

WebTestPilot: Agentic End-to-End Web Testing against Natural Language Specification by Inferring Oracles with Symbolized GUI Elements

XIWEN TEOH, National University of Singapore, Singapore

YUN LIN*, Shanghai Jiao Tong University, China

DUC-MINH NGUYEN, Shanghai Jiao Tong University, China

RUOFEI REN, Shanghai Jiao Tong University, China

WENJIE ZHANG, National University of Singapore, Singapore

JIN SONG DONG, National University of Singapore, Singapore

Visual language model (VLM) agents show great promise in automating graphical user interface (GUI) testing against requirements in natural language. However, the probabilistic nature of language models can have inherent hallucinations. Therefore, given a detected inconsistency between the requirement and the web application, it is hard to distinguish whether it stems from the hallucination or a real application bug. Addressing this issue presents two core technical challenges: (1) limited capability and accuracy in deriving implicit test oracles, where the agent must act as its own oracle to implicitly decide if the application's behavior is correct without guidance, and (2) limited reliability due to probabilistic inference, where an LLM's inconsistent reasoning undermines its trustworthiness as an oracle.

We introduce WEBTESTPILOT, a neurosymbolic LLM-based approach that addresses both challenges through symbolization. WEBTESTPILOT detects and abstracts critical GUI elements of a web application into symbolic variables. This design improves reliability by constraining assertion generation to operations grounded in explicitly defined symbols, thereby reducing unconstrained or inconsistent reasoning. At the same time, it improves accuracy by representing application states and their relationships in a structured symbolic form, which increases the likelihood of the agent recognizing data, causal, and temporal dependencies across states. Together, these capabilities enable WEBTESTPILOT to generate reliable and accurate test oracles that capture meaningful implicit expectations derived from test requirements. To advance research in this area, we build a benchmark of bug-injected web apps for evaluating NL-to-E2E testing. The results show that WebTestPilot achieves a task completion rate of 99%, with 96% precision and 96% recall in bug detection, outperforming the best baseline (+70 precision, +27 recall). The agent generalizes across diverse natural language inputs (i.e., those containing typos, grammatical errors, redundant sentences, stylistic restyling, or abbreviations) and model scales (3B–72B). In a real-world deployment with a no-code platform, WebTestPilot discovered 8 bugs during development, including data binding, UI, and navigation issues.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging; Domain specific languages; Consistency.**

*Corresponding author.

Authors' addresses: [Xiwen Teoh](mailto:xiwen.teoh@nus.edu.sg), National University of Singapore, Singapore, xiwen.teoh@nus.edu.sg; [Yun Lin](mailto:yun.lin@sjtu.edu.cn), Shanghai Jiao Tong University, China, lin_yun@sjtu.edu.cn; [Duc-Minh Nguyen](mailto:duc-minh.nguyen@sjtu.edu.cn), Shanghai Jiao Tong University, China, minh.nguyen@sjtu.edu.cn; [Ruofei Ren](mailto:renruofei0120@sjtu.edu.cn), Shanghai Jiao Tong University, China, renruofei0120@sjtu.edu.cn; [Wenjie Zhang](mailto:wjzhang@nus.edu.sg), National University of Singapore, Singapore, wjzhang@nus.edu.sg; [Jin Song Dong](mailto:dcsdjs@nus.edu.sg), National University of Singapore, Singapore, dcsdjs@nus.edu.sg.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/7-ARTFSE087

<https://doi.org/10.1145/3797115>

ACM Reference Format:

Xiwen Teoh, Yun Lin, Duc-Minh Nguyen, Ruofei Ren, Wenjie Zhang, and Jin Song Dong. 2026. WebTestPilot: Agentic End-to-End Web Testing against Natural Language Specification by Inferring Oracles with Symbolized GUI Elements. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE087 (July 2026), 25 pages. <https://doi.org/10.1145/3797115>

1 INTRODUCTION

The global progressive web application market is projected to reach USD 9.4 billion by 2030 [51]. As web applications grow in scale and complexity, companies turn to end-to-end (E2E) testing to safeguard reliability for end users, in which testers translate requirements into executable scripts (e.g., Selenium, Playwright, Cypress) that simulate user interactions and verify that applications behave as intended through their end-user interfaces. Without such safeguards, unchecked bugs can escalate into failures that have caused high-profile breakdowns [64, 65].

E2E testing has two main branches. *Exploration-based testing* explores all possible states of a web application to maximize coverage. *Specification-based testing* [48] verifies that the web application behaves consistently with business requirements. Most prior work targets the former, using techniques such as random exploration [18, 45], model-based testing [43, 44], search-based testing [6, 7, 14, 15, 41, 74], symbolic execution [3], and reinforcement learning [9, 19, 42, 75, 76, 81]. While effective for finding vulnerabilities and corner cases, these methods ignore documentation produced during development, and thus fail to capture meaningful user behaviors.

In this work, we look into the latter branch. We transform natural language requirements drawn from any source (i.e., UX/UI specifications, product requirements, technical designs, quality assurance plans, or API documents) into executable test actions. Existing approaches both in industry (Cucumber [12], RSpec [53], Squish [21]) and academia (GUIPilot [32], Appflow [22]) also look into requirements but rely on rigid input formats (e.g., Sketch files, Gherkin) compatible with their parser. In contrast, we propose a flexible framework that generates tests with verifiable oracles from any natural language excerpt, which (1) directly validates applications against business requirements, (2) speeds up testing for continuous integration and deployment, and (3) reduces maintenance by regenerating tests when requirements change.

Recent advances in large language model (LLM) agents with multimodal reasoning open new possibilities for specification-based GUI testing. According to the State of Software Quality Report [58] in 2024, over 58% of respondents use LLM-based tools in automated testing, yet adoption remains limited by capability gaps (44%) and reliability concerns (30%). When an agent flags an inconsistency, it is unclear whether the issue stems from the agent itself (hallucination) or the web application (a real bug). Effective automated testing must distinguish between these sources, which give rise to our two key technical challenges:

Limited capability and accuracy in deriving test oracles. Automated E2E testing requires the agent to act as its own oracle, which is non-trivial [5]. An effective test oracle must infer underlying test requirements and translate implicit expectations into concrete assertions. These assertions are only meaningful if they are grounded in data (values reflect prior inputs and computations), causal (state transitions result from the intended actions), and temporal (changes in states referencing the same page over time) dependencies across one or more states. For example, verifying that a “*product has been added to cart*” requires not only checking the newly added item, but also ensuring consistency with existing items in the cart (i.e., product types, quantities, and subtotals). Existing works such as NAVIQATE [55] and LAVAGUE [26] lack oracle capability. They translate test requirements directly into actions without generating assertions. Although PINATA [10] generates assertions, it relies on a global memory that is (1) precomputed, (2) unstructured, and (3) capacity-limited.

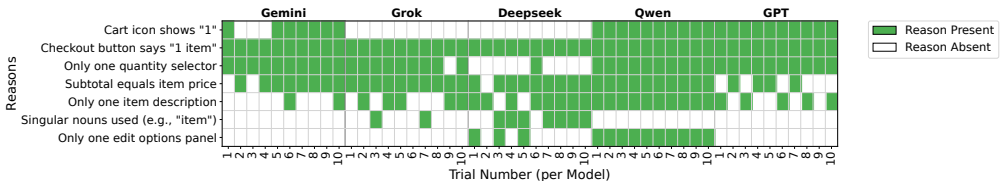


Fig. 1. Inconsistent reasoning by different LLMs with multiple trials in test state verification.

Consider a shopping scenario. On the cart page, PINATA preemptively stores cart items in free-form natural language (e.g., “Cart contains: Laptop – \$1200, Mouse – \$25”) and carries this textual summary forward throughout test execution. This design leads to three limitations:

- (1) *Loss of recoverability due to precomputed memory.* Only information explicitly recorded at observation time is preserved. If the agent later reaches checkout and intermediate values (e.g., subtotal or applied discounts) were not stored, it cannot reconstruct how the final total was derived. Without these transformation links, it cannot verify whether the total is correct.
- (2) *Loss of dependencies due to unstructured memory.* Historical information accumulates as loosely organized natural language snippets without symbolic identifiers across states. To verify that “the checkout total equals the sum of item prices minus discount,” the agent must retrieve fragments from noisy text. Lacking structured references (cart → shipping → checkout), it may confuse entries or miss updates, making dependency reasoning brittle.
- (3) *Loss of changes over time due to capacity-limited memory.* As execution lengthens, earlier states are summarized or truncated. For example, if a user applies a 10% discount and then removes an item, the specification requires the discount to be recalculated. If only the final total is retained, the agent cannot verify whether the recalculation occurred after the removal.

Limited reliability due to probabilistic inference. By design, LLMs are stochastic: the same prompt can yield different responses even with fixed model settings. When tasked with verifying states during testing, this randomness can lead to inconsistent reasoning. To illustrate, we conducted an empirical study (Figure 1): five mainstream LLMs (GPT, Gemini, Grok, Deepseek, and Qwen) were each prompted 10 times with the same page screenshot and the question, “Does the shopping cart contain only one item?” Across trials, the models produced seven distinct answers, with reasoning varying both across and within models. This variability has visible consequences for multi-step tests. Consider a test with n sequential steps, where each step relies on correct reasoning from the LLM. Even if the probability of a single step being correct (p) is high, the stochastic nature of the model means that completing the full trajectory successfully becomes unlikely (p^n), as errors compound across steps. Inconsistent outputs in individual steps produce flaky end-to-end verdicts, and interpreting the models’ natural language reasoning adds further manual overhead, breaking the assumption of a stable test oracle in automated testing.

