Are Autonomous Web Agents Good Testers?

ANTOINE CHEVROT, Smartesting, France
ALEXANDRE VERNOTTE, Smartesting, France
JEAN-RÉMY FALLERI, University of Bordeaux - LaBRI - UMR 5800, France
XAVIER BLANC, University of Bordeaux - LaBRI - UMR 5800, France
BRUNO LEGEARD, Smartesting, France
AYMERIC CRETIN, Smartesting, France

Despite advances in automated testing, manual testing remains prevalent due to the high maintenance demands associated with test script fragility—scripts often break with minor changes in application structure. Recent developments in Large Language Models (LLMs) offer a potential alternative by powering Autonomous Web Agents (AWAs) that can autonomously interact with applications. These agents may serve as Autonomous Test Agents (ATAs), potentially reducing the need for maintenance-heavy automated scripts by utilising natural language instructions similar to those used by human testers. This paper investigates the feasibility of adapting AWAs for natural language test case execution and how to evaluate them.

We contribute with (1) a benchmark of three offline web applications, and a suite of 113 manual test cases, split between passing and failing cases, to evaluate and compare ATAs performance, (2) SeeAct-ATA and pinATA, two open-source ATA implementations capable of executing test steps, verifying assertions and giving verdicts, and (3) comparative experiments using our benchmark that quantifies our ATAs effectiveness. Finally we also proceed to a qualitative evaluation to identify the limitations of PinATA, our best performing implementation.

Our findings reveal that our simple implementation, SeeAct-ATA, does not perform well compared to our more advanced PinATA implementation when executing test cases (50% performance improvement). However, while PinATA obtains around 60% of correct verdict and up to a promising 94% specificity, we identify several limitations that need to be addressed to develop more resilient and reliable ATAs, paving the way for robust, low maintenance test automation.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Autonomous Tester Agent, Benchmark, Web Automation

1 Introduction

In the ever-evolving landscape of web development, ensuring the quality and reliability of web applications is a critical endeavour. Traditionally, application testing has relied heavily on manual processes, particularly for end-to-end test cases where the system under test is a web application.

Authors' Contact Information: Antoine Chevrot, Smartesting, Besançon, France, antoine.chevrot@smartesting.com; Alexandre Vernotte, Smartesting, Besançon, France, alexandre.vernotte@smartesting.com; Jean-Rémy Falleri, University of Bordeaux - LaBRI - UMR 5800, Bordeaux, France, falleri@labri.fr; Xavier Blanc, University of Bordeaux - LaBRI - UMR 5800, Bordeaux, France, xavier.blanc@u-bordeaux.fr; Bruno Legeard, Smartesting, Besançon, France, bruno.legeard@smartesting.com; Aymeric Cretin, Smartesting, Besançon, France, aymeric.cretin@smartesting.com.

While these manual approaches are effective, they have significant drawbacks regarding time, cost, and scalability. As web applications grow in complexity and the demand for faster development cycles intensifies, the limitations of manual testing become more apparent. Manual tests are not only time-consuming to execute, but also prone to human error and inconsistency. Furthermore, the cost of maintaining a large team of manual testers can be prohibitive for many organisations, especially as applications require more frequent updates and releases, which is the case of web applications.

To overcome these limitations, test automation was introduced, with the main objective of providing executable test code. In the context of web applications, several frameworks and tools have been proposed to facilitate the production of this code, from automatic browser manipulation to automated test code generated by recorded user behaviour. Today, however, test automation fails to live up to its promise, mainly because automated test scripts are too often fragile and require significant maintenance effort. This conclusion for web applications is observed in both Record/Replay tests [17], and end-to-end tests [22, 29]. We are even witnessing the emergence of methods for estimating the cost of maintenance and thus measuring the benefits of deploying full test automation [9].

Artificial intelligence (AI), particularly the generative capabilities of large language models (LLMs), has demonstrated potential for generating automated test scripts [34, 40, 41]. The primary goal is to reduce the time required to create automated tests and to minimise errors. Additionally, LLMs provide valuable support in generating CSS selectors, a common source of test fragility. However, while these advancements enhance the efficiency of test code production, the generated code remains specific to a particular version of the target application, leaving test fragility unresolved.

Recently, Autonomous Web Agents (AWA), which leverage LLMs and integrate the capability to interact with web applications [35], have demonstrated the potential to perform tasks autonomously [5, 18, 19, 42], as described in Section 2. This development raises an intriguing question: could an AWA execute test cases, effectively taking over the role of manual testers in web application testing?

Manual test instructions closely resemble the input formats that AWAs use, and their broad prior knowledge may allow these agents to be more naturally resilient to minor changes in web application design compared to traditional test code. This is therefore not a surprise that we are now witnessing the emergence of autonomous testing agents in commercial products such as Flowtest.ai or Octomind.dev. However, while a few products exist, little is known about how these autonomous testing agents work and what results they are capable of, raising the question of their concrete interest.

In this article, we seek to answer these questions. Since there is no prior literature on how to build an autonomous testing agent, we leverage the literature on AWAs to create and implement two alternative designs for a test agent, presented in Section 3: a baseline design and a more advanced design where we select and adapt to the testing context state-of-the art concepts from web agents.

To evaluate these designs on their ability to execute test cases on web applications, we create a fully reproducible benchmark, presented in Section 4, containing web applications and manual test cases written by four professional testers (i.e., written in natural language) and define metrics to evaluate the ability of an LLM agent to execute these tests automatically. This benchmark is built with extensibility in mind and we hope that it will help driving the research in the area of autonomous test agents.

Finally, we use our benchmark to evaluate our two alternative designs and analyse their performances though a quantitative study, described in Section 5. We also proceed to a qualitative assessment of the behaviour of agents to identify the pain points of our agents, yielding future research directions for the community.

►	TC-4-P :: Login, search by keyword & category, choose an add, and leave a comment - updated	
#	Actions	Expected Result
1	Start on the Homepage: http://www.vtaas-benchmark.com:9980/	Home page OsClass loads with search bar, category dropdown, and "Login" option visible
2	Click "Login" button	Login page appears with email and password fields
3	Enter email "blake.sullivan@gmail.com" and password "Password.123", check "Remember me", click "Log in"	User is logged in, homepage reloads with "My account" and "Logout" options visible
4	Enter "camera" in keyword field, select "Photo + video"	Search results page loads with Photo + video listings
5	Enter "Falls Church" in City field, then click "Apply"	Search results page loads with Photo + video Falls Church listings
6	Click on "Nikon N50 Camera" in the listing	Detailed listing page opens showing item specifics, price ($30.00), and seller information
7	Scroll to comment section, enter title "Nice camera" and comment "Nice product", click "Send"	Green notification bar appears stating "Your comment has been approved"
8	Scroll through listing page to check if your comment is visible	A comment "Nice camera by Blake Sullivan" is visible
9	Delete the comment "Nice camera by Blake Sullivant"	Comment list is empty

Fig. 1. An E2E test case for the Classified Application

Our results show that it is possible to transform autonomous agents from the web automation domain so that they can automatically execute tests without any prior knowledge of the application to be tested: more than the half of the tests of our benchmark are successfully run by the agent as they were run by a manual tester. However, a range of issues still prevent web agents from reliably performing end-to-end tests, notably, their inclination toward misinterpreting actions or assertions.

2   Test and Automation for Web Applications

This section begins by introducing an example of an end-to-end (E2E) test case, which will serve as a continuous reference throughout the article. The example is used to illustrate key concepts and methodologies discussed in subsequent sections. The section proceeds by demonstrating how the E2E test case can be automated by generating executable code, leveraging a framework like Playwright [1] to facilitate browser manipulation and interaction. Finally, the section concludes with a discussion on the use of autonomous agents, specifically designed for autonomous task execution on the web, and explores how these agents can be utilised to automate and execute test cases effectively.

2.1   Manual E2E Testing

An E2E test case simulates a complete user scenario to verify that the workflow functions as expected within a web application. It describes the scenario sequentially, specifying the test actions required at each step, as well as any assertions to validate the outcome by observing the application's state. For simplicity, an E2E test case can be represented as a two-column table: the left column outlines the test actions, and the right column lists the test assertions. Each row corresponds to a step in the scenario, and it is possible for some steps to omit test assertions. Figure 1 illustrates an example of an E2E test case designed for a marketplace application[2]. The objective of this nine-step test is to ensure that a user can successfully log into the application, search for a product using a keyword, select the product, and add a comment.

