery Success Rate	20.0%	85.0%	Self-healing capability

The cost and efficiency analysis reveals a fundamental trade-off between flexibility and operational efficiency when comparing LLM Agent–based testing with JSON DSL–based testing. LLM Agents demonstrate strong adaptability to UI changes through context-aware decision-making and self-healing behaviors, which significantly reduce manual maintenance efforts. However, these advantages are accompanied by increased execution latency, reliance on external APIs, and recurring operational costs. In contrast, JSON DSL–based testing offers deterministic execution, zero per-run API costs, and faster execution times, making it highly suitable for large-scale and performance-critical automation pipelines; however, it provides limited resilience to UI changes and requires higher maintenance effort due to brittle selectors. Table 6 presents a quantitative comparison of token usage, API cost, and execution time across 12 test scenarios. As shown in Table 6, LLM Agent–based tests incur an average token usage of 7,232 tokens per scenario, corresponding to an average API cost of approximately $0.02 per execution. Moreover, the average execution time of the LLM Agent reaches 296.63 seconds, which is substantially longer than that of the JSON DSL approach (39.33 seconds on average). This performance gap is primarily attributable to repeated reasoning cycles, network latency, and retry mechanisms inherent in agent-driven execution. In contrast, JSON DSL executions benefit from direct, predefined workflows that eliminate external dependencies, resulting in consistently faster runtimes and predictable performance characteristics.

Table 6. Comparison of Token Usage, API Cost, and Execution Time between LLM Agent and JSON DSL across Test Scenarios

Test Scenarios	Token Usage (LLM Agent)	API Cost ($) (LLM Agent)	Execution Time (sec)	
			LLM Agent	JSON DSL
TS#1	1,283	0.004	602.04	35.94
TS#2	5,046	0.016	297.80	18.52
TS#3	1,691	0.005	602.05	30.24
TS#4	3,982	0.013	602.03	45.07
TS#5	1,134	0.004	14.27	10.78
TS#6	1,447	0.004	71.67	7.60
TS#7	5,903	0.022	88.18	20.13
TS#8	13,713	0.044	57.16	45.06
TS#9	11,180	0.037	194.36	71.71
TS#10	11,367	0.039	192.07	71.59
TS#11	12,951	0.042	297.02	50.04
TS#12	17,090	0.054	540.95	65.23
Avg.	7,232	0.02	296.63	39.33

To further contextualize these empirical results, Table 7 summarizes the key operational trade-offs between the two approaches across multiple dimensions, including flexibility, cost, execution speed, maintenance effort, reliability, and scalability. As indicated in Table 7, LLM Agent–based testing excels in flexibility and maintenance efficiency by dynamically adapting to UI changes and reducing the need for manual selector updates. However, this flexibility comes at the expense of higher operational costs and slower execution. Conversely, JSON DSL–based testing achieves superior execution speed, reliability, and scalability by enforcing deterministic control flows and avoiding API dependencies, though it remains less adaptable to interface changes and requires more frequent manual updates. Overall, the results indicate that neither approach is universally optimal. Instead, their strengths are complementary: LLM Agents are better suited for adaptive, low-frequency, or exploratory testing scenarios, while JSON DSL–based testing is more appropriate for stable, high-frequency, and cost-sensitive CI/CD pipelines. These findings motivate hybrid testing strategies that selectively combine LLM Agents for resilience and adaptability with JSON DSLs for efficient and scalable execution.

Table 7. Trade-off Matrix: LLM Agent vs. JSON DSL

Factor	LLM Agent	JSON DSL
Flexibility	High	Low
Operational Cost	High ($ 0.02 per scenario)	Zero
Execution Speed	Slow (296.63 sec)	Fast (39.33 sec)
Maintenance Effort	Low	High
Reliability	Medium	High
Scalability	Medium	High