On one hand, effective test oracles must establish data, causal, and temporal dependencies across states to infer implicit expectations that satisfy test requirements. On the other hand, free-form reasoning and verification with stochastic LLMs is inherently unstable. To address these challenges, we propose WEBTESTPILOT, a neurosymbolic approach capable of acting as its own accurate and reliable test oracle. WEBTESTPILOT uses symbolization to uniformly improve both oracle accuracy and reliability. Building on the success of effective perception models [39] to extract symbols from states, our approach is guided by two key insights: (1) Symbolization improves reliability by converting test oracle generation from a continuous space into a discrete one with finite bounds. By defining explicit symbols, the set of possible assertions is constrained, which reduces uncertainty and guides the agent towards assertions that are semantically valid and consistent. In addition,

WEBTESTPILOT can perform retrials when generating assertions to reduce hallucinations and maintain stability. and (2) Grounding assertions in symbols improves accuracy. By representing states and their relationships as structured symbols, the agent can explicitly track how UI elements evolve across states. the agent is more likely to recognize data, causal, and temporal dependencies across states. This structured exposure increases the chance that generated assertions capture implicit expectations and faithfully reflect the intended behavior of the application. However, designing this approach involves two technical challenges:

How to link symbols with assertions? After extracting symbols, the agent must compose them into correct, executable assertions. The challenge is designing a domain-specific language (DSL) that balances expressiveness and simplicity: it must be rich enough to capture application behaviors, yet simple enough to avoid hallucination or retraining. We address this by extending an existing programming language, using its familiar syntax and native libraries for data processing, while providing a predefined set of operators over symbols (e.g., relational and compositional predicates).

How to achieve effective and efficient symbolization? A naive approach would symbolize all visible UI elements and track dependencies for every symbol, leading to combinatorial explosion and the same limitations as global memory in PINATA. We instead propose *page reidentification*. It assigns consistent identifiers to logically equivalent pages (e.g., two or more states pointing to the Cart page) and maintains a structured *Session* history of states. Rather than symbolizing eagerly, symbols are derived *on demand* by retrieving states with the same page identifier and extracting only the relevant elements. It enables focused (fewer symbols) and scalable (more states) reasoning.

Specifically, given a natural language test requirement, WEBTESTPILOT decomposes it into n (condition, action, expectation) steps. For each step, WEBTESTPILOT translates the condition and expectation into pre- and post-condition assertions. It then applies symbolization to extract relevant UI components as symbols, which are composed via a DSL to construct executable assertions satisfying the specified constraints. To support cross-state reasoning, WEBTESTPILOT uses page reidentification to detect revisited pages and maintain a structured history of test states.

We evaluate WEBTESTPILOT on a newly constructed benchmark of four bug-injected web applications, comparing its performance against three LLM-based GUI testing baselines (NAVIQATE [55], LAVAGUE [26], PINATA [10]). Our results show that WEBTESTPILOT achieves a test completion rate of 99%, with 96% precision and 96% recall in bug detection, outperforming the strongest baseline by +70 precision and +27 recall. WEBTESTPILOT is robust across diverse natural language inputs (i.e., those containing typos, grammatical errors, redundant sentences, stylistic restyling, or abbreviations) as well as across model scales from 3B to 72B parameters. In a real-world deployment with our collaboration partner, a no-code platform, WEBTESTPILOT discovers 8 bugs during development.

In summary, our contributions are as follows:

- **Methodology:** We propose the first neurosymbolic GUI testing approach. The neural component extracts symbols from application states to capture dependencies. The symbolic component constructs assertions over the properties, values, and relations of these symbols, ensuring that implicit expectations in the test requirements are satisfied.
- **Implementation:** We present WEBTESTPILOT, a framework realizing our approach, which has been successfully adopted by our industry collaborator, China Mobile.
- **Experiments:** We build a benchmark of four open-source, real-world web applications with 100 injected bugs. We evaluate WEBTESTPILOT against LLM baselines on this benchmark and in real-world settings (industry collaborations and GitHub issues), showing that it outperforms state-of-the-art methods in bug detection.

The source code for WEBTESTPILOT and the benchmark are available at <https://github.com/code-philialia/WebTestPilot>.

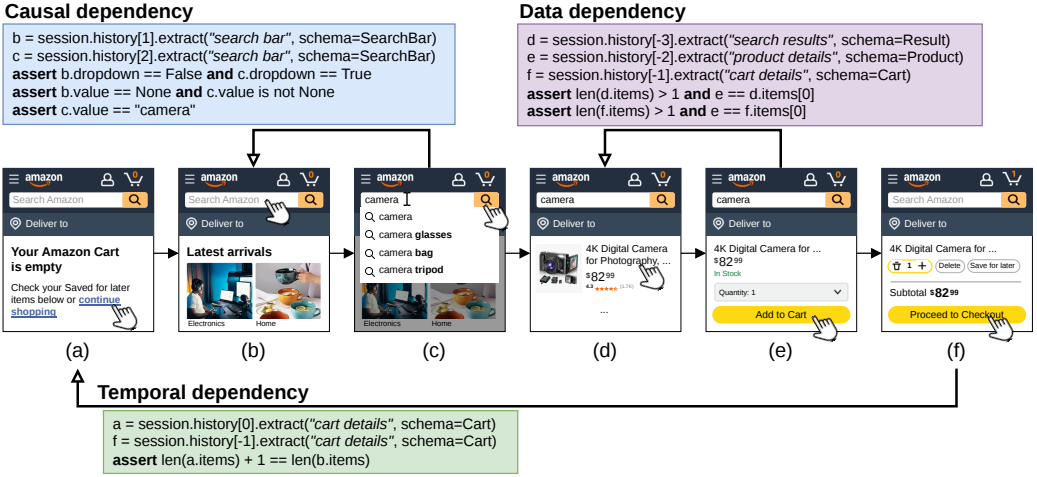


Fig. 2. A test flow depicting search, shopping, and checkout on e-commerce platform amazon.com.

2 MOTIVATING EXAMPLE

Figure 2 shows a test flow on amazon.com, a representative e-commerce scenario. The flow begins on the cart page (State 2(a)) with an empty cart. The user clicks “Continue Shopping” to navigate to the homepage (State 2(b)), enters the query “camera” in the search bar, triggering a suggestion dropdown (State 2(c)), and submits the query to reach a results page (State 2(d)). The user then clicks the first product to view its details (State 2(e)) and adds it to the cart, completing the test flow.

Applying LLM agents to verify such flows is challenging because meaningful test oracles must reason over dependencies across states, not just the correctness of individual steps. Prior work, such as NAVIQATE [55] and LAVAGUE [26], considers reaching the end as success without verifying intermediate states. Using the scenario above, if they successfully navigate from State (a) to (f), then the test case passes. Although PINATA [10] maintains a memory to store information and compare it against expected outcomes, its general and unstructured design limits its ability to retrieve task-relevant context for constructing assertions. For example, to verify that the cart subtotal increases exactly by the price of the newly added product, it may fail to recognize that states (a) and (f) correspond to the same logical page, preventing detection of inconsistent incremental changes (i.e., subtotal difference (f) - (a) equals the price of the newly added product). Similarly, to verify that every selected search attribute (e.g., the price range) is preserved across the search results (d), which may include hundreds of products, an unstructured memory may omit a single attribute, leading to incomplete verification. Many real bugs arise from such inconsistencies. To catch them, it is necessary to reason about the implicit *causal*, *data*, and *temporal* dependencies between states, which are explained below:

- **Causal Dependency:** A relation between adjacent states that holds when UI elements in the current state are created, updated, or deleted as a direct effect of executing an action in the previous state. For example, the auto-complete suggestion dropdown and the populated search input in state (c) depends on the typing action in state (b).
- **Data Dependency:** A relation between states that holds when information extracted in one state is propagated to and reused in another, forming a data flow across the execution trace. For example, the product details in (e) depend on the selected item from the search results in (d), and the cart items in (f) depend on the product details in (e).

- **Temporal Dependency:** A relation between states corresponding to the same logical page that holds when a later state must be interpreted relative to an earlier state to detect incremental changes over time. For example, state (f) depends on state (a), both representing the cart page, to determine how the cart contents have evolved after user actions.

To enable robust cross-state verification of implicit expectations, WEBTESTPILOT supports declarative schemas, which act as symbol templates (or “variables”) representing structured UI data. The schemas are implemented as strongly typed models that can automate parsing, normalization, and validation of extracted content. They define not only the expected data structure, but also type-level constraints (e.g., supported strings), field-level requirements (e.g., required vs. optional), and domain-specific rules (e.g., non-negative prices).

Concretely, consider a test step where after executing the action “click Add to Cart”, its corresponding expectation is “the product is now in the cart.” To act as a test oracle for this post-condition, WEBTESTPILOT first applies symbolization to define relevant symbols (Figure 3).

<pre>class Product(BaseModel): title: str = Field(...) price: float = Field(..., ge=0) quantity: Optional[int] = Field(None, gt=0)</pre>	<pre>class Cart(BaseModel): items: List[Product] = Field(...)</pre>
--	---

Fig. 3. Definition of the Product and Cart symbols, represented as Pydantic schemas.