E2E test cases are frequently performed manually by human testers. At each step, the tester must carry out the specified test action and verify that the observed result matches the expected result. If the tester is unable to execute an action or confirm the expected result for any step, that step is marked as a failure, and consequently, the entire test is considered to have failed. Conversely, if the

---

[1] https://playwright.dev/
[2] https://osclass-classifieds.com/

tester successfully completes all steps, performing all test actions and confirming that the results match the expected results, the test is marked as passed.

2.2 E2E Testing Automation

The manual execution of E2E test cases can be both time-consuming and prone to errors. To address these challenges, the automation of E2E tests has been explored as a solution [10]. Automating an E2E test involves generating executable code to carry out all the steps defined in the test scenario. This includes automating both the test actions with the web application and the verification of expected results. To facilitate this process, several frameworks specifically designed for web application automation have been developed [2, 12].

The code in Listing 1 demonstrates the automation of the first three steps of the E2E test case illustrated in Figure 1. In this example, the Playwright web automation framework is used to interact with the web application. For instance, line 16 shows how a click event on the login button is automated. Additionally, the code demonstrates how test assertions are implemented. For example, lines 11, 12, and 13 validate the presence of three critical web elements—the search bar, category drop-down list, and login options, which confirm that the homepage has successfully loaded.

Listing 1. Excerpt of the source code automating the test case presented in Figure 1

```
1   // Step 1: Start on the Homepage
2   await page.goto('http://www.vtaas-benchmark.com:9980/');
3
4   // Verify homepage loaded correctly
5   await page.waitForSelector('input[name="sPattern"]'); // Search bar
6   await page.waitForSelector('select[name="sCategory"]'); // Category dropdown
7   await page.waitForSelector('a:has-text("Login")'); // Login option
8
9   // Step 2: Click "Login" button
10  await page.click('a:has-text("Login")');
11
12  // Step 3: Enter email, password, check "Remember me", click "Log in"
13  await page.fill('input[name="email"]', 'blake.sullivan@gmail.com');
14  await page.fill('input[name="password"]', 'Password.123');
15  await page.check('input[name="remember"]');
16  await page.click('button:has-text("Log in")');
17
18  // Verify user is logged in
19  await page.waitForSelector('a:has-text("My account")');;
```

It is important to note that the source code of an automated test is inherently tied to a specific version of the web application being tested. This dependency arises from assumptions the test makes about the Document Object Model (DOM) structure of the application, which are used to uniquely identify the web elements with which it interacts. For instance, on line 12, the test assumes that the search bar can be located using the CSS selector input[name="sPattern"]. Similarly, on line 16, it relies on the assumption that the login button can be identified by the CSS selector a ("Login").

These assumptions render automated tests fragile, as they are often invalidated by changes in the web application's structure over time [17]. For example, the CSS selectors for the search bar and login button may be altered in future versions of the application. As a result, maintaining the automated test code to accommodate these changes can be both time-consuming and costly [22], introducing significant overhead in terms of maintenance and additional resource requirements.
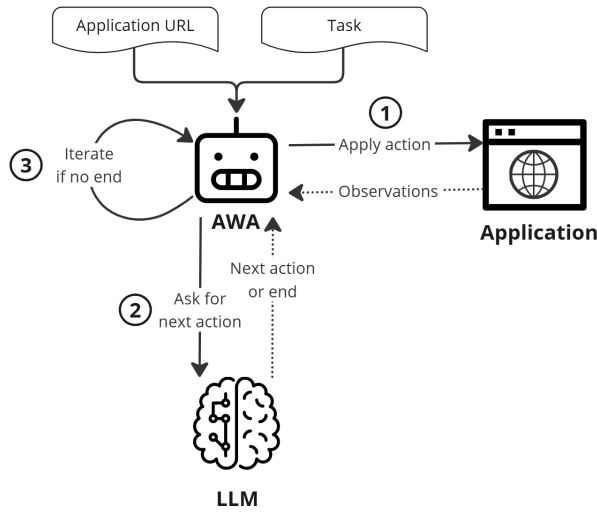
Fig. 2. Autonomous Web Agent main process

2.3 Autonomous Web Agents

Recent advancements in the domain of autonomous agents present promising opportunities for addressing long-standing challenges in test automation, particularly regarding the fragility of test code. An autonomous agent is an entity designed to perform tasks independently, capable of interacting with its environment through actions, and utilising the capabilities of LLMs to interpret its surroundings without prior knowledge [35]. The agent determines the necessary actions to achieve its assigned task based on this understanding. In the context of web applications, this entails executing tasks such as ordering an item from an online store, booking a trip, or searching for information on a website.

Existing approaches that implement AWAs typically take a task description in natural language and the base URL of the web application as input. These approaches generally follow a structured iterative process as presented in the 2. In each iteration, (1) the agent performs the most appropriate action on the web application (in the first iteration, this action involves opening the application at the base URL) and then observes the environment by capturing some elements such as screenshots and/or the DOM source code; (2) the agent analyses these elements, interpret the task requirements, and infer the next appropriate action or determine whether the task has been completed. To do this, the agent constructs one or several prompts that include relevant questions for the LLM, as well as all necessary data, such as observations, task details, and previously performed actions; (3) based on the LLM's response, the agent decides whether to initiate a new iteration or terminate the process.

AWA approaches offer significant potential for applications in test automation for web applications. In this context, the task assigned to the agent can be interpreted as executing a test scenario. For example, an agent could be tasked with executing the test outlined in Figure 1. The agent would autonomously analyse the test, observe the application to identify the necessary actions, and perform the required assertions. This approach enhances the robustness of the test automation process by reducing the vulnerability of changes in the web application's structure.

3   From Autonomous Web Agent to Autonomous Test Agent

This section starts by highlighting the key differences between AWA and ATA. It then presents the 4 main modules of a classical AWA architecture and details the main modifications that should be applied to them for transforming an AWA into an ATA. Finally, it presents two ATAs we developed. The first one, SeeAct-ATA, is a basic ATA that relies entirely on a LLM and is based on the SeeAct AWA [45]. The second, PinATA, is a more advanced ATA, incorporating state-of-the-art techniques for perception, reasoning, evaluation, and grounding [28].

3.1   Main Differences Between AWAs and ATAs

While the execution of an E2E test case can be considered a task in its own right, several unique features differentiate it from conventional tasks, requiring modifications to an AWA to transform it into an Autonomous Test Agent (ATA).

The first critical distinction between an E2E test case and a typical task is the strict sequence of actions that must be followed. An E2E test case outlines a predefined scenario consisting of a series of steps that must be executed in the correct order, like in Figure 1. Unlike an AWA, which focuses on achieving the overall task objective, regardless of the actions taken or their order, an ATA is constrained by the test scenario and must strictly adhere to the specified sequence of steps.

The second key feature of E2E testing is the requirement for multiple intermediate assertions throughout the process. If any of these assertions fail, the entire test is deemed unsuccessful. This behaviour differs from that of standard AWAs, which aim to complete the task without performing intermediate validations. Despite these two key differences, ATAs and AWAs share many fundamental similarities. Both agents must be capable of executing actions, observing their environment, and using this information to guide their decisions.

3.2   AWA Architectures

Wang et al. [35] define the architecture of an AWA as consisting of four fundamental modules: profile, memory, planning, and action. The profile module specifies how the agent interacts with the LLM, including its role and self-description, all of which influence the LLM's responses [32]. The memory module manages the information necessary for the task execution, encompassing both data collected from observing the application and external knowledge retrieved from the internet. The planning module serves as the agent's reasoning component, enabling decision-making for selecting subsequent actions. Finally, the action module defines the set of tools available to the agent to interact with the application and its broader execution environment.

Adapting an AWA into an ATA requires modifications to all four modules. Specifically, the profile module must explicitly define the agent's role as a tester, with the primary objective of executing a predefined test scenario. The memory module should retain a record of all performed test steps, as future steps may rely on this information. For example, certain tests require verifying that a value obtained at a specific step matches one retrieved earlier; the memory module should facilitate such verifications. The planning module must strictly enforce adherence to the test scenario, ensuring that all actions are executed in the specified order. The action module should not only support the test actions with the application but also the performing of assertions. The two following subsections present how we realised these modifications to build two ATAs.