4.4 Qualitative Experimental Results

Table 8 summarizes the relative comparison between traditional Playwright testing and JSON-MCP testing, presenting their performance across key dimensions such as implementation effort, synchronization robustness, performance efficiency, and maintainability using a relative evaluation scale (High, Medium, and Low). The table highlights that JSON-MCP generally demonstrates higher adaptability and ease of maintenance, whereas Playwright excels in performance, debugging transparency, and handling complex procedural logic. Implementation effort is lower in JSON-MCP because it minimizes manual coding and accelerates test creation for large or repetitive suites, while Playwright requires more effort but enables detailed, condition-based control. In handling complex logic, Playwright scores higher due to its imperative scripting flexibility, whereas JSON-MCP, as a declarative model, favors simplicity over granular control. Synchronization robustness is greater in JSON-MCP because it dynamically adapts to asynchronous UIs and variable network latency, while Playwright relies on more static wait conditions. Playwright maintains higher runtime efficiency under CPU-bound operations, but JSON-MCP outperforms it in long-term maintainability, as its schema-based structure allows quick adaptation to UI or selector changes. JSON-MCP also enhances accessibility, allowing non-developers to author or modify tests easily, and offers superior portability through a transferable JSON schema that can integrate with other frameworks like Cypress or Selenium. Conversely, Playwright's strengths lie in its fine-grained debugging capabilities and precise runtime visibility, making it indispensable in diagnosing complex performance issues. Overall, both paradigms play complementary roles rather than competing ones. Traditional Playwright provides the control, transparency, and performance precision essential for complex, timing-sensitive workflows, while JSON-MCP delivers adaptability, maintainability, and framework-agnostic scalability suited to dynamic, data-driven environments. The most sustainable strategy is a hybrid integration, where Playwright functions as the execution engine for complex, performance-critical tests and JSON-MCP serves as the orchestration layer for regression and synchronization-stable suites. This layered approach combines procedural precision with declarative flexibility, reducing maintenance time while improving test reliability.

Table 8. Integrated Comparative Summary between traditional Playwright and JSON+MCP-Based Testing

Evaluation Dimension	Traditional Playwright	JSON + MCP	Complementary Insight
Implementation Effort (Ease of Authoring)	Medium	High	JSON requires minimal manual coding; faster for repetitive or large suites.
Complex Logic & Conditional Flow	High	Low	Playwright allows detailed procedural control for complex test logic.
Synchronization Robustness (Flakiness)	Medium	High	JSON adapts dynamically to unstable UI or slow network latency.
Performance Efficiency	High	Medium	Playwright achieves faster runtime under CPU-bound operations.
Maintenance Adaptability	Low	High	JSON structure is easier to update when UI selectors or elements change.
Debugging & Transparency	High	Medium	Playwright provides clearer stack traces and runtime visibility.
Team Accessibility / Usability	Medium	High	JSON enables participation of non-developer testers.
Cross-Framework Portability	Medium	High	JSON schema is easily transferable to Cypress or Selenium.
Standardization & Consistency	Medium	High	JSON promotes uniform test definitions and governance.
Overall Suitability (Aggregate Level)	Medium–High	High	JSON shows higher adaptability; both are complementary in hybrid use.

Table 9 summarizes the relative comparison between LLM Agent-based testing and traditional Playwright-based testing, presenting their flexibility levels across multiple operational dimensions using a qualitative three-level scale (High, Medium, and Low). The table shows that the LLM Agent approach consistently achieves “High” scores in most areas—such as requirement adaptation, cross-platform migration, incremental enhancement, domain switching, and natural-language variation handling—while traditional Playwright testing typically maintains “Medium” or “Low” levels, excelling primarily in code expressiveness and quality consistency. This indicates that Playwright’s deterministic scripting excels at handling complex control logic and ensuring predictable output, whereas the LLM Agent framework demonstrates greater adaptability and scalability when responding to dynamic, evolving environments. Specifically, LLM Agents adjust faster to changing requirements, generalize learned interaction patterns across domains, and handle linguistic variations robustly. Their reliance on natural-language prompts also removes framework-specific barriers, allowing easier migration between platforms such as Cypress and Selenium. However, Playwright remains advantageous for maintaining explicit procedural control, offering transparency and stability for regression or compliance testing.