WEBTESTPILOT then instantiates these schemas with values extracted from the current and prior states. By referencing page reidentification, it recognizes that State (a) and State (f) correspond to the Cart page and learns a high-level overview of its layout (e.g., the cart contains a list of items, each displaying specific information), allowing the Cart symbol to be applied. Similarly, it identifies State (e) as the Product Detail page, where the Product symbol is relevant. By combining this information with the historical actions, the agent establishes the semantic connection of adding a product to the cart through the transitions State (a) → State (e) → State (f). See Figure 4.

<pre># State (a): Extract previous cart summary from the initial state prior = session.history[0].extract("Get cart summary", schema=Cart).items # State (e): Extract added product from the product details page added = session.history[-2].extract("Get product detail", schema=Product) # State (f): Extract current cart summary from the latest state current = session.history[-1].extract("Get cart summary", schema=Cart).items</pre>
--

Fig. 4. Extracting the added product from product page and comparing current and prior cart details.

Using the DSL, the agent constructs a formal assertion on these symbols to verify that the post-action state satisfies both explicit expectations (the product is in the cart) and implicit expectations derived from prior states (the cart contains the same items as before plus the new product, the product type matches the previously viewed item in (e), and the quantity is 1). See Figure 5. This allows WEBTESTPILOT to detect bugs from implicit and cross-state causal, data, or temporal violations (e.g., missing/duplicate items, wrong quantities or prices, or UI inconsistencies).

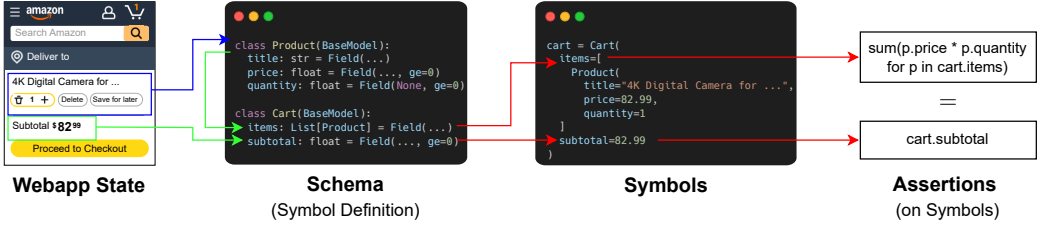


Fig. 6. Example (from motivating scenario): WEBTESTPILOT extracts *symbols* via declared *schemas* that correspond to GUI elements for making *assertions* on the application *state*.

```
# All product attributes (title, quantity, price) match prior and added items
for prod in prior + [added]:
    match = next((p for p in current if p.title == prod.title), None)
    assert match is not None, f"Product {prod.title} missing in current cart"
    assert match.quantity == prod.quantity, f"Quantity mismatch for {prod.title}"
    assert match.price == prod.price, f"Price mismatch for {prod.title}"

# The cart subtotal correctly reflects the addition of the new product.
prior_subtotal = sum(p.price * p.quantity for p in prior)
added_total = added.price * added.quantity
current_subtotal = sum(p.price * p.quantity for p in current)
assert current_subtotal == prior_subtotal + added_total, "Cart subtotal mismatch"
```

Fig. 5. Assertion generated by WEBTESTPILOT.

3 PROBLEM STATEMENT

Preliminary. We model a web application \mathcal{W} as a graph of states $s \in \mathcal{S}$. Each state is defined as a tuple $s = (\text{screenshot}, \text{DOM})$, where screenshot encodes the visual appearance of the page, and DOM is a rooted, ordered tree of UI elements e , where each element encodes its type (i.e., button, input), relevant attributes (e.g., name, value, enabled/disabled), and child elements. A user interacts with \mathcal{W} by executing an action $a = \langle t, e, p \rangle$, where t is the action type (e.g., click, type), e is the target UI element, and p is an optional parameter (e.g., text to enter). The state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ maps a state and action to a successor state $s' = \mathcal{T}(s, a)$. Finally, executing a sequence of actions $A = \langle a_1, a_2, \dots, a_n \rangle$ from an initial state s_0 produces an execution trace $\tau = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ or $s_0 \xrightarrow{A} s_n$.

Objective. Given a natural language test requirement D , an automated tester T parses D into a sequence of steps $\langle \text{step}_1, \dots, \text{step}_n \rangle$, $D = \text{step}_1 \oplus \dots \oplus \text{step}_n$, and maps each step to an output $o_i = T(\text{step}_i)$. The details of the input and output are as follows:

Input. Let the natural language test requirement be a finite sequence of textual tokens $D = (w_1, \dots, w_m)$. We assume D can be mapped to an ordered sequence of executable steps $\langle \text{step}_1, \dots, \text{step}_n \rangle$, where each step step_i is derived from a contiguous token sequence in D . The sequence satisfies:

- (1) *Disjointness.* The contiguous token sequences corresponding to the steps are pairwise disjoint, i.e., no token in D contributes to more than one step.
- (2) *Monotonicity.* The steps preserve the left-to-right order of D : if step_i is derived from tokens that precede those of step_j , then $i < j$.
- (3) *Semantic completeness.* Each step step_i encodes: (i) a set of predicates over the input state, (ii) an action to be executed, and (iii) a set of predicates over the output state.

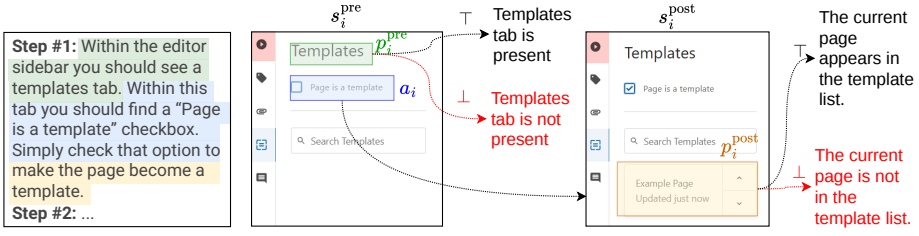


Fig. 7. Visualization of the problem statement’s input and output. D is parsed into steps. The colored overlays highlight regions of interest on the state screenshots: green denotes the portion evaluated by p_i^{pre} , blue corresponds to the action a_i , and yellow denotes the portion evaluated by p_i^{post} . For each predicate, we illustrate the conditions under which the state is consistent or inconsistent with the requirement D .

Output. We define a predicate p_i as a property over application states, $p_i : \mathcal{S} \rightarrow \{\top, \perp\}$. We write $s_i \models p_i$ if the state s_i satisfies p_i , i.e., $p_i(s_i) = \top$. A bug occurs when $s_i \not\models p_i$ or $p_i(s_i) = \perp$, meaning the state is reported as inconsistent by T with the requirements specified in step i . During execution, for each parsed step step_i from D , T produces three artifacts $o_i = T(\text{step}_i) = (p_i^{\text{pre}}, a_i, p_i^{\text{post}})$, where:

- (1) A predicate p_i^{pre} evaluated on the state s_i^{pre} before the action.
- (2) An action a_i applied on s_i^{pre} , which transitions \mathcal{W} to a new state s_i^{post} .
- (3) A predicate p_i^{post} evaluated on the state s_i^{post} after the action.

For T without assertion capability, p_i^{pre} and p_i^{post} will always evaluate to \top .

4 APPROACH

Overview Figure 8 shows WEBTESTPILOT’s overall approach. Its key novelty is serving as a capable and reliable test oracle, generating predicates p_i that verify implicit expectations from test requirements. This section is organized as follows:

- **Input Parsing (Section 4.1).** WEBTESTPILOT parses a natural language requirement into a structured sequence of steps $\langle \text{step}_1, \dots, \text{step}_n \rangle$, where each step specifies the state before the action ($\text{condition}_{\text{NL}}$), the action itself ($\text{action}_{\text{NL}}$), and the state after the action ($\text{expectation}_{\text{NL}}$).
- **Oracle Inference (Section 4.2).** For each step, WEBTESTPILOT analyzes the explicit requirements $\text{condition}_{\text{NL}}$ and $\text{expectation}_{\text{NL}}$. It inspects the execution trace τ to identify temporal, data, and causal dependencies, which it uses to infer implicit requirements. WEBTESTPILOT then defines symbols that abstract relevant states and establishes schemas for their expected content. Finally, it uses a DSL to formalize predicate assertions over the symbols from implicit expectations inferred from requirements, producing $\text{precondition}_{\text{DSL}}$ and $\text{postcondition}_{\text{DSL}}$.
- **Oracle Execution (Section 4.3).** With the assertions ready, WEBTESTPILOT maps $\text{action}_{\text{NL}}$ to an executable action on the web application. Before the action, it evaluates $\text{precondition}_{\text{DSL}}$ to ensure that the current state satisfies the step’s conditions. After the action, it evaluates $\text{postcondition}_{\text{DSL}}$ to verify the resulting state meets the expected outcome. If any assertion fails, WEBTESTPILOT retries the action up to n times. If all retries fail, it reports a bug.