3.3   SeeAct-ATA

We first choose to develop a very basic ATA that serves as a baseline. We opted to develop it by adapting the SeeAct AWA [45] since it uses a very elementary architecture where the 4 main modules are encoded into a prompt, relying mainly on the LLM for most of the work, making it easy

to adapt as an ATA. This prompt contains multiple questions to guide the LLM in proposing the next action required to complete the given task. A simplified version of this prompt is presented in Listing 2. The prompt begins by defining a human user profile. Next, relevant data is integrated into the prompt, including the task to be completed, the previous actions undertaken, and the current screenshot of the application. Following this, several instructions are provided, utilising the Chain of Thought [36] approach – prompting process by delineating complex tasks into a sequence of logical steps towards a final resolution –, which has been shown to enhance the LLM's reasoning and output quality. While the full set of instructions is not detailed here, the prompt's goal is to have the LLM describe the web page, summarise the previous actions and the screenshot, and then propose the next action. The LLM is also instructed to align its proposed action with the interactive elements identified in the DOM, as provided in the observations. Finally, the LLM returns the result in a specific format. The action 'TERMINATE' allows the LLM to indicate that the task has been completed. If not completed, the LLM returns the next action to be performed. The SeeAct AWA then relies on Playwright to perform this action.

Listing 2. Prompt used by seeAct to ask the LLM for the next step

```
Imagine that you are imitating humans doing web navigation. You need to decide on the first
    following action to take.

You are asked to complete the following task: {{ TASK }}

Previous Actions: {{ PREVIOUS_ACTIONS }}

The screenshot below shows the webpage you see. {{ SCREENSHOT }}

Follow the following guidance:

[CURRENT WEBPAGE IDENTIFICATION]: Firstly, think about what the current webpage is.

[PREVIOUS ACTION ANALYSIS]: Secondly, combined with the screenshot, analyse each step of the
    previous action history and their intention one by one.

[SCREENSHOT DETAILS ANALYSIS]: Closely examine the screenshot to check the status of every part of
     the webpage to understand what you can operate with and what has been set or completed.

[NEXT ACTION BASED ON WEBPAGE AND ANALYSIS]: Then, based on your analysis, decide on the following
     action. And clearly outline which element will operate with as the first next target element,
     its detailed location, and the corresponding operation.

[MULTICHOICE QUESTION]: Below is a multi-choice question, where the choices are elements in the
    webpage. Determine whether one matches your target element.
If none of these elements match your target element, please select N. None of the other options
    match the correct element.
A. <a>Jump to sidebar</a>
B. <a>Jump to main content</a>
C. <a>Go to HomePage</a>
[...]
N. None of the other options match the correct element

[FINAL ANSWER]: Finally, conclude your answer using the format below.
    ELEMENT: The uppercase letter of your choice. (No need for PRESS ENTER)
    ACTION: Choose an action from {CLICK, SELECT, TYPE, PRESS ENTER, TERMINATE, NONE}.
    VALUE: Provide additional input based on ACTION.
```

To transform SeeAct into an ATA, we modified its prompt to instruct the LLM to propose the next action according to the specifications of the test scenario and to perform verification assertions. Listing 3 highlights the key differences between the original SeeAct prompt and the revised version we developed to adapt it into an ATA. First, we defined a new profile representing a manual tester to better align with a test execution mindset. Next, we set the E2E test case as the task to be executed. Additionally, we introduced two new instructions:

- *Test Case Progress:* This instruction directs the LLM to identify which steps of the test have already been executed and determine the current step. This helps focus the LLM on the specific part of the test that needs to be executed.
- *Test Step Assertion Control:* This instruction asks the LLM to perform the assertions specified in the current test step.

The rest of the prompt remains unchanged from the original SeeAct configuration that can be found on their Github repository [3].

Listing 3. Our ATA prompt with main differences with the AWA prompt highlighted in blue

```
Imagine that you are imitating a manual tester performing a test case.You receive a test case and
    you should follow it and determine its verdict.

You are asked to complete the following task: {{ TEST }}

Previous Actions: {{ PREVIOUS ACTIONS }}

The screenshot below shows the webpage you see.

[CURRENT WEBPAGE IDENTIFICATION]: ...

[PREVIOUS ACTION ANALYSIS]: ...

[TEST CASE PROGRESS]: Based on the past actions, determine the state of the test case.Go through
    each step, one by one, and give it a status: DONE, CURRENT, TODO.Finish by confirming the
    current test step.

[Screenshot Details Analysis]: ...

[TEST STEP ASSERTION CONTROL]: based on past actions, if any, control the test step assertion. Make
    a numbered list of all the atomic assertions and, for each one, based on what you see in the
    screenshot, assign a status VERIFIED, NOT VERIFIED. If all assertions are verified, write the
    sentence: All assertions have been verified

[NEXT ACTION BASED ON WEBPAGE AND ANALYSIS]: ...

[MULTICHOICE QUESTION]: ...

[FINAL ANSWER]: ...
```

Our transformation of SeeAct into an ATA, named SeeAct-ATA, is intentionally basic and minimal to evaluate the feasibility of using autonomous agents in a test automation context. This streamlined adaptation allows us to explore how effectively autonomous agents can handle test cases while maintaining the core functionality of the original AWA architecture.

3.4 PinATA

Our second ATA, called PinATA for **P**lanned **IN**centive ATA is more advanced than the first one. We picked up from the literature approaches that we believe are well suited to comply with the

---

[3] https://github.com/OSU-NLP-Group/SeeAct

specific objectives of an ATA. Unlike our first ATA, which relies on a single prompt, PinATA is composed of three specialised components: the orchestrator, the actor, and the assertor.

The orchestrator oversees the entire process, delegating action execution to the actor and verification tasks to the assertor. According to the four modules architecture, the orchestrator corresponds to the planning module, while the actor and assertor together form the action module. They all share a global memory module and further have a similar profile module.

The orchestrator is leading the process employing a *Planning with Feedback* strategy [35]. It maintains a model of the test scenario and monitors its execution progress. At each step, it instructs the actor to perform the corresponding action. Based on the actor's feedback, the orchestrator determines whether the action was successfully executed. If not, it requests a retry until either the action is completed or deemed unfeasible, in which case the test is marked as failed. Once an action is executed, the orchestrator delegates the verification task to the assertor. Similarly, it evaluates the assertor's feedback to determine whether the assertion was successfully performed. If necessary, it requests a retry. Depending on the outcome of the verification, the orchestrator either proceeds to the next step or concludes that the test has failed.

The actor, tasked with executing actions, identifies the relevant elements of the target application for the test action—an operation known as *grounding*, which remains a key challenge in AWA research [4, 14]. Our implementation leverages the *Set-Of-Marks* technique [39] and uses the LLM's capability to determine X and Y coordinates from screenshots. The actor component uses the Playwright framework for interacting with the browser.

The assertor is responsible for performing verifications, employing the *Agent as a Judge* approach to harness the LLM's evaluative capabilities [47]. Our implementation asks the LLM to perform an evaluation just by analysing the screenshot of the application.

All three components (orchestrator, actor and assertor) share a global memory system that follows a *long-term memory* approach for the memory module [43], storing all processed information. Notably, the system does not retrieve external data from the internet; it relies solely on observations of the target application.

Finally, all three agents share a similar profile module, presenting themselves as part of a multi-agent system while clearly defining their specific roles.

PinATA incorporates what the authors' consider to be a representative selection of state-of-the-art approaches from existing AWAs. However, we recognise that the design space for ATAs is vast, necessitating systematic evaluation methodologies. To address this need, we introduce in the following section a comprehensive benchmark designed to enable objective comparison of different ATA implementations.

4 A Benchmark for Assessing ATA Effectiveness

This section presents the fully reproducible benchmark developed to objectively evaluate ATA effectiveness in executing E2E test cases originally designed for manual execution. It begins by outlining the motivations for defining a new benchmark, followed by presentation of the three web applications serving as the system under test and the test cases. It ends with a description of the metrics used to measure effectiveness.

4.1 Why Yet Another Benchmark?

Several benchmarks have been developed to evaluate AWA effectiveness. Famous examples include Mind2Web, which defines 2,350 tasks across 137 websites to assess how well AWAs can accomplish them [8]; WebArena, which provides a standalone, self-hostable web environment with hundreds of tasks to measure AWA performance [46]; the TheAgentCompany, which provides more complex tasks [38]; and AgentQuest, which offers a similar benchmark but incorporates more flexible

evaluation metrics [13]. Qin et al. [28] provide a comprehensive analysis of existing benchmarks designed for AWA evaluation.

All of these benchmarks define a set of tasks that AWAs must complete, making them unsuitable for evaluating ATAs. As discussed in Section 2, ATAs take as input a test case written in natural language, which consist of sequential steps and are not directly comparable to standalone tasks. This fundamental difference motivated the development of a new benchmark specifically designed for ATA evaluation.

We observed that web applications used in existing benchmarks are either public applications, which are in production and thus subject to uncontrollable modifications, or packaged applications, which can be deployed as-is and fixed in time. Given the importance of reproducibility, we opted for the latter approach and leveraged three applications provided by the WebArena benchmark.