Table 9. Relative Flexibility Assessment between Traditional Playwright and LLM Agent-Based Testing

Flexibility Dimension	Traditional Playwright	LLM Agent Approach	General Observation
Requirement Adaptation	Medium	High	LLM Agents adjust faster to changing or evolving user stories.
Cross-Platform Migration	Low	High	Natural-language prompts remove framework-specific barriers.
Incremental Enhancement	Medium	High	New features can be incorporated with minimal manual edits.
Domain Switching (Portability)	Low	High	LLM Agents generalize learned patterns across domains.
Natural-Language Variation Handling	Medium	High	Robust to linguistic variation and multilingual phrasing.
Code Expressiveness (Control Logic)	High	Medium	Traditional scripting excels in complex logic.
Quality Consistency	High	Medium	Manual code retains more predictable quality.
Overall Flexibility Level	Medium	High	LLM Agent testing demonstrates broader adaptability.

The comparative analysis highlights that LLM Agent-based testing represents a paradigm shift from procedural to intention-driven automation. Rather than coding explicit "how to test" instructions, testers define "what to test," allowing the LLM Agent to generate or repair executable steps dynamically. This significantly reduces manual maintenance, as the system can reinterpret high-level intents when workflows or UI components evolve. Nonetheless, LLM-driven tests can vary in consistency, as model outputs depend on contextual inference, while handcrafted Playwright scripts guarantee repeatable precision. Consequently, the most effective strategy is a hybrid testing model: LLM Agents function as adaptive orchestrators for exploratory or rapidly evolving environments, while Playwright acts as the execution engine for deterministic, performance-critical workflows.

5. Discussion

The Discussion section provides a comparative analysis of the testing approaches proposed in this work, followed by a comparison with existing testing approaches to contextualize the contributions and clarify the relative advantages and limitations of the proposed methods.

❑ Comparative analysis of the proposed testing approaches

This study compares traditional Playwright testing with two LLM-based approaches—JSON-based testing via Playwright-MCP and autonomous agent-based testing. In the traditional Playwright approach, developers explicitly author test scripts in TypeScript or JavaScript. While this approach ensures precise control over browser actions and assertions, it is inherently brittle: even minor user interface changes may invalidate selectors, causing frequent test failures. Moreover, the cost of writing and maintaining test code is high, requiring specialized programming expertise and significant manual effort. These limitations make the approach less scalable in dynamic environments where the application evolves rapidly.

By contrast, the LLM-based approaches shift the burden of explicit scripting toward higher-level representations and adaptive planning. In the JSON-based Playwright-MCP method, tests are expressed in a declarative JSON format generated from DSL specifications. Test authors can describe scenarios at a higher level of abstraction, improving readability and reducing maintenance overhead. At the same time, the deterministic JSON schema guarantees that outputs remain machine-executable, bridging human accessibility with reliable automation. However, this method still requires scenario design in advance and may lack the flexibility to adapt when unexpected UI variations occur. The Agent-based approach further extends this paradigm by delegating decision-making to the LLM itself. Instead of relying on pre-authored JSON scripts, the agent dynamically interprets goals, progress, and UI snapshots, planning the next action step-by-step. This enables a form of self-healing automation, where the test can adapt to variations in UI layout, labels, or workflows without manual updates. The trade-off is the introduction of uncertainty: while agent-based testing is more flexible, it may also be less predictable, with potential risks of incorrect decisions or infinite loops if not properly constrained. The use of memory, verification, and schema-based prompts mitigates these issues but does not entirely eliminate them.

These findings in the study highlight a continuum: traditional testing offers precision but lacks adaptability, JSON-based testing balances structure and abstraction, and agent-based testing emphasizes autonomy and resilience at the cost of predictability. For practical deployment, the choice among these approaches may depend on project context: highly stable systems may benefit from traditional scripting, rapidly evolving interfaces may favor JSON-based methods, and exploratory or adaptive environments may find the agent-based model most effective. Overall, the introduction of LLM-based testing demonstrates a promising shift toward more maintainable and intelligent test automation. Future work should further investigate hybrid strategies that combine the determinism of JSON scripts with the adaptivity of agent-based reasoning, thereby achieving both robustness and flexibility in large-scale end-to-end testing. Table 10 provides comparative analysis of testing approaches.