4.1 Input Parsing

Input requirements come in many forms, from formal (PRD, user stories) to informal sources (meeting notes, messages, emails). WEBTESTPILOT normalizes these into a sequence of steps $\langle \text{step}_1, \dots, \text{step}_n \rangle$, each a 3-tuple $(\text{condition}_{\text{NL}}, \text{action}_{\text{NL}}, \text{expectation}_{\text{NL}})$ specifying when and where an action occurs, how it is performed, and the expected outcome in natural language. To extract

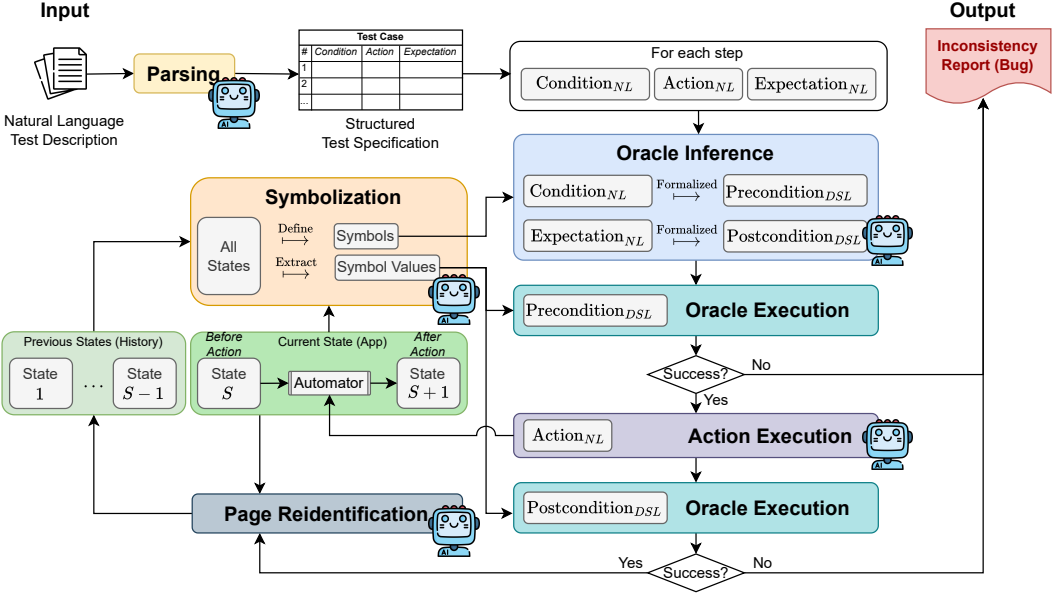


Fig. 8. Overview of WEBTESTPILOT. WEBTESTPILOT parses a natural language requirement into structured steps (**Input Parsing**), each specifying a condition, action, and expectation. For each step, it performs **Oracle Inference** to generate predicate assertions over symbols capturing explicit and implicit requirements. During **Oracle Execution**, it checks preconditions, executes the action, checks postconditions. Failed assertions trigger retries, and persistent failures are logged as bugs. (🤖) means the process prompts an LLM.

this structure, WEBTESTPILOT prompts an LLM with the raw input and parses its JSON output. It can be configured to infer and fill-in any missing steps.

4.2 Oracle Inference

For each step, $step_i = (condition_{NL}, action_{NL}, expectation_{NL})$, WEBTESTPILOT prompts an LLM in two stages. First, the LLM receives the explicit requirements ($condition_{NL}, expectation_{NL}$) and the execution trace in text form $string(\tau) = [string(s_0), \dots, string(s_n)]$ (see Section 4.3.2 for details) to infer implicit requirements by identifying causal, temporal, and data dependencies. Second, the LLM uses $string(\tau)$ together with explicit and implicit requirements to define custom symbols for relevant concepts (e.g., Car t, Product; Section 4.2.1). It then applies the DSL (Section 4.2.2) to generate formal predicate assertions over these symbols. This yields the mapping $condition_{NL} \mapsto precondition_{DSL}$ and $expectation_{NL} \mapsto postcondition_{DSL}$, where preconditions and postconditions are predicate assertions over the starting and ending states s and s' in a step, respectively.

4.2.1 State Symbolization. To let WEBTESTPILOT reason effectively and identify dependencies, it can abstract domain-specific concepts from any state via custom symbols (e.g., Car t or Product). It defines symbols in Pydantic with type constraints, descriptions, and default values. These symbols can be referenced inside predicate assertions, while their actual values are instantiated at execution time. Predicate assertions are evaluated over the instantiated symbols (see Section 4.3).

4.2.2 Domain Specific Language. To construct and manipulate predicate assertions over symbols, we design a Python-extended domain-specific language (DSL). Its BNF syntax is shown in Figure 9.

$\Phi ::= \text{assert Pred}$	<i>Top-level assertion</i>
$\text{Pred} ::= \text{Pred and Pred}$	Boolean logic and comparisons
$\text{Pred} ::=$ <ul style="list-style-type: none"> Pred or Pred not Pred (Pred) Expr comp Expr 	
$\text{Expr} ::= \text{value}$	Values, variables, field/method access
$\text{Expr} ::=$ <ul style="list-style-type: none"> var var.attr var.method(args) 	
$\text{comp} ::= =, !=, >, >=, <, <=, \text{in}, \text{not in}$	Comparison operators
$\text{args} ::= \text{Expr} ('; \text{Expr})^*$	Argument list (comma-separated)
$\text{var} ::= \text{identifier}$	Valid variable name in Python
$\text{attr} ::= \text{id} \text{text} \text{children} \dots$	Of Session, State, or Element object
$\text{method} ::= \text{extract}() \text{find}() \dots$	Of Session, State, or Element object

Fig. 9. BNF syntax of DSL for writing test assertions

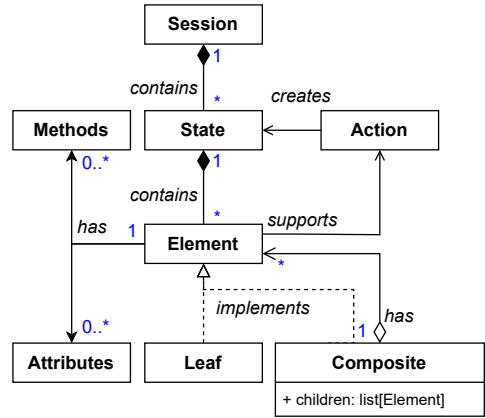


Fig. 10. Class diagram for built-in symbols.

Built-in Symbols. In addition to custom-defined symbols, the DSL provides a set of general-purpose symbols always available at every step. Figure 10 depicts their class structure. At the top level, Session stores the sequence of states and provides global access to past and current states. Each Session contains multiple State objects, each modeling a specific test step with page metadata and layout information represented as a tree of Elements. The State class offers methods to extract custom symbol values or directly access elements. Table 1 lists all the methods and attributes for these symbols.

Expressibility. By combining custom and built-in symbols, the DSL enables WEBTESTPILOT to reason about causal, data, and temporal dependencies. It supports five types of assertions:

- **Existence.** Verify the presence or absence of data, e.g., `state.find("profile") is not None`
- **Relational.** Verify spatial, structural, or logical relationships, e.g., `state.find("checkout button")[0].ymin > state.find("cart icon")[0].ymax`
- **Temporal.** Ensure events occur in a specific order, e.g., `all(a.extract(Banner).countdown >= b.extract(Banner).countdown for a, b in zip(states, states[1:]))`
- **Causal.** Check cause-effect relationships, e.g., `len(session.history[0].extract(Cart).items) - len(session.history[-1].extract(Cart).items) == 1`
- **Data Integrity.** Verify extracted or computed data matches expectations, e.g., `subtotal == sum(item.price for item in cart.items)`

Compositability. DSL predicates can be single first-order clauses or combinations of multiple clauses connected with logical operators (and, or, not) and grouped with parentheses. Predicates can also span multiple lines of assertions.

Manipulability. By extending Python, WEBTESTPILOT benefits from the LLM’s pre-existing Python knowledge. Its DSL supports Python Standard Library functionality for functional programming (e.g., `itertools`, `functools`, `operator`), text processing (`re`), built-in functions (`all()`, `any()`, `filter()`, `map()`, `len()`, `min()`, `max()`, etc.), and data types (`datetime`, `enum`). WEBTESTPILOT can also control execution with conditional statements and loops.

4.3 Oracle Execution

Once WEBTESTPILOT infers the $\text{Precondition}_{\text{DSL}}$ and $\text{Postcondition}_{\text{DSL}}$ predicate assertions, it executes the step in three stages. First, it executes $\text{Precondition}_{\text{DSL}}$. Then, it maps $\text{Action}_{\text{NL}}$ to an

Table 1. Built-in DSL symbols, their attributes and methods

Type	Methods / Attributes	Return Type	Description
Session	history state	list[State] State	Chronological list of all states. Current browser page.
State	page_id elements find(description: str, top_k: int) extract(instruction: str, schema: BaseModel)	string set[Element] list[Element] BaseModel	Logical page identifier shared across states. All state elements (flattened). Top K elements matching description (may be empty). Extracts a schema-conforming symbol from the state.
Element	xmin, ymin, xmax, ymax parent children extract(instruction: str, schema: BaseModel)	int Element list[Element] BaseModel	Bounding box coordinates. Parent element. Child elements. Extracts a schema-conforming symbol from the element.

executable action $a = \langle t, e, p \rangle$ (see Section 4.3.1) and executes it on the current state, producing $s \xrightarrow{a} s'$. Finally, it executes $\text{Postcondition}_{\text{DSL}}$. Formally, a step_{*i*} is successful iff:

$$(s \models \text{Precondition}_{\text{DSL}} \wedge s \xrightarrow{a} s') \implies s' \models \text{Postcondition}_{\text{DSL}}.$$

A test case is successful if every step_{*i*} in the sequence is successful. A bug occurs whenever any predicate assertion p at any step_{*i*} fails, i.e., $p(s) = \top$, $s \not\models p$.