AWA benchmarks also highlight the importance of using real-world data, particularly for the tasks they evaluate. Following this principle, we incorporated test cases written by professional testers into our benchmark to ensure its realism and applicability.

Finally, we developed new metrics specifically tailored to measure ATA effectiveness, as existing AWA metrics focus on task completion rather than test execution.

As a summary, our benchmark has been developed following established guidelines [33], ensuring its suitability and replicability for future studies. It includes realistic and fully deployable web applications, including code and data to enable reproducibility. It includes realistic passing and failing end-to-end test cases associated with a verdict given by expert testers. It defines a comprehensive set of metrics that allows the assessment and comparison of ATA effectiveness.

4.2 Web Applications

Our benchmark contains three web applications previously utilised in the Visual Web Arena benchmark [46]. Each provides a unique environment for diverse test cases:

- **Classifieds**: Inspired by real-world marketplaces like Craigslist and Facebook Marketplace, this site includes 65,955 listings and offers a distinct environment focused on marketplace-style user test actions.
- **Postmill**: A social networking site inspired by Reddit, allowing users to create comments on various topics organised into subsections.
- **OneStopShop**: A shopping site featuring extensive search filters for a wide array of products, with capabilities for account creation, shopping cart management, and order placement.

This choice, as outlined in the previous subsection, is driven by our primary goal of creating a fully reproducible benchmark. Each web application is provided as a Docker image, allowing for easy deployment. Even though the web applications of our benchmark target popular domains (marketplaces, social networking, and e-commerce), incorporating a larger set of web applications would further strengthen the benchmark. However, we feel that this first core set of applications constitutes a good first milestone to evaluate ATA. We hope that open-sourcing the benchmark and the models will allow the growth of this core through community-driven efforts.

4.3 Test Cases

We engaged four professional testers to design test cases for the three web applications in our benchmark. Each tester has over five years of experience in web application testing and holds an ISTQB [4] Foundation Level certification. We instructed them to design passing test cases and document them using a two-column matrix, as illustrated in Figure 1. Each test case comprises multiple steps, detailing specific actions along with optional assertions. Each application was assigned a

---

[4] https://www.istqb.org/

▶ TC-4-F :: Login, search by keyword & category, choose an add, and cancel the comment before validation	
# Actions	Expected Result
1 Start on the Homepage: http://www.vtaas-benchmark.com:9980/	Home page OsClass loads with search bar, category dropdown, and "Login" option visible
2 Click "Login" button	Login page appears with email and password fields
3 Enter email "blake.sullivan@gmail.com" and password "Password.123", check "Remember me", click "Log in"	User is logged in, homepage reloads with "My account" and "Logout" options visible
4 Enter "camera" in keyword field, select "Photo + video" category, then click "Search"	Search results page loads with Photo + video listings
5 Enter "Falls Church" in City field, then click "Apply"	Search results page loads with Photo + video Falls Church listings
6 Click on "Nikon N50 Camera" in the listing	Detailed listing page opens showing item specifics, price ($30.00), and seller information
7 Scroll to comment section, enter title "Nice camera" and comment "Nice product", click "**Cancel**"	**Title and comment fields should be empty**
8 **Scroll through listing page to check if the comment is not displayed**	**The new comment is not visible in the comment section**

Fig. 3. A failing test case obtained by modifying the test case presented in fig 1. Changes appear in bold red.

dedicated tester, except for the *Classified* application, which was tested by two professionals. Over three weeks, the testers designed a total of 62 test cases.

After receiving the test cases, we applied several post-processing steps to ensure consistency and reliability. First, we verified that all test cases were self-contained, including any required data (e.g., login credentials) to facilitate successful execution. We assumed that each test case would be executed on a freshly deployed web application to prevent interference from previous test runs. Additionally, we ensured that all test cases were written in English to enhance the benchmark's usability. Finally, one of the authors manually re-executed each test case to validate its correctness.

Considering the time required for test case design by professional testers—along with additional efforts such as translation and manual execution—each successfully passing test case represents an average investment of approximately one hour.

To integrate non-passing test cases into our benchmark and ensure a balanced dataset, two authors collaborated to derive failing test cases from existing passing ones. One of these authors is an experienced professional tester who verified the correctness and industry compliance of the generated tests. Failing test cases were constructed by modifying passing ones to check for non-implemented features within the application. As shown in Figure 3, we adapted the passing test case from Figure 1 to create a corresponding failing version. In this example, the cancel button is absent, making it impossible to click. A correctly functioning ATA should halt execution at this point rather than proceeding further. Each failing test case includes an explicitly annotated expected failure step, enabling precise metric calculations during execution. Out of the 62 passing test cases, 51 corresponding failing test cases were developed. The discrepancy arises because some passing tests were highly similar, leading us to generate a single failing test case for such groups. For instance, the passing tests P19, P20, and P21 in OneStopShop take place on the same web page and differ only slightly in their final actions. Instead of creating separate failing test cases for each, we produced a single representative failing test case.

Our benchmark ultimately comprises 62 passing test cases, crafted by professional testers, and 51 failing test cases manually derived from the passing test cases. These test cases are distributed across the three selected websites, as detailed in Table 1.

4.4 Metrics

In this section, we present a set of metrics for evaluating the performance of ATAs against human testers. Our metrics focus on two key aspects: verdict alignment and step alignment.

Table 1. The distribution of the 113 test cases split among the 3 chosen websites

Application	Application Domain	Passing Tests	Failing Tests
Classified	Shopping	15	15
Postmill	Social Network	18	16
OneStopShop	Flea Market	29	20

Verdict Alignment. Our metrics are defined based on evaluating an ATA's judgment as a binary classification problem. This approach models test case evaluation by defining two classes:

- *Negative class:* a test case passes according to manual evaluation,
- *Positive class:* a test case fails according to manual evaluation.

To assess the alignment between human and agent verdicts, we define the following outcomes:

- *True Positive (TP):* Both human and agent identify the test case as failing. This represents the correct identification of a failing test case.
- *True Negative (TN):* Both human and agent identify the test case as passing. This represents the correct identification of a passing test case.
- *False Positive (FP):* The agent identifies a test case as failing, but the human evaluator deems it passing. This is a Type I error, potentially leading to unnecessary debugging efforts.
- *False Negative (FN):* The agent identifies a test case as passing, but the human evaluator finds it failing. This is a Type II error, potentially allowing bugs to go undetected.

We then use a set of traditional metrics to evaluate how the results from an agent align with the manual results:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \qquad \text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} \qquad \text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Accuracy provides an overall measure of how often the agent's verdict aligns with the human judgment. To provide a more nuanced evaluation, we also consider specificity that measures the agent's ability to correctly identify passing test cases. A high specificity indicates that the agent is effective at avoiding false alarms. Finally, sensitivity, also known as recall, assesses the agent's performance in detecting failing test cases. A high sensitivity suggests that the agent is effective at identifying discrepancies between test case and application when they are present.

Step Alignment. Note that an agent can identify a test as failing similarly to a human, but not for the same underlying reason (i.e. not failing on the same assertion). This would be a type III error [30] where the verdict is correct but for a wrong reason. To account for this possible behaviour, we define a second set of metrics based on the recorded step index where the failure is observed. We decompose the true positives into three categories :

- **AFB**: **A**gent **F**ails **B**efore human represents the number of cases where the agent fails at an earlier step than the human.
- **AFA**: **A**gent **F**ails **A**fter human represents the number of cases where the agent fails at a later step than the human.
- **AFC**: **A**gent **F**ails **C**orrectly represents the cases where the agent fails at the same step as the human.

With this decomposition, we introduce four novel metrics: the Automation Error Rate (AER), the Hallucination Error Rate (HER), the Step Mismatch Error Rate (SMER), and the True Accuracy (TruAcc):

$$AER = \frac{AFB}{TP} \qquad HER = \frac{AFA}{TP} \qquad SMER = AER + HER \qquad TruAcc = \frac{AFC + TN}{TP + TN + FP + FN}$$

The AER rate describes the number of time the agent failed before the human without even having seen the failing step. These errors are likely due to the agent's limitation in exploring the website or validating an assertion.

Hallucinations are produced outputs that are coherent and grammatically correct but factually incorrect or nonsensical. In that case, hallucinations would be unlawfully validated step assertions and the HER quantify them.

The SMER metric helps quantify all the Type III errors by adding the two metrics introduced before. It provides insight into how often the agent arrives at the correct verdict (test case failure) but for the wrong reasons. A low SMER indicates a good alignment between the agent's bug identification process and human evaluation, while a high SMER suggests that the agent may be prone to failures due to its limitations rather than actual bug detection.