Table 10. Comparative Analysis of Testing Approaches

Approach	Pros	Cons
Traditional Playwright Testing	- Precise, deterministic control of test execution - Mature ecosystem and tooling - Well-suited for stable UI environments	- Brittle: minor UI changes often break tests - High maintenance cost - Requires programming expertise
LLM-based JSON Testing (Playwright-MCP)	- Declarative, human-readable format - Reduces coding effort and maintenance - Deterministic: JSON schema ensures machine executability - Easier integration into CI/CD pipelines	- Still requires predefined scenarios - Limited adaptability to unexpected UI changes - Relies on manual scenario design
LLM-based Agent Testing	- Dynamic and adaptive (self-healing capabilities) - Requires only high-level goals, not detailed scripts - Can handle UI variations and evolving workflows - Reduces manual maintenance	- Less predictable outcomes - Potential for incorrect or looping actions - Requires strict constraints (schemas, rules, verifiers) - Higher computational cost

❑ Comparison with existing testing approaches

Unlike Selenium IDE, which provides rich control-flow constructs (e.g., loops, conditional branching, and variables) through a GUI-based recorder, the JSON DSL employed in this work adopts a deliberately simple, linear execution model consisting of sequential action steps. This design choice enables lightweight representation, programmatic generation, and seamless integration with CI/CD pipelines, but comes at the cost of limited expressiveness. In particular, the JSON DSL does not natively support loops, conditional branching, variables, or complex business logic, and lacks built-in error recovery and test suite organization mechanisms, as confirmed through an analysis of 12 JSON DSL files. Compared to existing LLM-based testing studies, which typically focus on one-time generation of static test scripts or emphasize agent architectures without systematic evaluation, this work is distinctive in several key aspects. First, it presents a hybrid perspective by explicitly comparing an LLM agent–based approach with a DSL-based alternative within the same experimental framework. Second, all evaluations are conducted on a real-world e-commerce application (Vendure), rather than simulated or toy environments. Third, the study provides comprehensive quantitative metrics—including success rate under both stable and changed UI conditions, productivity measured by authoring time, and self-healing capability measured by recovery success rate—thereby offering objective evidence beyond qualitative observations. Finally, the technical limitations of the JSON DSL are explicitly documented, including its inability to handle loops, conditional logic, dynamic values, and adaptive error recovery, which clarifies the boundary conditions under which the DSL-based approach is appropriate and highlights scenarios where the LLM agent offers clear advantages.

6. Conclusions and Future Work

This paper presented the design and implementation of an automated framework for generating end-to-end test scripts using the MCP in combination with large language models. The proposed approach introduces a JSON-based domain-specific language that allows high-level testing scenarios to be expressed declaratively and then systematically transformed into machine-executable Playwright scripts. Along with the JSON-based approach, an LLM Agent-based testing model is also explored, in which the language model assumes a more autonomous role by dynamically planning and executing test actions based on high-level goals and evolving UI states. This agent-driven paradigm introduces adaptivity and self-healing capabilities that go beyond rigid pre-scripted tests, offering resilience in environments where user interfaces evolve rapidly. While the agent-driven approach holds promise, it also raises challenges of predictability, reliability, and the need for strict constraints to ensure safe and correct execution. Findings from the evaluation indicate that both approaches have significant potential to advance automated software testing. The JSON-based DSL ensures stability and reproducibility, whereas the agent-based approach introduces adaptability and robustness. Together, these methods illustrate the broader potential of LLMs to augment or transform test automation practices. To validate this concept, the proposed approach is applied to the Vendure e-commerce framework, generating end-to-end test scripts for

both user-facing and administrative functionalities. The case study demonstrates that the JSON-based testing paradigm reduces scripting effort, improves readability, and enhances maintainability while ensuring deterministic execution.