4.3.1 Action Execution. Set-of-Mark prompting (e.g., OmniParser [39]) is stable but costly and sensitive to noise, such as when the page contains too many elements, while GUI grounding models (e.g., [17, 20]) are fast but unstable, especially in out-of-distribution settings such as unseen websites. Inspired by ScreenSeeker [29], WEBTESTPILOT combines these approaches to achieve a balance of stability and efficiency. Given $\text{Action}_{\text{NL}}$ and the full page screenshot in the current state s , WEBTESTPILOT uses a GUI grounding model (UI-Venus-7B [20]) to predict coarse target coordinates (x, y) , and then apply Set-of-Mark prompting to annotate all interactable elements on the screenshot with bounding boxes and IDs by analyzing the tree of UI elements in s . A square crop centered at (x, y) , together with $\text{Action}_{\text{NL}}$, is fed to an LLM, which outputs precise executable actions $\langle t, e, p \rangle$, where $t \in \{\text{click, type, press, scroll, wait}\}$, e is the element ID, and p are action parameters. This approach mitigates GUI grounding instability by using it for predicting coarse approximate locations, while reducing cost and improving effectiveness by focusing the LLM on a cropped screenshot of the target region.

4.3.2 Page Reidentification. To store a new state s' in τ , WEBTESTPILOT assigns it a page_id so that later states can be recognized as referring to the same logical page (e.g., returning to the shopping cart page). WEBTESTPILOT first selects s'' from τ with the smallest DOM tree edit distance to s' , then it provides screenshots of s' and s'' to an LLM to decide if they belong to the same page. If so, $s'.\text{page_id} = s''.\text{page_id}$; otherwise, a new page_id is assigned. WEBTESTPILOT also generates a textual representation $\text{string}(s') = (\text{page_id}, \text{summary}, \text{layout})$ and appends s' to τ .

4.3.3 Retry on Assertion Failure. When a predicate assertion fails, WEBTESTPILOT can regenerate it and retry up to n times to reduce the possibility of LLM hallucination. Alternatively, it can generate n candidate predicates upfront and resolve the outcome via majority voting. As long as the assertion holds, the reliability of test results scales with n . In this work, we use $n = 1$.

5 EXPERIMENTS

We design our experiments to answer the following research questions:

- **RQ1 (Test Flow Completion):** How effectively does WEBTESTPILOT generate test trajectories that align with human-authored test scripts, compared to baseline GUI testing agents?
- **RQ2 (Bug Detection):** How effective is WEBTESTPILOT at detecting visual and functional faults during GUI testing, relative to existing agent-based baselines?
- **RQ3 (Robustness Evaluation):** To what extent can WEBTESTPILOT generate correct test cases when provided with requirements expressed in a varied, unstructured, or freely written natural language, without relying on a fixed input format?
- **RQ4 (Model Comparison)** How much does WEBTESTPILOT’s overall performance depend on the capabilities of the underlying language model, and to what extent can its refinement mechanisms compensate when using a lightweight and cost-effective LLM?

5.1 Benchmark Construction

To the best of our knowledge, existing automated UI testing benchmarks primarily target Android mobile applications [23, 32, 60, 62, 79]. While there are datasets for web navigation [28], UI understanding, and test scripts [13] they do not focus on E2E bug detection. To address this gap, we construct a benchmark of live web applications for E2E test execution and bug detection. The construction proceeds as follows. First, we identify a set of candidate web applications $\{\mathcal{W}_1, \dots, \mathcal{W}_n\}$ that satisfy our selection criteria. Next, for each application \mathcal{W}_i , we curate a set of natural-language test requirements $\{D_{i1}, \dots, D_{im}\}$ by referring to the application’s user documentation and extracting its key functional features. For each requirement D_{ij} , we define an executable test script $\mathcal{A}_{ij} = \langle \alpha_{ij}^1, \dots, \alpha_{ij}^k \rangle$ consisting of test assertions, where each assertion $\alpha_{ij}^k : \mathcal{S} \rightarrow \{\top, \perp\}$ is a predicate over system states. Each assertion corresponds to a single expected test step to be parsed from D_{ij} by the evaluated automated tester T , and serves as a ground truth evaluation oracle for assessing the correctness of its execution trace τ . To evaluate bug detection, we additionally inject a bug for each D_{ij} in the form $\text{bug}_{ij} : \mathcal{S} \rightarrow \mathcal{S}$. When applied to a state s_t , it either leaves the state unchanged if the bug should not trigger, or produces a modified state s'_t with buggy behavior. In summary, each benchmark sample is represented as a tuple $(\mathcal{W}, D, \mathcal{A}, \text{bug})$.

5.1.1 Web Applications. We search GitHub for open-source web applications and select those based on five criteria: (1) popularity, with $\geq 5,000$ stars; (2) active development, with >50 contributors and $>1,000$ commits, and a commit in the past month; (3) maturity, publicly available for >5 years; (4) practical relevance, indicated by active deployment, recognizable domain or organization, commercial support, or adoption by well-known entities; and (5) user-facing documentation describing core features. We select the following four web applications:

- **BookStack** [8]: A hierarchical documentation management platform with rich text editing.
- **Indico** [24]: An event manager for conferences, meetings, and lectures.
- **InvoiceNinja** [25]: A business-oriented invoicing platform with multi-step workflows.
- **PrestaShop** [50]: A full-stack e-commerce platform with store management feature.

We package the applications into reproducible Docker Compose environments.

5.1.2 Natural Language Test Requirements. For each application \mathcal{W}_i , we construct $\{D_{i1}, \dots, D_{im}\}$. We start by adapting verbatim extracts from user documentation, which typically provides how-to guides for key features, and write scripts that follow the “happy paths” (intended successful usage scenarios). We then extend this initial set by extrapolating additional requirements based on the Create, Read, Update, Delete (CRUD) paradigm. For example, if the documentation describes a book management feature, we create test flows for adding, viewing, editing, and deleting book entries. Throughout this process, we follow ISTQB Certified Tester Foundation Level (CTFL) v4.0 guidelines. This yields 100 test requirements. See Table 2 for more details.

Table 2. Overview of the benchmark and its injected bugs.

Web Application	Test Cases	Lines of Code	Example Bug (from GitHub Issues)
BookStack [8]	27	214,819	No error message shown when user does not have permission to delete attachment (bookstack/#5323).
Indico [24]	25	573,316	"Send" button is missing from request recording in lectures (indico/#239).
InvoiceNinja [25]	25	1,513,289	Generating a PDF statement for a client shows the wrong client name and address (invoiceninja/#10351).
PrestaShop [50]	23	2,234,514	Clicking a product in "All Stores" send you to the "Order" page not the "Edit Product" page (prestashop/#39044).

5.1.3 Test Scripts. For each test requirement D_{ij} , the corresponding test script \mathcal{A}_{ij} consists of sequential test assertions α_{ij}^t . During testing, at step t , if T proposes an action a_t that transitions the system from state $s_t \xrightarrow{a_t} s_{t+1}$, we evaluate $\alpha_{ij}^t(s_{t+1})$: \top if the resulting state is expected state, and \perp otherwise. This is applied sequentially over the entire execution trace τ . We implement test scripts using Playwright. Figure 11 shows an example test assertion.

```

action:      Click 'Books' link in navigation
expectation: Books listing page with title 'Books' appears
assertion:   expect(page.get_by_role('heading', name='Books')).to_be_visible()

```

Fig. 11. An example test assertion for a test step.

5.1.4 Injected Bugs. We design a single artificial bug $\text{bug}_{ij} : \mathcal{S} \rightarrow \mathcal{S}$ for each test requirement D_{ij} . These bugs induce incorrect behaviors while ensuring stable and reproducible experiments by locking application versions. To ensure realism, we examine closed GitHub issues labeled "Bug" from each application repository. From a total of 2,043 issues, we randomly sample 10%. We perform open coding on the titles and descriptions of the sampled issues to identify meaningful labels, and then conduct a thematic analysis to group these labels into broader bug categories. Two co-authors independently perform the analysis, with a third resolving any disagreements. We exclude crash bugs and purely cosmetic bugs (e.g., layout or positioning issues) that do not affect functionality, as prior work has already addressed them. Based on our analysis, we focus on four categories:

- **Missing UI elements:** Required interface components are absent, breaking feature functionality. For example, in [prestashop/#22170](#), the "Configure" button is missing for newly installed modules.
- **Data inconsistency:** Information shown to the user does not match expected values. For example, in [indico/#5197](#), the category search results include items that were previously deleted.
- **No-op actions:** User actions fail silently or have no effect. For example, in [invoiceninja/#11188](#), the filter button in "Customer > Documents" does not sort or filter and always shows the full list.
- **Navigation failures:** Pages fail to transition correctly. For example, in [prestashop/#14796](#), a logged-in user selecting any option in the back-office menu is redirected to the login page.