Finally, the TruAcc which, unlike the initially proposed accuracy, takes into account detectable Type III errors of the ATA to represent only verdicts.

Based on our defined metrics, an ideal ATA should achieve perfect accuracy, specificity, and sensitivity while maintaining a SMER of zero. However, in practice, trade-offs between these metrics may arise. We prioritise sensitivity over specificity, as failing test cases should not go unnoticed, whereas false positives can still be manually reviewed. Additionally, SMER serves as a failsafe metric to assess whether high accuracy is genuinely meaningful. A high accuracy score accompanied by a high SMER is problematic, as it indicates that the ATA, despite matching human tester results, does not interpret assertions in the same way as a human tester.

5 Experiments

This section presents the evaluation of our benchmark by assessing the two ATAs introduced in Section 3. First, we compare the SeeAct-ATA with PinATA, using GPT-4o, given its strong performance. As expected, our results indicate that PinATA outperforms SeeAct-ATA, achieving a 50% increase in true accuracy. Next, we focus on PinATA and evaluate its performance across different LLMs (GPT-4o, Sonnet, and Gemini) to analyse their impact. While no major differences are observed, results show a 6% performance variation between the best and worst-performing models. Finally, we conduct a qualitative analysis of test cases that remain infeasible for PinATA, regardless of the LLM used. This analysis aims to identify key challenges and potential areas for improvement in autonomous test execution. All code and data used in this experiment are available on GitHub [5].

5.1 SeeAct-ATA vs PinATA with GPT-4o

We evaluated both the SeeAct-ATA and PinATA by executing the 113 test cases from our benchmark across our three target applications. To ensure consistency, we reset each system under test to its initial state before every execution, preventing interference from prior runs. For all executions, both agents were connected to GPT-4o from OpenAI, configured with a temperature of 0 to enhance determinism, though slight variations in responses remained possible.

At each execution, the agent produces a verdict, indicating whether the test passes or fails. In the case of a failure, the agent specifies the failed step and identifies the cause—whether it stems from an action or a verification error. Additionally, our execution framework records the entire execution trace of the agent. This trace helps pinpoint the exact step where execution was interrupted if

---

[5]https://github.com/Smartesting/pinata

any. The collected data, including verdicts and execution traces, enables the computation of our evaluation metrics, detailed in Table 2.

Table 2. Performance Comparison between SeeAct-ATA and PinATA using GPT-4o.

Architecture	Application	Evaluation Metrics						
		Acc	Spec	Sens	AER	HER	SMER	TruAcc
SeeAct-ATA	Classified	0.50	0.33	0.67	0.30	0.15	0.45	0.20
	Postmill	0.56	0.67	0.44	0.13	0.07	0.20	0.47
	OneStopShop	0.59	0.76	0.35	0.17	0	0.17	0.53
	Average	0.55	0.59	0.48	0.20	0.07	0.28	0.40
PinATA	Classified	0.64	0.47	0.81	0.08	0.08	0.15	0.48
	Postmill	0.76	0.61	0.94	0.00	0.06	0.06	0.71
	OneStopShop	0.73	0.62	0.90	0.09	0.02	0.11	0.63
	Average	0.71	0.57	0.88	0.06	0.05	0.11	0.61

Our results demonstrate a clear performance gap between the two ATAs, with PinATA significantly outperforming the SeeAct-ATA. Notably, the true accuracy metric is substantially higher for PinATA, showing an improvement of over 50% on average (0.61 vs. 0.40). This improvement is largely attributed to the reduction of Type III errors, as reflected in the SMER metric, which is 60% lower for PinATA (0.11 vs. 0.28). Additionally, PinATA exhibits a much higher sensitivity (0.88 vs. 0.48, an 80% increase), demonstrating its superior ability to correctly identify failing test cases. Finally, specificity remains comparable between both ATAs, indicating that both agents have a tendancy to generate false failure alerts.

5.2 GPT-4o vs Sonnet vs Gemini

We leveraged our benchmark to assess the impact of different LLM choices on the design of PinATA. Specifically, we executed the 113 test cases from our benchmark using three commercial high-performing LLMs: GPT-4o, Sonnet, and Gemini (see Table 3). Although additional LLMs could be integrated, our primary goal was to demonstrate the benchmark's usefulness to compare LLMs rather than conducting an exhaustive comparison.

Table 3. The size of the 3 LLMs used in this experiment (" * " are indications, as the exact number of parameters is not confirmed by their providers).

Backbone LLM	LLM size	Provider	Licence
GPT-4o	200B*	OpenAI	Proprietary
claude-3-5-sonnet	175B*	Anthropic	Proprietary
gemini-2.0-pro	Unknown	Google	Proprietary

Our results reveal only minor differences across the three LLMs (see Table 4), with no single model emerging as significantly superior. True accuracy remains relatively stable across models (0.61 vs. 0.58 vs. 0.62), and the SMER metric exhibits minimal variation (0.11 vs. 0.12 vs. 0.10). The only notable distinction is Sonnet's superior sensitivity (0.94 vs. 0.88 for GPT-4o and 0.85 for Gemini), indicating a lower rate of undetected failing test cases when using Sonnet. However Sonnet has a lower specificity (0.47 vs 0.57 for GPT-4o and 0.56 for Gemini).

Table 4. Performance Comparison of GPT-4o, Sonnet and Gemini with PinATA.

Model	Application	Evaluation Metrics						
		Acc	Spec	Sens	AER	HER	SMER	TruAcc
GPT-4o	Classified	0.64	0.47	0.81	0.08	0.08	0.15	0.48
	Postmill	0.76	0.61	0.94	0.00	0.06	0.06	0.71
	OneStopShop	0.73	0.62	0.90	0.09	0.02	0.11	0.63
	Average	0.71	0.57	0.88	0.06	0.05	0.11	0.61
Sonnet	Classified	0.60	0.27	0.93	0.07	0.07	0.14	0.47
	Postmill	0.71	0.44	1.00	0.06	0.03	0.09	0.62
	OneStopShop	0.78	0.69	0.90	0.09	0.05	0.14	0.65
	Average	0.69	0.47	0.94	0.07	0.05	0.12	0.58
Gemini	Classified	0.70	0.47	0.93	0.07	0.04	0.11	0.60
	Postmill	0.62	0.44	0.81	0.04	0.04	0.07	0.56
	OneStopShop	0.78	0.76	0.80	0.10	0.03	0.13	0.71
	Average	0.70	0.56	0.85	0.07	0.03	0.10	0.62

Beyond the quantitative results, this experiment highlights that the choice of LLM influences the verdict produced during test case executions. Figure 4 presents the outcomes of all test case executions across the three selected LLMs, distinguishing between executions that align with the verdict of the human tester (i.e. stopping at the same step with the same verdict) and those that diverge. Notably, our analysis reveals 26 test cases for which PinATA is not aligned with the verdict of the human tester, regardless of the LLM used. These 26 test cases serve as the foundation for our subsequent qualitative analysis, aiming at identifying the underlying challenges and potential areas for improvement.

5.3 ATA Limitations Taxonomy

To gain deeper insights into the limitations of the ATA, we conducted a qualitative analysis. Our analysis is based on our second experiment, and focused on the executions where, regardless of the selected LLM, the verdict provided by PinATA differed from the one of the professional tester (26 executions out of 113). By isolating these cases, we aimed to identify the intrinsic limitations of PinATA, considering them as potential research directions to advance the field.

Two co-authors of this article manually analysed the 26 executions and categorised the errors using an open card-sorting approach [48]. In this process, the authors independently grouped errors based on perceived similarities and then reached a consensus on the final set of categories through collaborative discussion. This analysis resulted in five categories: Action Capacity, Action Versatility, User-Interface Observability, Assertion Verifiability and Scenario Conformity. The first three categories illustrate errors also encountered by AWAs, as they stem from challenges in interacting with the web application. In contrast, the last two categories are specific to ATAs.