For future work, several promising research directions emerge that can further extend the capabilities, generality, and practical applicability of the proposed approach. First, large-scale empirical studies are required to systematically evaluate the scalability, efficiency, and accuracy of MCP-based LLM testing across a wider range of web applications, UI complexities, and domain contexts. Expanding the experimental scope beyond a single case study will allow more robust conclusions to be drawn regarding generalization performance and operational feasibility in real-world testing environments. Second, hybrid testing strategies that combine the determinism and reproducibility of JSON-based DSL scripts with the adaptability and reasoning capabilities of LLM Agent–based execution represent an important direction for future exploration. Such hybrid approaches could dynamically switch between scripted execution and agent-driven reasoning depending on UI stability, test criticality, or runtime uncertainty, thereby achieving a better balance between reliability and flexibility. Third, future research should investigate richer verification and validation mechanisms, including the integration of formal specifications, natural language requirements, assertion mining, or runtime monitoring. These techniques could strengthen the semantic validity of generated test cases and improve confidence in automated test outcomes, particularly in safety- or compliance-critical systems. In addition, the current study focuses on GPT-4o as the underlying language model for the LLM Agent to maintain experimental consistency and to isolate the effects of the proposed MCP-based interaction protocol and agent architecture. However, a comprehensive evaluation of alternative large language models—such as Gemini, Claude, and Llama—constitutes an important and meaningful extension of this work. Comparative studies across different model families would provide deeper insights into the impact of model architecture, reasoning capability, and cost-performance trade-offs on automated testing behavior. Furthermore, prompt calibration and optimization strategies warrant systematic investigation. Comparative analyses before and after prompt refinement could clarify how prompt design influences agent planning quality, execution stability, and self-healing performance. Such studies are expected to contribute practical guidelines for deploying LLM-based testing agents in production settings. Finally, cost optimization and model selection strategies remain a critical consideration for real-world adoption. Future work should explore adaptive model selection, caching mechanisms, and execution policies that minimize token usage and latency while preserving test effectiveness, particularly in high-frequency CI/CD pipelines.

Acknowledgments: This research was supported by Mokpo National Maritime University in 2024.

Conflicts of Interest: The authors declare no conflict of interest.

References

[1] Playwright. Accessed: Nov. 16, 2025. [Online] Available: https://playwright.dev/

[2] Model Context Protocol. Accessed: Nov. 16, 2025. [Online] Available: https://modelcontextprotocol.io/introduction

[3] Model Context Protocol Github. Accessed: Nov. 16, 2025. [Online] Available: https://github.com/modelcontextprotocol

[4] Vendure remix storefront starter. Accessed: Nov. 16, 2025. [Online] Available: https://github.com/vendure-ecommerce/storefront-remix-starter

[5] M. Moń and B. Pańczyk, "A comparative analysis of web application test automation tools," JCSI Journal, vol. 35, pp. 159–165, 2025, doi: https://doi.org/10.35784/jcsi.7119.

[6] S. Müller, "Comparing Static, and Dynamic Synchronization of GUI-based Tests," Master Thesis, Department of Software Engineering, Blekinge Institute of Technology, Karlskrona, Sweden, 2023.

[7] N. L. P. M. Wati, I M. D. P. Asana, N. W. S. Putri, K. J. Atmaja, and I G. I. Sudipa, "Comparison of Automation Testing on Card Printer Project Using Playwright and Selenium Tools," Journal of Computer Networks, Architecture and High Performance Computing, vol. 6, no. 3, pp. 1309–1320, Jul. 2024, doi: https://doi.org/10.47709/cnahpc.v6i3.4362.

[8] X. Hou, Y. Zhao, S. WANG, and H. WANG, "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," arXiv:2503.23278, Mar. 2025, doi: https://doi.org/10.48550/arXiv.2503.23278.

[9] P. P. Ray, "A Survey on Model Context Protocol: Architecture, State-of-the-art, Challenges and Future Directions," TechRxiv, Apr. 2025, doi: https://doi.org/10.36227/techrxiv.174495492.22752319/v1.

[10] C. Augusto, A. Bertolino, G. De Angelis, F. Lonetti, and J. Morán, "Large Language Models for Software Testing: A Research Roadmap," arXiv:2509.25043, Sep. 2025, doi: https://doi.org/10.48550/arXiv.2509.25043.