Our categorization aligns with prior studies on Android applications [79]. Following these categories, we manually study the source code of each benchmarked web application and implement the bug in JavaScript. The bug function bug_{ij} is invoked at every state transition during testing. It will automatically modify the system state according to its behavior.

5.2 RQ1: Test Flow Completion

In this section, we evaluate WEBTESTPILOT's ability to complete test steps on web applications by parsing natural language test requirements.

5.2.1 *Baselines.* We select three baseline agents for GUI testing, described in detail below.

- **LAVAGUE:** The most popular open-source, community-supported multi-agent approach. LAVAGUE follows a two-stage architecture: the *World Model* interprets the user’s objective in the context of the current webpage architecture to produce the next high-level instruction, while the *Action Engine* translates this instruction into executable automation code. It utilizes both the HTML DOM and a visual screenshot of the page to generate DOM-level actions. LAVAGUE focuses solely on test step completion, without verification or assertion.
- **NAVIQATE:** The first single-agent approach guided by functional descriptions. NAVIQATE operates through a three-step process: (1) *Action Planning* uses retrieval-augmented generation (RAG) to identify relevant prior tasks that guide planning; (2) *Choice Extraction* collects actionable elements from the webpage, ranks them based on relevance to the current step, and annotates their functionality; (3) *Decision Making* prompts the LLM to select an action using an annotated screenshot. Like LAVAGUE, NAVIQATE focuses only on test step completion.
- **PINATA:** The state-of-the-art (SOTA) multi-agent approach that separates planning, execution, and verification into three agents: the *Orchestrator*, *Actor*, and *Asserter*. The orchestrator manages the test flow, instructing the actor to perform UI actions and the asserter to verify outcomes. The actor grounds actions using page screenshots and executes them via code actions, while the asserter checks expected results through visual analysis. All agents share a long-term memory and operate solely on the application’s observable state.

5.2.2 *Evaluation Metrics.* Let $\tau = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ denote the execution trace produced by the automated tester T (either WEBTESTPILOT or a baseline) for a given test requirement D_{ij} . Here, s_k is the system state after step k , and $\alpha_{ij}^k \in \mathcal{A}_{ij}$ is the assertion for that step in the test script. We evaluate the effectiveness of T in completing the test using two metrics:

- **Task Completion (TC):** A test is considered complete if and only if all state transitions in the execution trace satisfy their corresponding test assertions. Formally:

$$\text{TC}_{ij} = \begin{cases} 1 & \text{if } s_k \models \alpha_{ij}^k, \forall k = 1, \dots, |\mathcal{A}_{ij}| \\ 0 & \text{otherwise} \end{cases}$$

- **Correct Trace (CT):** Measures the fraction of the test script correctly executed before the first assertion failure (prefix). It quantifies how far T progresses along the test. Formally:

$$\text{CT}_{ij} = \frac{\max \left\{ k \in \{1, \dots, |\mathcal{A}_{ij}|\} \mid s_\ell \models \alpha_{ij}^\ell \text{ for all } \ell = 1, \dots, k \right\}}{|\mathcal{A}_{ij}|}$$

5.2.3 *Experiment Setup.* For each test requirement D_{ij} in our benchmark, we let T parse it into a sequence of steps, where each step is a tuple $\text{step}_t = (\text{condition}_{\text{NL}}, \text{action}_{\text{NL}}, \text{expectation}_{\text{NL}})$. At step t , T proposes an action a_t corresponding to $\text{action}_{\text{NL}}$, transitioning the system from $s_t \xrightarrow{a_t} s * t + 1$. The evaluation environment automatically checks s_{t+1} against the step’s assertion α_{ij}^t to update the metrics (Section 5.2.2). After the test terminates, we store the execution trace τ_{ij} of T for analysis. A test terminates when the number of steps executed by T reaches $|\mathcal{A}_{ij}|$, the expected length of the corresponding test script. This termination criterion prevents unbounded execution. Tests are executed sequentially to avoid shared-state interference. For each test case, we initialize a fresh instance of the web application and restore its database to a state before the test.

5.2.4 *Results & Discussion.* Table 3 summarizes the results. WEBTESTPILOT achieves the highest TC and CT scores, both at 0.99, outperforming the best baseline by 54.7% in TC and 28.6% in CT.

Table 3. Task Completion (TC) and Correct Trace (CT) across web applications (App #1: BookStack, App #2: Indico, App #3: InvoiceNinja, App #4: PrestaShop). TOTAL denotes aggregated results across all webapps.

Approach	Task Completion (TC)					Correct Trace (CT)				
	App #1	App #2	App #3	App #4	TOTAL	App #1	App #2	App #3	App #4	TOTAL
LAVAGUE	0.85	0.32	0.80	0.57	0.64	0.93	0.49	0.88	0.78	0.77
NAVIQATE	0.78	0.44	0.60	0.30	0.54	0.92	0.61	0.78	0.46	0.70
PINATA	0.11	0.04	0.08	0.09	0.08	0.27	0.09	0.16	0.22	0.18
WEBTESTPILOT	1.00	1.00	0.98	1.00	0.99	1.00	1.00	0.99	1.00	0.99

Why is WEBTESTPILOT more effective? WEBTESTPILOT’s effectiveness stems from three reachability advantages. First, it can propose multiple actions when a test step requires them (e.g., filling multiple form fields). Second, grounding actions at the visual level using GUI grounding and SoM prompting avoids common DOM-based pitfalls such as iframes, shadow DOMs, and custom form components. Third, its two-stage action execution makes WEBTESTPILOT more robust to out-of-distribution settings and noisy or complex UIs, enabling stronger generalization.

Is WEBTESTPILOT efficient? WEBTESTPILOT achieves a median execution time of 29 seconds and consumes a median of 10k tokens per step. It is the fastest among the compared methods, outperforming LAVAGUE (33s), NAVIQATE (40s), and PINATA (38s). In terms of token consumption, it ranks second, using fewer tokens than LAVAGUE (49k) and PINATA (19k), but slightly more than NAVIQATE (9k). Overall, WEBTESTPILOT is a balance between speed and cost. This efficiency stems from its hybrid action execution design, which avoids both noisy DOM-based multi-round ranking (NAVIQATE) and multi-agent communication overhead (PINATA). Total computational cost scales linearly with the number of steps. Breaking down the costs by stage, token usage is dominated by Action Execution (33%) and Page Reidentification (32%), while execution time is primarily spent on Oracle Inference and Symbolization (34%) and Page Reidentification (39%). Page Reidentification is therefore the main bottleneck and a key target for future optimization.

Is WEBTESTPILOT maintainable? Test actions generated by WEBTESTPILOT may be fragile as web applications evolve. To mitigate this, WEBTESTPILOT can cache test action and only re-invokes the action execution pipeline in Section 4.3.1 when meaningful changes to the content/layout of the state are detected. We evaluate maintainability in a study inspired by prior work on GUI evolution [56]. We compare WEBTESTPILOT with XPath-, CSS-, and Playwright-based test scripts under UI changes. The evaluation measures each method’s ability to re-identify the same GUI widgets before and after interface updates, using five tests per application: two real-world changes (on Amazon and USPS) and three synthetic changes (on BookStack) generated using transformation techniques from [54]. WEBTESTPILOT preserves test actions in 39/40 cases, outperforming XPath (32/40), CSS (33/40), and Playwright (29/40).

5.3 RQ2: Bug Detection

In this section, we evaluate how well WEBTESTPILOT and the baselines detect injected bugs in web application, using the same benchmark as in RQ1.

5.3.1 Baselines. We exclude LAVAGUE and NAVIQATE. We directly compare WEBTESTPILOT against PINATA, the only baseline capable of bug detection.

5.3.2 Evaluation Metrics. We use step-level outcomes. Let s_{bug} denote the bug-injected state in a test and \bar{s} the set of all other states. If T generates a predicate $p(s)$, we define a true positive (TP) as $p(s_{\text{bug}}) = \perp$, a false positive (FP) as $p(s) = \perp$ for any $s \in \bar{s}$, a false negative (FN) as $p(s_{\text{bug}}) = \top$, and a true negative (TN) as $p(s) = \top$ for all $s \in \bar{s}$. Then precision = $\frac{|TP|}{|TP|+|FP|}$ and recall = $\frac{|TP|}{|TP|+|FN|}$.

Table 4. Precision and Recall for bug detection across web applications (App #1: BookStack, App #2: Indico, App #3: InvoiceNinja, App #4: PrestaShop). TOTAL denotes aggregated results across all webapps.

Approach	Precision					Recall				
	App #1	App #2	App #3	App #4	TOTAL	App #1	App #2	App #3	App #4	TOTAL
PINATA	0.31	0.20	0.26	0.29	0.26	0.70	0.68	0.68	0.70	0.69
WEBTESTPILOT	0.98	0.94	0.94	1.00	0.96	0.93	0.96	0.98	0.96	0.96

Finally, to ensure that assertions capture application behavior and to rule out coincidental matches that could be spurious TPs, we manually verify the semantic correctness of all assertions.

5.3.3 Experiment Setup. We follow the setup in Section 5.2, with two changes: (1) for each step, T now generates assertion predicates p that check against $\text{condition}_{\text{NL}}$ and $\text{expectation}_{\text{NL}}$. (2) bug_{ij} is automatically injected into the web application at the start of each test D_{ij} .