Action Capacity (6 errors out of 26). This category encompasses errors caused by limitations in the technical framework for interacting with the web browser, which restrict the agent's ability to perform certain actions. For instance, the Classified test cases P11 and P12 involve opening a second browser tab, while test case P20 requires accessing the page print options, and test case P7 necessitates opening the browser settings—none of which are currently supported by our agent. Addressing these errors requires a technical framework that provides a broader range of services while ensuring they are easily accessible to the agent.
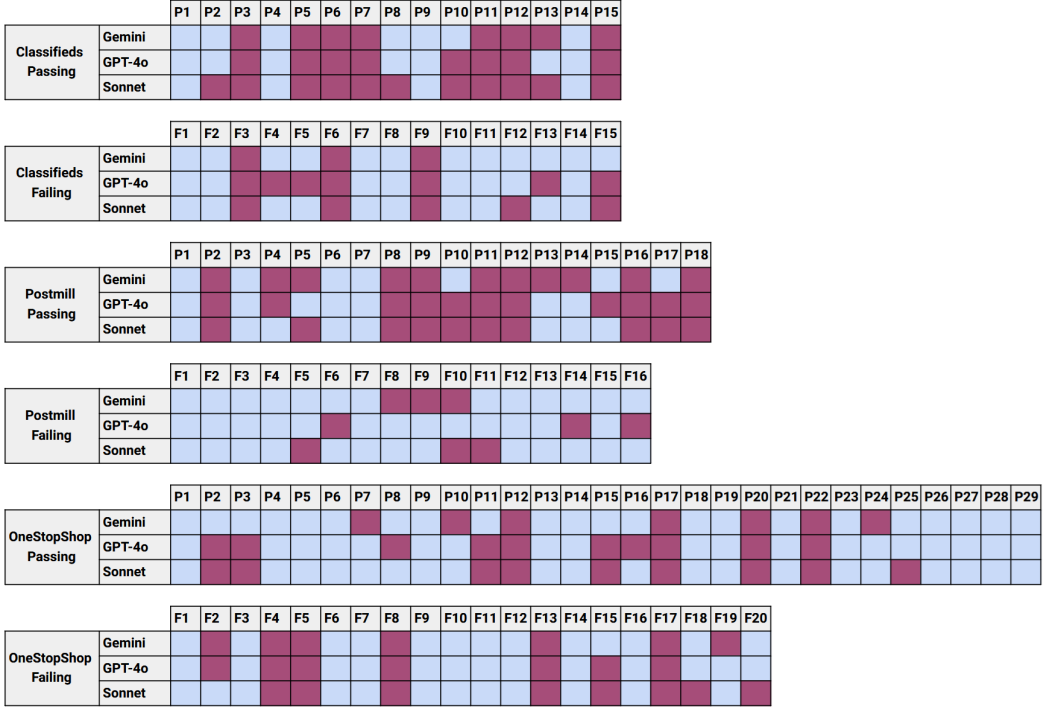
Fig. 4. Executions of the 113 test case with PinATA with three LLMs. Each cell represent an execution by a model. If the cell is in light blue, then the returned verdict is aligned with the one of the human tester. Otherwise, the cell is in purple.

Action Versatility (6 errors out of 26). This category includes errors resulting from the agent's inability to perform complex graphical interface manipulations that cannot be executed directly using the actions provided by the technical framework. For instance, in the Classified application, test case P3 requires entering a location letter by letter to trigger an auto-complete suggestion. However, our agent inputs the entire location at once, disrupting the system and preventing further execution. Figure 5 illustrates this issue, where the location "Rochester" must be typed incrementally to select the suggestion "Rochester (Pennsylvania)." Another example of the lack of action versatility can be found in F5 of the OneStopShop application where the model does not properly navigate to sub-categories of items due to dropdown menus. Addressing these errors necessitates the development of refined strategies for interacting with the graphical interface. These strategies should dynamically adapt based on the current step and the interface's responses to ensure successful execution.

User-interface Observability (5 errors out of 26). This category encompasses errors resulting from the agent's failure to accurately observe the graphical interface, preventing it from recognising the elements it needs to interact with. For instance, in the PostMill test cases P8 and P9, the agent was required to interact with a popup window. However, since popups are not standard HTML elements, our agent—which generates screenshots based on the HTML structure—failed to capture the popup in the screenshot, thereby disrupting the test execution. Another limitation stems from the Set-Of-Mark (SOM) technique we employed, which occasionally lacks precision in distinguishing small
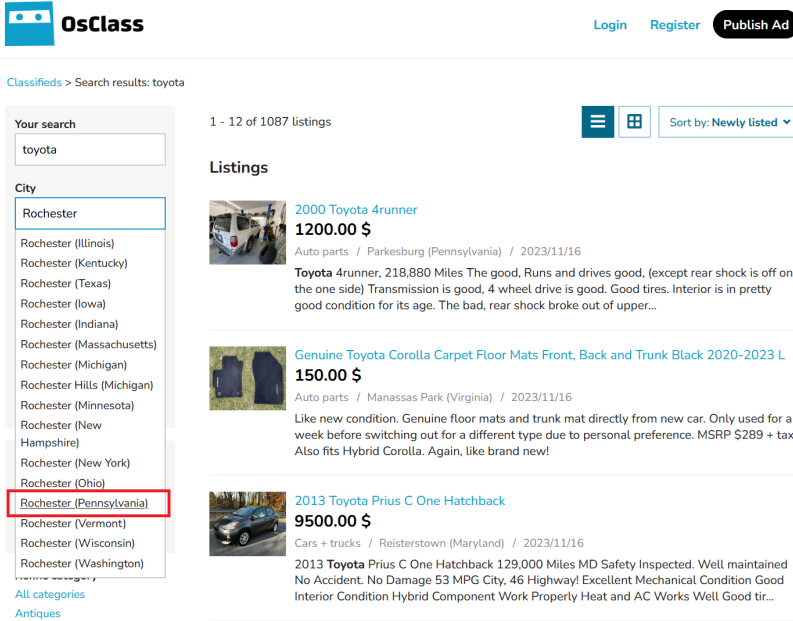
Fig. 5. Screenshot of the Classifieds application showing the completion dropdown menu that need to be used to ensure nominal behaviour.

UI elements. This issue is evident in the Classified P6 test case, where the technique struggled to accurately identify the necessary elements.

Assertion Verifiability (4 errors out of 26). This category encompasses verification errors, which may arise from the agent misunderstanding the verification task, failing to obtain the necessary information, or making an error in analysis. For instance, in test case F9 of the Classified application, the agent was required to verify whether the page displayed elements in a grid format. However, despite the layout not meeting this requirement, the agent incorrectly validated the assertion. Similarly, in test case P11 of application Postmill, the agent is unable to scroll to the bottom of the page to complete its verification. To mitigate these errors, we could enhance the assertor by equipping it with its own specialised action module. It would be capable of not only analysing screenshots but also interacting with elements to confirm their properties. We could also add multi-modal capacities such as DOM analysis. This dedicated module would allow for more thorough and accurate assessments without relying solely on a passive visual analysis.

Test Case Conformity (5 errors out of 26). This category includes errors caused by the agent performing unnecessary actions that introduce side effects, resulting in the test not being followed up. As a result, the intended workflow is not tested and actions and/or assertions are not realised. For instance, in F4 of OneStopShop, the wrong assertion of checking for a red heart is not respected by the models and unnecessary navigations to other pages are wrongfully added. Another example can be found in the P17 test case of OneStopShop where the agent anticipates the validation of a popup message, which was supposed to be done in the next step, resulting in a confused orchestrator. Addressing these errors necessitates the development of a more robust planning module that systematically controls the agent's actions and prevents unintended side effects. For instance, adding future steps in the memory module could limit the anticipation done in some case.

5.4 Threats to Validity

In this section, we discuss the threats to validity according to the taxonomy of Wohlin et al. [37].

Construct validity. We introduced a set of metrics to evaluate to what extent an ATA has similar results compared to a human tester. However, as we explained in Section 4, it is totally possible that an ATA and a human tester obtain a same test verdict by chance (meaning that the ATA has not ensured all the assertions written in the tests), leading to type-3 errors. To mitigate this phenomenon we introduced four special metrics for the failing tests, based upon a comparison of the exit step. Unfortunately, this strategy is not possible for the passing tests since the exit step is necessarily the last. Therefore, the results achieved by ATAs on the passing tests may also subjects to correct verdicts for wrong reasons.

Internal validity. The validity of the test results in our benchmark is the result of manual test execution by the authors. Even if the authors were very careful performing the manual execution, we cannot guarantee that some tests were not misidentified.

Our taxonomy of reasons for discrepancies between PinATA and human tester verdicts is derived from qualitative analysis of the ATA's output data, including prompts, responses, and browsing engine logs. Using this data to determine an underlying reason is a hard task that bear some subjectivity. We mitigated this subjectivity by proceeding to collaborative tagging sessions, but we cannot guarantee that some reasons were not misidentified.

External validity. The results of the experiments are obtained using a limited number of LLM backbones, fondationnaly trained. The results obtained using fine-tuned LLMs or different language models could be different. In addition, the temperature set to the minimum ensured a reproducibility of our results. However, it might have impacted the quality of the answers of the LLM. A more thorough hyperparameter exploration could improve the generation.