[11] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Models: Survey, Landscape, and Vision," IEEE Trans. Software Engineering, vol. 50, no. 4, pp. 911–936, Apr. 2024, doi: https://doi.org/10.1109/TSE.2024.3368208.

[12] T. Li, C. Cui, R. Huang, D. Towey, and L. Ma, "Large Language Models for Automated Web-Form-Test Generation: An Empirical Study," arXiv:2405.09965, May. 2024, doi: https://doi.org/10.48550/arXiv.2405.09965.

[13] A. Lops, F. Narducci, A. Ragone, M. Trizio, and C. Bartolini, "A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites," arXiv:2408.07846, Aug. 2024, doi: https://doi.org/10.48550/arXiv.2408.07846.

[14] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, "Automatic Programming: Large Language Models and Beyond," ACM Trans. Software Eng. & Methodology, vol. 34, no. 5, May. 2025, doi: https://doi.org/10.1145/3708519.

[15] X. Tan, Y. Jiang, Y. Yang, and H. Xu, "Towards End-to-End Optimization of LLM-based Applications with Ayo," arXiv:2407.00326, Jun. 2024, doi: https://doi.org/10.48550/arXiv.2407.00326.

[16] LLM Agents. Accessed: Nov. 16, 2025. [Online] Available: https://www.promptingguide.ai/research/llm-agents

[17] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying LLM-Based Software Engineering Agents," in Proc. ACM FSE 2025, Article FSE037, vol. 2, pp. 801–824, Jul. 2025, doi: https://doi.org/10.1145/3715754.

[18] C. Liu, L. Zhang, X. Xu, W. Guo, and Y. Liu, "Towards the Versioning of LLM-Agent-Based Software," in Proc. FSE Companion '25, Trondheim, Norway, Jun. 2025, doi: https://doi.org/10.1145/3696630.3728714.

[19] W. Xu, C. Huang, S. Gao, and S. Shang, "LLM-Based Agents for Tool Learning: A Survey," Data Science and Engineering, vol. 10, pp. 533–563, Jun. 2025, doi: https://doi.org/10.1007/s41019-025-00296-9.

[20] Playwright MCP Github. Accessed: Nov. 16, 2025. [Online] Available: https://github.com/microsoft/playwright-mcp

[21] T. Ozodakhon, J. S. Jeong, and D. K. Kim, "Automated E2E Testing via Prompt-Driven DSLs and Language Models," in Proc. The 26th International Symposium on Advanced Intelligent Systems, Nov. 6–9, 2025.

APPENDIX

A. Hardware & Software Environment

The experiments were conducted on a Windows 11 Pro system equipped with an Intel(R) Core(TM) Ultra 9 185H processor (16 physical cores and 22 logical processors, x64 architecture) and a single NVIDIA GeForce RTX 4070 GPU. The LLM model used for the experiments is GPT-4o (gpt-4o), which is accessed via OpenAI API. GPT-4o is a large language model developed by OpenAI with multimodal capabilities supporting both text and image inputs. For inference, the mode was configured with the temperature parameter as 0.1 to ensure deterministic and consistent output, and set the top P parameter to 0.95 for nucleus sampling. The maximum number of steps per test scenario was set to 50, which is configurable via the MAX_STEPS environment variable, and each API call has a timeout of 60 seconds. The experiments utilized Playwright version 1.54.1 for browser automation, running on Node.js version 20.19.4. The target application for our experiments was the Vendure e-commerce platform, with the storefront running on http://localhost:8002 and the admin panel on http://localhost:3000. In addition, the model was configured with the maximum input sequence length to be determined dynamically based on the prompt content, with system prompts enforcing JSON-only output format to prevent hallucinations and ensure parseable responses.