5.3.4 Results & Discussion. Table 4 summarizes the results. WEBTESTPILOT achieves both a precision and recall of 0.96, with absolute improvements of 0.70 and 0.27 over PINATA, respectively.

Why is WEBTESTPILOT more effective? Analyzing execution traces τ , we identify three key reasons why WEBTESTPILOT outperforms PINATA: (1) *Dynamic cross-state reasoning*: WEBTESTPILOT can generate and track symbolic representations on the fly across multiple states. In contrast, PINATA depends on a static memory where agents must choose in advance what information to retain, and anything unrecorded is lost. This limits reasoning in tasks where crucial information is only known later or when large amounts of data make prioritization challenging. For example, in InvoiceNinja, forgetting a single detail about multiple invoices can break assertions later. (2) *Exploration capability*: WEBTESTPILOT’s two-stage action execution supports robust navigation, while PINATA fails to reach certain UI elements (e.g., the timetable in Indico). (3) *Full-page perception*: WEBTESTPILOT processes the entire page screenshot at each state, whereas PINATA observes only visible elements without scrolling, potentially missing information in long lists, tables, or grids.

Can WEBTESTPILOT detect real-world bugs? We replicated 23 real-world bugs from GitHub issues. WEBTESTPILOT detected 22 of them, compared to 15 by PINATA. The 7 bugs not detected by PINATA were due to: missing cross-state context (1), requirement misinterpretation (2), and verifier agent hallucinations on detailed pages (4). More details in Appendix A.

5.4 RQ3: Robustness Evaluation

To test whether WEBTESTPILOT’s can generalize given all necessary information, we conduct an experiment by modifying the test requirements provided to the agent.

5.4.1 Input Transformations. We design a set of input transformations under the assumption that all essential information (i.e., the condition, action, expectation) are preserved. In other words, these transformations do not remove or alter the core semantics of the test case, but instead change how the information is expressed. We introduce the following four transformations:

- **Dropout**: Randomly removes 10% of sentences to mimic incomplete requirements.
- **Add Noise**: Adds typos, filler or informal words to simulate casual language in communication.
- **Summarize**: Produces a brief, draft-style version of the test description with abbreviations.
- **Restyle**: Rewrites it in a different documentation style (e.g., procedural, technical, narrative).

Let the transformation functions be $f_{\text{add_noise}}$, f_{dropout} , f_{restyle} , and $f_{\text{summarize}}$, each defined as $f : \mathcal{D} \rightarrow \mathcal{D}$, where \mathcal{D} denotes the space of test requirements. In other words, given $D \in \mathcal{D}$, each transformation produces a modified test requirement $f(D) \in \mathcal{D}$. We implement these functions by prompting LLMs to perform an initial guided transformation, followed by heuristic post-processing

to produce the final output. For example, for **Add Noise**, we apply typo-generation libraries (e.g., typo, nlpaug) to introduce lexical perturbations.

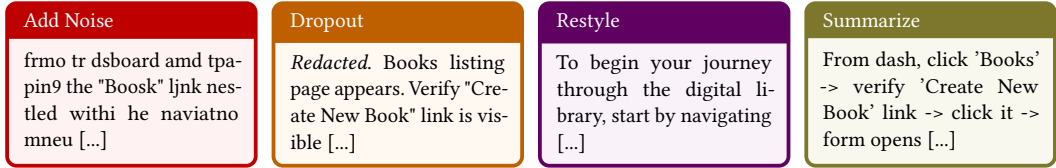


Fig. 12. Example of transformed test requirements. Original text: “Click “Books” link in navigation. Books listing page appears. Verify “Create New Book” link is visible [...]”

5.4.2 Model Selection. Beyond WEBTESTPILOT’s base model (GPT-4.1), we evaluate four open-source Qwen2.5-VL models (72B, 32B, 7B, and 3B). Qwen2.5-VL is trained with GUI grounding data, and has shown strong generalization on GUI testing and agentic benchmarks [4, 67, 79].

5.4.3 Experiment Setup. We follow the same setup as in Section 5.3, with the difference that, for each test, the input test requirement D_{ij} is first transformed using the transformation functions: f_{default} , $f_{\text{add_noise}}$, f_{dropout} , f_{restyle} , and $f_{\text{summarize}}$. We use metrics defined in Section 5.2 and 5.3.

5.4.4 Results & Discussion. Table 5 shows the results. We observe that no transformation consistently reduces TC or CT for every model, indicating the absence of a universal “worst-case” transformation. For instance, DO reduces TC for Qwen2.5-VL-32b from 0.65 (DF) to 0.63, while Qwen2.5-VL-7b remains largely unaffected at 0.70. This suggests that each model exhibits its own strengths and weaknesses, reacting differently to various transformations: GPT-4.1 maintains high performance under noise (AN = 0.93) but is more impacted by DO, RS, and SU (0.70 each), whereas smaller models like Qwen2.5-VL-3b are highly sensitive to noise (AN = 0.48, SU = 0.41).

In general, performance declines as model size decreases, but the decline is not uniform. Initially, reducing model size by roughly half results in modest performance drops of less than 10%. However, 7B is a critical threshold where TC and CT start to decline sharply by 20–30%. Thus, we suggest that for cost considerations, 7B models may serve as the minimum viable option, whereas for performance, local models should be at least 72B parameters to reliably match or exceed GPT-4.1.

Finally, there is a noticeable gap between TC and CT, with differences ranging from 0.05 to 0.09 across models. We observe that models can omit or introduce redundant steps in transformed test requirements, leading to errors in trace execution. For example, details about filling a form (how many fields, what expected input) are not interpreted correctly during parsing.

There are two key takeaways. First, all necessary information must be present in the requirements, as accurate input parsing is the strongest predictor of downstream task performance. Second, style and formatting variations can be overcome by designing specialized semantic parsers tailored to the specific domain and language style (e.g., PRD parsers, email parsers, or Slack/chat message parsers) that restructure inputs into step sequences that are correct, complete, and concise.

5.5 RQ4: Model Comparison

We perform an ablation study of WEBTESTPILOT by replacing its base model with alternative LLMs of varying sizes and capacities.

5.5.1 Experiment Setup. We follow the setup and metrics in Section 5.4, but we evaluate only WEBTESTPILOT and vary its underlying model on the benchmark without transformation.

Table 5. Task Completion (TC) and Correct Trace (CT) across test requirement transformations, evaluated using WEBTESTPILOT. TOTAL denotes aggregated results across all transformations. Abbreviations: DF = Default, AN = Add Noise, DO = Dropout, RS = Restyle, SU = Summarize.

Model	Task Completion (TC)						Correct Trace (CT)					
	DF	AN	DO	RS	SU	TOTAL	DF	AN	DO	RS	SU	TOTAL
GPT-4.1	1.00	0.93	0.70	0.70	0.70	0.81	1.00	0.95	0.81	0.78	0.78	0.86
QWEN2.5-VL-72B	1.00	0.93	0.85	0.70	0.81	0.85	1.00	0.95	0.89	0.82	0.85	0.90
QWEN2.5-VL-32B	0.65	0.89	0.63	0.78	0.81	0.75	0.76	0.94	0.76	0.85	0.88	0.84
QWEN2.5-VL-7B	1.00	0.74	0.70	0.70	0.63	0.72	1.00	0.84	0.84	0.83	0.74	0.83
QWEN2.5-VL-3B	1.00	0.48	0.52	0.70	0.41	0.57	1.00	0.61	0.66	0.77	0.47	0.66

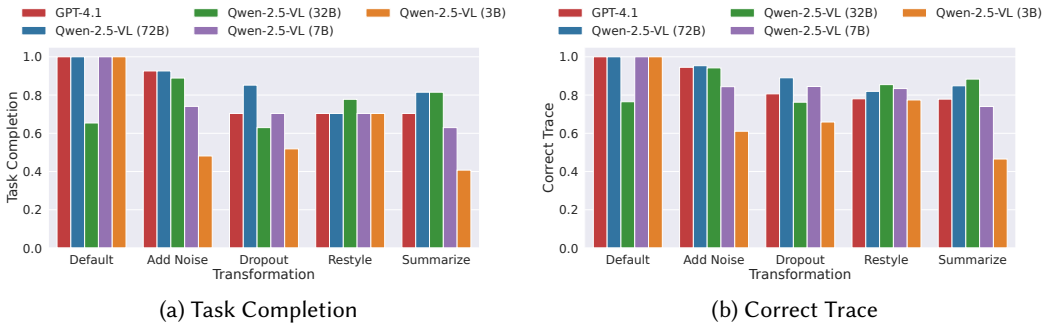


Fig. 13. Performance of different models (RQ4) under different transformed input requirements (RQ3).

5.5.2 *Results & Discussion.* Table 5 (default, DF column) shows that model performance is generally stable. However, the picture changes when models generate predicate assertions.

Assertion Quality. GPT-4.1 shows strong DSL usage: 91% of predicate assertions reference both prior and current states, 8% only the current state, and 0.2% non-adjacent states. Of declared symbols, 41.3% represent physical concepts (e.g., Car t) and 58.7% UI components (e.g., DropDown). Most assertions perform existence checks (membership 78%, is None 14%, list length 18%) and 51% use relational comparisons. Common built-ins include len, any, set, all, next, and reversed.