While we included web applications from different domains and tests written by professional testers from different companies, the number of tests and applications is still small and therefore our results could be different for other kinds of applications or tests.

Our taxonomy of current limitations of the Advanced ATA has been only based on a small set of cases, we cannot guarantee that this taxonomy is complete. Notably, some other limitations could appear using other web application, tests or ATAs.

6 Related Work

In this section, we present the related work on AWAs and on scripted test automation.

6.1 Scripted Test Automation

Among the most prominent techniques in web test automation are scripted automation and keyword-driven testing. Scripted automation involves manually writing test scripts using programming languages or automation frameworks to simulate user interactions with a web application. Similarly, keyword-driven testing structures test cases using predefined keywords representing specific test actions to structure test scripts that can be executed within automation frameworks [2, 12]. Usually, the testers can add assertions inside the navigation script to ensure the application is correct. Automation scripts are well known to suffer from the test fragility problem [17, 22, 29]. Since scripting actions often rely on CSS selectors to find the relevant elements in the web application to interact with, tiny changes in the application design or the browsing context can prevent the script from running and require manual efforts to fix the tests.

Many approaches have tried to mitigate this web test fragility problem. The first line of approaches aims at automatically repairing a broken test by fixing the broken selector. Historically, the first

approaches used the structure of the DOM [3, 6, 16]. Later, some approaches that use visual elements of the page instead of selectors have been introduced [31]. Empirical results show that DOM-based approaches tend to be more robust and easier to evolve [21]. Another line of approaches targeted the synthesis of robust CSS selectors [23, 24, 27]. Finally, some approaches focus on the record and replay mechanism to improve its efficiency [1], for instance by automatically replaying the script during recording to eliminate fragile actions [26].

6.2 Autonomous Web Agents and Existing ATAs

The emergence of LLMs and their demonstrated potential as agents in many applications [35] has spurred a surge in research efforts aimed at automating web tasks.

6.2.1 Autonomous Web Agents. Initial approaches relied on a textual representation of the application, i.e. the Document Object Model (DOM) of web pages, to define the action space for agents. However, since the DOM can be considerably large, often exceeding 10,000 tokens, techniques were required to filter out irrelevant information and reduce noise. Mind2Act consists of a two-stage process where a task specific small language model (SLM) is first involved to rank DOM elements based on their relevance to the task, selecting a subset of candidates [8]. Then, a LLM such as GPT-4 is queried to predict the next action based on these selected elements. WebAgent also relies on two specialised LLMs [15]. First, HTML-T5 has been specifically trained to receive the entire DOM and extract the most relevant HTML sections, along with planning for actions. Flan-U-PaLM, a general-purpose LLM, then generates Python programs to perform the actions on the application. AutoWebGLM forms its observation space by performing optical character recognition (OCR) on a screenshot of the application and combines it with the output of an HTML simplification algorithm [20]. The combined observation is fed to a relatively small language model (ChatGLM3-6B) to determine the actions to perform. SteP employs a Markov decision process to dynamically compose policies from a library of LLM policies. Each policy provides instructions and examples for a particular subproblem (e.g., searching a list).

LLMs have recently become multimodal by acquiring vision understanding capabilities. It further reinforced their potential as autonomous agents in web automation, as it augmented their ability to perceive complex web environments. Although perception is improved, interaction becomes a bit more complex as it requires grounding between the visual and web elements. With SeeAct, the agent we used as basis for our experiments, the authors experimented with several grounding techniques such as Image annotation (similar to Set-of-marks [39]), textual choices, and element attributes [45]. They report that textual choices is the most efficient grounding method. WebVoyager combines image annotation and textual choices grounding methods to translate the agent intentions into actions [18]. CogAgent takes another route: training a 18B visual language model dedicated to GUI perception and interaction, hence eliminating the grounding problem [18]. Given a task, the model outputs an action and X,Y coordinates. VIPACT leverages multi-agent collaboration: an orchestrator agent analyses tasks and coordinates specialised agents and vision expert models to provide detailed visual analysis [44]. Finally, autonomous agents can also be found in mobile applications. For instance, UI-TARS [28] is a GUI framework that leverages a native 72B LLM model to execute various tasks and compete with frameworks using OpenAI's GPT-4 model.

6.2.2 AI-powered Test Agents. The literature does not yet contain much information about Autonomous Testing Agents for web applications. This term was first used in the vision paper of Feldt et al. [11] where they envision ATAs in a very similar manner as the one defined in our article.

GPTDroid [25] leverages the generation capabilities of LLMs to automate mobile applications GUI testing. It extracts and encodes information into prompts, and then uses the LLM's answers to execute operation within the app, effectively generating activity. These operations are then recorded

and saved into a crash report when a bug appears, easing the process of reproducing context. This approach differs from ours as it creates its own actions and expected results from the nominal behaviours of apps the LLM has seen during its training, lacking the precise assertions an expert tester writes in natural language test cases. This type of approach could thus be complementary to ours to discover bugs outside predefined end-to-end test cases.

Finally, commercial solutions can be found on the internet to automate both test case generation and their executions, such as Flowtest.ai or Octomind.dev. None of these solutions disclose their architecture, nor their effectiveness across applications and test case types. This lack of transparency motivated us to develop a benchmark that enables end-users to easily compare ATAs architectures, fostering a clearer understanding of their strengths and limitations. In addition, by providing open-source ATAs we hope to contribute to a more collaborative ATA community.

7 Conclusion

The rapid advancement of Autonomous Web Agents powered by modern LLMs has opened new possibilities for automating web application interactions. Given the resemblance between manual end-to-end test case instructions and the input formats used by AWAs, a compelling question arises: can AWAs be transformed into Autonomous Test Agents capable of executing end-to-end test cases autonomously? This idea has gained traction in industry, with several closed-source ATA solutions emerging. However, little is known about their design or real-world effectiveness.

In this article, we explored the distinction between AWAs and ATAs, introduced two ATA implementations—SeeAct-ATA and PinATA—and developed a realistic benchmark for evaluating ATA performance. Our experimental study demonstrated that PinATA can successfully execute 61% of benchmark test cases with effectiveness comparable to manual testers, even without prior knowledge of the applications. Additionally, when paired with the Sonnet LLM, PinATA achieved an impressive 94% sensitivity, detecting nearly all failing test cases. However, both SeeAct-ATA and PinATA exhibited a tendency to misclassify passing test cases as failing, with specificity not exceeding 57%.

While these results are promising, several challenges remain. Some limitations are shared with AWAs, meaning ATAs could benefit from advancements in the broader AWA research community. However, ATAs also face unique obstacles, particularly in assertion verifiability and test case conformity. Addressing these challenges will be critical to improving ATA reliability and effectiveness, ultimately paving the way for ATAs to become robust autonomous testers for web application testing.

8 Data Availability

The benchmark, the results of the different executions, the manual verification and the code repository are all openly available on a Zenodo repository[7]. All the instructions to use the models can be found in a readme at the root of the replication package.

The dockers for the offline web applications used for this experiment can be found on the WebArena GitHub repository [6] along with the populated databases.

Acknowledgments

---

[6]https://github.com/OSU-NLP-Group/SeeAct

References

[1] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, Hong Kong, 403–410. https://doi.org/10.1109/DSN.2011.5958253 ISSN: 2158-3927.

[2] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. 2000. Automating Web navigation with the WebVCR. *Computer Networks* 33, 1 (June 2000), 503–517. https://doi.org/10.1016/S1389-1286(00)00073-6

[3] Sacha Brisset, Romain Rouvoy, Lionel Seinturier, and Renaud Pawlak. 2022. Erratum: Leveraging Flexible Tree Matching to repair broken locators in web automation scripts. *Information and Software Technology* 144 (April 2022), 106754. https://doi.org/10.1016/j.infsof.2021.106754

[4] Khyathi Raghavi Chandu, Yonatan Bisk, and Alan W. Black. 2021. Grounding 'Grounding' in NLP. http://arxiv.org/abs/2106.02192 arXiv:2106.02192 [cs].

[5] Qi Chen, Dileepa Pitawela, Chongyang Zhao, Gengze Zhou, Hsiang-Ting Chen, and Qi Wu. 2023. WebVLN: Vision-and-Language Navigation on Websites. https://doi.org/10.48550/arXiv.2312.15820 arXiv:2312.15820 [cs].