B. System Prompt Example

The system prompts illustrated in Figure 4 define the role of the LLM and impose strict output format constraints, thereby playing a critical role in ensuring consistent and machine-parseable responses. Specifically, they prevent hallucination by explicitly prohibiting explanatory text, markdown code blocks, or any non-JSON content, ensure that the output can be directly parsed as a JSON object without preprocessing, and reduce token usage by eliminating unnecessary verbosity. These system prompts are applied when the agent requires a structured JSON response conforming to a predefined schema, enforcing the LLM to output only the JSON object without any additional text. Based on an analysis of 12 agent implementations, several system prompt design principles are recommended to ensure reliable and consistent LLM behavior. First, explicit format specification is essential, requiring the exact output format to be clearly stated (e.g., "JSON only"), unwanted formats such as markdown or code blocks to be explicitly prohibited, and imperative language (e.g., "Output only," "Do not include") to be used to enforce constraints. Second, system prompts should emphasize brevity by remaining concise (typically one to two sentences), avoiding redundancy with user prompts, and focusing

primarily on output constraints rather than contextual information, which should be delegated to user prompts. Third, consistency should be maintained by applying the same system prompt patterns across similar agents, standardizing format requirements to minimize parsing errors, and documenting any prompt variations when deviations are necessary. Finally, careful language choice is required, recommending alignment with the language of user prompts for coherence, the use of English to enhance international reproducibility when appropriate, and ensuring clarity and unambiguity regardless of the selected language.

```
javascript
      messages: [
            { role: 'system', content: ' Output only JSON. No explanations, no code fences.' },
            { role: 'user', content: prompt },
      ],
```

Figure 4. System Prompt Example

C. User Prompt Examples

The user prompt example shown in Figure 5. provides the LLM agent with explicit task objectives, execution context, and operational constraints to support goal-directed web testing. It specifies the overall testing goal and dynamically tracks progress by distinguishing between completed and remaining products, along with the current browser URL. To enable informed decision-making, the prompt exposes the current page state, including the page title, a subset of clickable elements, and high-level semantic signals indicating whether the agent is on a product page, authentication page, account page, or whether key actions such as “Add to cart,” “Open cart,” or “Checkout” are available. In addition, a strict JSON schema is defined to constrain the agent’s output to a single, machine-parseable action, ensuring compatibility with Playwright-based execution. Finally, a set of domain-specific rules narrows the agent’s focus to product selection, cart interaction, and checkout workflows, while explicitly excluding authentication and enforcing deterministic navigation and action-selection behavior.

```
Goal:
${goal}

Progress Status:
✓ Added products: ${JSON.stringify(completed)}
☐ Remaining products: ${JSON.stringify(remaining)}
Current URL: ${snap.url}

Current State:
TITLE: ${snap.title}
CLICKABLE: ${JSON.stringify(snap.clickable.slice(0,40))}
SIGNALS: {"isProductPage": ${snap.isProduct}, "isSignIn": ${snap.isSignIn}, "isAccount": ${snap.isAccount},
"hasAddToCart": ${snap.hasAddToCart}, "hasOpenCart": ${snap.hasOpenCart}, "hasCheckout":
${snap.hasCheckout}}

Schema (output exactly one JSON object only):
{
   "action": "navigate" | "click" | "waitForSelector" | "waitForURLContains" | "complete" | "fail",
   "args": {
      "url"?: string,
      "selectorText"?: string,
      "urlPart"?: string
   },
   "why": "brief reason"
}
Rules:
-Authentication is handled by a separate routine; focus only on product selection, cart interaction, and checkout.
-If there are remaining products, navigate to the first remaining product. If already on that product page, click "Add to
cart" (select options if required).
-After all products have been added, click the cart icon (accessibility name: "Open cart tray") to open the tray, then
click the "Checkout" button and confirm arrival at /checkout.
-Always use absolute URLs.
```

Figure 5. User Prompt Example

D. Prompt Design Principles

Prompt engineering best practices were derived from empirical observations of the agent implementations and are summarized across temperature configuration, prompt length, and language choice. For temperature settings, a low value of 0.1 (with top_p set to 0.95, as used in mcp-login-checkout-agent.js) is recommended to promote deterministic behavior, ensure consistent JSON output structure, and reduce hallucinations by limiting creative but incorrect responses; although this configuration reduces exploratory diversity and may overlook alternative valid actions, the trade-off is acceptable given the strict reliability requirements of structured output. With respect to prompt length optimization, an effective range of approximately 200–500 words was identified, as overly short prompts (<100 words) often lack sufficient contextual and state information, leading to incorrect action selection, while excessively long prompts (>1000 words) increase token costs, risk information overload, and yield diminishing returns. The current implementation adopts prompts of approximately 300–400 words, which fall within the optimal range. Finally, regarding language choice, English is recommended to enhance international reproducibility; consistent language usage across system and user prompts is essential, and clarity must be preserved regardless of language.