[6] Wei Chen, Hanyang Cao, and Xavier Blanc. 2021. An Improving Approach for DOM-Based Web Test Suite Repair. In *Web Engineering*, Marco Brambilla, Richard Chbeir, Flavius Frasincar, and Ioana Manolescu (Eds.). Springer International Publishing, Biarritz, France, 372–387. https://doi.org/10.1007/978-3-030-74296-6_29

[7] Antoine Chevrot, Alexandre Vernotte, Jean-Rémy Falleri-Vialard, Xavier Blanc, and Bruno Legeard. 2025. *Autonomous Tester Agent Benchmark*. Smartesting. https://doi.org/10.5281/zenodo.15198569

[8] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a Generalist Agent for the Web. https://doi.org/10.48550/arXiv.2306.06070 arXiv:2306.06070 [cs].

[9] Sergio Di Meglio and Luigi Libero Lucio Starace. 2024. Towards Predicting Fragility in End-to-End Web Tests. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*. Association for Computing Machinery, Salerno, Italy, 387–392. https://doi.org/10.1145/3661167.3661179 event-place: Salerno, Italy.

[10] Ignacio Dobles, Alexandra Martínez, and Christian Quesada-López. 2019. Comparing the effort and effectiveness of automated and manual tests. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, Coimbra, Portugal, 1–6. https://doi.org/10.23919/CISTI.2019.8760848 ISSN: 2166-0727.

[11] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards Autonomous Testing Agents via Conversational Large Language Models. https://doi.org/10.48550/arXiv.2306.05152 arXiv:2306.05152 [cs].

[12] Boni García, Jose M. del Alamo, Maurizio Leotta, and Filippo Ricca. 2024. Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. In *Quality of Information and Communications Technology*, Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini (Eds.). Springer Nature Switzerland, Pisa, Italy, 142–149. https://doi.org/10.1007/978-3-031-70245-7_10

[13] Luca Gioacchini, Giuseppe Siracusano, Davide Sanvito, Kiril Gashteovski, David Friede, Roberto Bifulco, and Carolin Lawrence. 2024. AgentQuest: A Modular Benchmark Framework to Measure Progress and Improve LLM Agents. https://doi.org/10.48550/arXiv.2404.06411 arXiv:2404.06411 [cs].

[14] Yu Gu, Xiang Deng, and Yu Su. 2023. Don't Generate, Discriminate: A Proposal for Grounding Language Models to Real-World Environments. http://arxiv.org/abs/2212.09736 arXiv:2212.09736 [cs].

[15] Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. 2023. A Real-World WebAgent with Planning, Long Context Understanding, and Program Synthesis. http://arxiv.org/abs/2307.12856 arXiv:2307.12856 [cs].

[16] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: an incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, Seattle, WA, USA, 751–762. https://doi.org/10.1145/2950290.2950294

[17] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. 2016. Why do Record/Replay Tests of Web Applications Break?. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Chicago, IL, USA, 180–190. https://doi.org/10.1109/ICST.2016.16

[18] Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models. https://doi.org/10.48550/arXiv.2401.13919 arXiv:2401.13919 [cs].

[19] Iat Long Iong, Xiao Liu, Yuxuan Chen, Hanyu Lai, Shuntian Yao, Pengbo Shen, Hao Yu, Yuxiao Dong, and Jie Tang. 2024. OpenWebAgent: An Open Toolkit to Enable Web Agents on Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Yixin Cao, Yang Feng, and Deyi Xiong (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 72–81. https://aclanthology.org/2024.acl-demos.8

[20] Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. 2024. AutoWebGLM: A Large Language Model-based Web Navigating Agent. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24)*. Association for Computing Machinery, New York, NY, USA, 5295–5306. https://doi.org/10.1145/3637528.3671620 event-place: Barcelona, Spain.

[21] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-Based Web Locators: An Empirical Study. In *Web Engineering*, Sven Casteleyn, Gustavo Rossi, and Marco Winckler (Eds.). Springer International Publishing, Toulouse, France, 322–340. https://doi.org/10.1007/978-3-319-08245-5_19

[22] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. 2023. Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Dublin, Ireland, 339–350. https://doi.org/10.1109/ICST57152.2023.00039 ISSN: 2159-4848.

[23] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing Web Test Cases Aging by Means of Robust XPath Locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, Naples, Italy, 449–454. https://doi.org/10.1109/ISSREW.2014.17

[24] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Graz, Austria, 1–10. https://doi.org/10.1109/ICST.2015.7102611 ISSN: 2159-4848.

[25] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, Lisbon, Portugal. https://doi.org/10.1145/3597503.3639180 event-place: Lisbon, Portugal.

[26] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, Virtual Event, USA, 1498–1508. https://doi.org/10.1145/3368089.3417069

[27] Vu Nguyen, Thanh To, and Gia-Han Diep. 2021. Generating and selecting resilient and maintainable locators for Web automated testing. *Software Testing, Verification and Reliability* 31, 3 (2021), e1760. https://doi.org/10.1002/stvr.1760 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1760.

[28] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. 2025. UI-TARS: Pioneering Automated GUI Interaction with Native Agents. https://doi.org/10.48550/arXiv.2501.12326 arXiv:2501.12326.

[29] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2019. Chapter Three - Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE. In *Advances in Computers*, Atif M. Memon (Ed.). Vol. 113. Elsevier, Philadelphia, PA, USA, 89–133. https://doi.org/10.1016/bs.adcom.2018.10.005 ISSN: 0065-2458.

[30] S Schwartz and K M Carpenter. 1999. The right answer for the wrong question: consequences of type III error for public health research. *American Journal of Public Health* 89, 8 (Aug. 1999), 1175–1180. https://doi.org/10.2105/AJPH.89.8.1175 Publisher: American Public Health Association.

[31] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, Lake Buena Vista, FL, USA, 503–514. https://doi.org/10.1145/3236024.3236063

[32] Guangzhi Sun, Xiao Zhan, and Jose Such. 2024. Building Better AI Agents: A Provocation on the Utilisation of Persona in LLM-based Conversational Agents. In *ACM Conversational User Interfaces 2024*. ACM, Luxembourg Luxembourg, 1–6. https://doi.org/10.1145/3640794.3665887

[33] Jóakim V. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, Austin Texas, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[34] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (April 2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208 Conference Name: IEEE Transactions on Software Engineering.

[35] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2024. A Survey on Large Language Model based Autonomous Agents. *Frontiers of Computer Science* 18, 6 (Dec. 2024), 186345. https://doi.org/10.1007/s11704-024-40231-1 arXiv:2308.11432 [cs].

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903 arXiv:2201.11903 [cs].

[37] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering*. The Kluwer International Series in Software Engineering, Vol. 6. Springer US, Boston, MA. https://doi.org/10.1007/978-1-4615-4625-2

[38] Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. 2024. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. https://doi.org/10.48550/arXiv.2412.14161 arXiv:2412.14161 [cs].

[39] Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. Set-of-Mark Prompting Unleashes Extraordinary Visual Grounding in GPT-4V. https://doi.org/10.48550/arXiv.2310.11441 arXiv:2310.11441 [cs].

[40] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, Sacramento, California, USA, 1607–1619. https://doi.org/10.1145/3691620.3695529

[41] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, Chiang Mai, Thailand, 206–217. https://doi.org/10.1109/QRS60937.2023.00029 ISSN: 2693-9177.

[42] Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. 2024. WebPilot: A Versatile and Autonomous Multi-Agent System for Web Task Execution with Strategic Exploration. https://doi.org/10.48550/arXiv.2408.15978 arXiv:2408.15978 [cs].

[43] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A Survey on the Memory Mechanism of Large Language Model based Agents. http://arxiv.org/abs/2404.13501 arXiv:2404.13501 [cs].

[44] Zhehao Zhang, Ryan Rossi, Tong Yu, Franck Dernoncourt, Ruiyi Zhang, Jiuxiang Gu, Sungchul Kim, Xiang Chen, Zichao Wang, and Nedim Lipka. 2024. VipAct: Visual-Perception Enhancement via Specialized VLM Agent Collaboration and Tool-use. http://arxiv.org/abs/2410.16400 arXiv:2410.16400.

[45] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. GPT-4V(ision) is a Generalist Web Agent, if Grounded. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, Vienna, Austria, 61349–61385. https://proceedings.mlr.press/v235/zheng24e.html

[46] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. http://arxiv.org/abs/2307.13854 arXiv:2307.13854 [cs].

[47] Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. 2024. Agent-as-a-Judge: Evaluate Agents with Agents. https://doi.org/10.48550/arXiv.2410.10934 arXiv:2410.10934 [cs] version: 1.

[48] T. Zimmermann. 2016. Card-sorting: From text to themes. In *Perspectives on Data Science for Software Engineering*, Tim Menzies, Laurie Williams, and Thomas Zimmermann (Eds.). Morgan Kaufmann, Boston, 137–141. https://doi.org/10.1016/B978-0-12-804206-9.00027-1