E. Hallucination Prevention Mechanism

Hallucinations in LLM agent-based testing typically manifest as invalid actions, non-existent selectors, or malformed JSON outputs, and are addressed through a multi-layered defense architecture. First, strict system-level output constraints combined with a robust JSON extraction and normalization process defensively parse the LLM response, removing markdown artifacts and isolating the first valid JSON object even when extraneous text is present. Second, action validation enforces a whitelist-based policy that permits only predefined, type-safe actions and argument structures, thereby rejecting unsupported or ill-formed plans at an early stage. Third, action normalization mitigates variability in LLM-generated field names by mapping semantically equivalent arguments to a standardized schema, ensuring execution consistency. Finally, when validation fails, the agent defaults to a safe fallback action (fail) rather than executing an undefined behavior, enabling graceful error handling and recovery. Collectively, these layered mechanisms prevent format, action, and field-level hallucinations while guaranteeing that the execution engine always receives a valid and interpretable plan.

F. Infinite Loop Prevention Mechanism

Infinite loops in LLM agent-based testing occur when an agent repeatedly executes identical or non-progressive actions without advancing toward the target goal, potentially leading to unbounded execution and excessive resource consumption. To address this issue, a multi-layered loop prevention strategy is employed. At the foundation, a hard execution limit (MAX_STEPS) enforces an absolute upper bound on the number of agent iterations, guaranteeing termination even in cases where higher-level detection mechanisms fail. Complementing this safeguard, action signature tracking monitors consecutive action–argument pairs and identifies repetitive behavior at an early stage; upon detecting such repetition, the agent is forced to navigate to a known valid state, such as the next unprocessed product URL, thereby breaking the loop and restoring progress. In addition, explicit goal completion checks terminate execution immediately once predefined success conditions—such as reaching the checkout page—are satisfied, preventing redundant actions after task completion. These mechanisms ensure bounded execution, reduce unnecessary LLM invocations, protect system resources, and improve overall robustness by handling both action-level repetition and stalled navigation patterns in a deterministic and recoverable manner.

G. Error Handling Mechanism

Error handling mechanisms are designed to ensure that LLM agent-based testing remains robust in the presence of unexpected runtime failures. At the action level, each Playwright operation is executed within a try–catch block (e.g., try { await page.goto(url); } catch (e) { log(e); }), allowing the agent to continue execution even when individual actions fail. To prevent indefinite blocking, timeout constraints are applied to waiting operations such as waitForSelector, ensuring bounded execution time. In addition, selector fallback strategies attempt multiple interaction patterns—such as role-based, aria-label, or text-based selectors—before declaring failure, increasing tolerance to UI variations. Finally, a process-level error handler captures uncaught exceptions and terminates execution gracefully with explicit error reporting (e.g., .catch(err => process.exit(1))), enabling reliable failure detection in automated pipelines. These mechanisms prevent crashes, reduce hangs, and support graceful degradation under adverse execution conditions.

H. Recovery Mechanisms

Recovery mechanisms enable the agent to actively regain progress after failures or stagnation by combining deterministic execution paths with adaptive state restoration. For critical and repetitive operations, such as adding products to the cart or initiating checkout, guard paths replace LLM-driven decisions with rule-based logic (e.g., if (snap.isProduct) ensureAddToCartClicked();), improving reliability and reducing unnecessary LLM calls. When loop-like behavior is detected through repeated action signatures (e.g., JSON.stringify([action, args])), the agent forcibly navigates to a known valid state, such as the next remaining product URL, thereby breaking the loop and restoring forward progress. In addition, recovery is supported at the action level through fallback selector strategies, which transparently retry alternative interaction methods when an initial attempt fails. These recovery mechanisms enhance execution resilience, improve success rates in dynamic web environments, and ensure that the agent can autonomously return to a productive state without manual intervention.