Analysis of Assertion Errors. Challenges occur in local models. On proprietary models (e.g., GPT-4.1), failures can be mitigated through prompt tuning. For local models (Qwen-2.5VL series), however, we observed the following issues despite prompt tuning:

- *Incorrect symbol declaration or usage (52% of cases):* Symbols declared but unused, or used without declaration. Some models treat BaseModel as a concrete symbol rather than an abstract schema (akin to using an abstract class), or misuse symbols to extract non-visual data (e.g., HTML).
- *Incorrect usage of the assertion DSL in Oracle Inference (39% of cases):* Hallucinated attributes, imports (e.g., state.isNotificationEnabled) and method calls (e.g., session.extract()).
- *Runtime errors in Oracle Execution (9% of cases):* Mismatched data types in equality comparisons and incorrect membership checks (e.g., in/not in applied to non-set data).

To improve local model performance, we recommend: (1) fine-tuning with DSL examples, (2) providing access to DSL references via an external source (e.g., retrieval-augmented generation), or (3) constraining outputs to be syntactically correct according to the DSL.

6 CASE STUDY

Setup. In addition to our empirical studies, we collaborated with China Mobile on its no-code platform *P*, which supports relational data modeling and drag-and-drop UI design for enterprise

Table 6. Bugs detected by WEBTESTPILOT in the case study.

#	Section	Page	Feature/Action	Bug Type	Bug Description
1	Warehouses	Warehouse Info	Table	Data	Some required fields in the table are empty.
2		Storage Area	Create	UI	Duplicate "Warehouse Name" form fields.
3		Storage Unit	Search	Data	Dropdown options for "Warehouse" is inconsistent with available warehouses.
4	Receipts	Receipt Info	Table	Nav	Clicking "Details" leads to an error page.
5	Assets	Inventory	Search Form	Data	Dropdown options for "Warehouse" are not bound to the actual table data.
6		Inventory	Search Form	Data	Dropdown options for "Storage Area" are not bound to the actual table data.
7		Asset	Search Form	Data	Dropdown options for "Supplier" is empty.
8	Device Management	Cameras	Search Form	UI	Query field names are incorrect.

applications. Its target users are non-technical staff who build internal enterprise applications. We were granted access to an in-progress warehouse management system w , and converted its PRD into individual requirements d for WEBTESTPILOT to check consistency against w . In total, WEBTESTPILOT uncovered eight bugs (Table 6).

Results. Of the eight bugs, five (62.5%) were data-binding issues, two were UI issues, and one was a navigation issue, demonstrating WEBTESTPILOT's strength in detecting data-related bugs often missed by baselines. Technically, PINATA is limited to detecting UI and navigation issues (3/8) and would catch data issues only if explicitly specified. The PRD was pre-processed into 56 test inputs (6.4 mins), and all tests ran in 32.7 mins. WEBTESTPILOT's mean time to detect (MTTD) was 4.9 mins, with a defect density of 0.14 bugs per page. These results show that WEBTESTPILOT provides both practical effectiveness and testing efficiency in real-world applications.

7 DISCUSSION

Threats to Validity. Internally, our metrics may underestimate tester performance, as multiple paths can achieve the same functionality. Future work could consider final page layout, content, or application state as additional indicators. Externally, our benchmark (webapp, test case, bugs) may not fully reflect real-world scenarios. However, since WEBTESTPILOT models testing as a consistency problem using a Pythonic DSL, it can handle any bug that causes behavior to diverge from requirements. A final limitation is that we assume requirements are self-contained and complete, specifying all conditions, actions, and expected outcomes in order.

Semantic Parsing for Specification-based Testing. Following our discussion above, experiments show that input parsing strategy and quality drive performance. Natural language requirements are often ambiguous, incomplete, or context-dependent. Parsing requires semantic understanding and pre-processing, not just extraction. LLM agents must act as proactive semantic parsers, transforming requirements into machine-understandable, executable representations. This includes identifying ambiguities, ask the user clarifying questions, and retrieving context from an external knowledge base where needed.

8 RELATED WORK

8.1 Automated GUI Testing

Automated GUI testing simulates user interactions (e.g., clicks) to validate application functionality via its GUI. Random techniques explore the AUT by fuzzing random actions (e.g., Monkey [45], Gremlins.js [18]) or by randomly interacting with detected widgets (White et al. [69]). Model-based approaches (e.g., Crawljax [43], ATUSA [44], Stoat [61]) construct navigational or behavioral models (e.g., flow graphs, state machines) of the AUT and derive test cases from them. To prune

redundant model states, works like Judge [31], WebEmbed [59], Corazza et al. [11], FragGen [72], and NDStudy [71] detect and remove near-duplicate states. Systemic strategies try to generate test cases that optimizes a test objective (e.g., code coverage), which can be done through search-based techniques (e.g., DIG [7], SubWeb [6], FeedEx [15], RoboTest [74], Sapienz [41], TimeMachine [14]) and symbolic execution (e.g., Apollo [3]). Reinforcement learning (RL) approaches frame testing as a sequential decision problem using Q-learning or policy optimization to guide exploration on the AUT (e.g., AutoBlackTest [42], QExplore [57], WebExplor [81], WebQT [9], WebRLED [19], UniRLTest [76], PIRL-Test [75], Hawkeye [49]). These exploration-based methods prioritize coverage over requirements. Specification-based testing uses requirements for targeted validation of user flows. Kea [70] uses a property description language to manually specify properties for Android apps. In contrast, WEBTESTPILOT automatically derives symbolic assertions from rich contextual test information. Complementary works improve test efficiency [46] and stability [33, 47, 77].

8.2 LLM for GUI Testing

Input Generation. QTYPIST [34] produces context-aware text inputs for realistic testing. INPUTBLASTER [37] mutates input strings to trigger crashes, and FORMNEXUS [1] validates form functionality via constraint-based testing. These approaches improve E2E testing reachability.

Mobile Applications. Several works, such as GPTDROID [35, 36], DROIDAGENT [73], LLMDROID, GUARDIAN [52], AUITESTAGENT [23], TRIDENT [38], A11YSCAN [78] and XUAT-COPILOT [68], focus on mobile E2E testing, using techniques like functionality-aware dialogues, coverage-guided exploration, multi-agent planning, and verification inference. Garcia et al. [16] also study how testers collaborate with LLMs in mobile testing. These works target mobile instead of web platforms.

Web Applications. Zimmermann et al. [82] and VETL [66] propose the first LLM and multimodal LLM-based GUI testing agent, respectively. AUTOAUT [42] and Leotta et al. [27] conduct feasibility studies and user interviews to understand how LLMs can support acceptance testing workflows. AXNAV [63] and UXAGENT [40] target accessibility and usability testing, respectively. These tools do not perform full E2E flow validation. AUTOE2E [2] and TEMAC [30] infer features from the application under test (AUT) and use them to drive test case generation. LLM-EXPLORER [80] maintains an abstract UI state and interaction graph to guide exploration. However, these systems primarily target coverage and do not verify expected outcomes. NAVIQATE [55] ranks actionable elements by relevance to a goal to guide interaction, but does not verify whether the final outcome satisfies the user objective. In summary, existing LLM-based web testers are limited oracles that focus on end states or explicit requirements, missing inconsistencies not captured in the specification. WEBTESTPILOT addresses this with formalized test specifications and pre/post-condition verification, enabling stable and reliable testing that also accounts for inferred implicit requirements.

9 CONCLUSION

In this work, we show that LLM agents, when paired with symbolic modeling and a DSL for formalized assertions, can serve as reliable automated GUI testers. We propose WEBTESTPILOT, which detects implicit, context-dependent bugs with high precision and recall, while remaining robust across diverse inputs and model scales.

DATA AVAILABILITY

Our benchmark, the source code of WEBTESTPILOT and baselines, and all scripts for setting up and running experiments are available at <https://github.com/code-phia/WEBTestPilot>. For more details (i.e., prompts, case study, etc.), please visit <https://sites.google.com/view/webtestpilot>.

ACKNOWLEDGEMENT

We thank the reviewers for their constructive feedback and Haozhe Wei for his contributions to the benchmark construction. This research is conducted in collaboration with China Mobile, and is supported in part by the National Natural Science Foundation of China (62572300), the Minister of Education, Singapore (MOE-T2EP20124-0017, MOET32020-0004), the National Research Foundation, Singapore and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008-1B), and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

REFERENCES

- [1] Parsa Alian, Noor Nashid, Mobina Shahbandeh, and Ali Mesbah. 2024. Semantic constraint inference for web form test generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 932–944.
- [2] Parsa Alian, Noor Nashid, Mobina Shahbandeh, Taha Shabani, and Ali Mesbah. 2025. Feature-Driven End-to-End Test Generation . 2025 *IEEE/ACM 47th International Conference on Software Engineering (ICSE) (2025)*, 450–462. <https://doi.org/10.1109/ICSE55347.2025.00141>
- [3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 261–272.
- [4] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibao Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. 2025. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923* (2025).
- [5] Kesina Baral, John Johnson, Junayed Mahmud, Sabiha Salma, Mattia Fazzini, Julia Rubin, Jeff Offutt, and Kevin Moran. 2024. Automating gui-based test oracles for mobile apps. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 309–321.
- [6] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. 2017. Search based path and input data generation for web application testing. In *International Symposium on Search Based Software Engineering*. Springer, 18–32.
- [7] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 142–153.
- [8] Bookstack 2015. <https://github.com/BookStackApp/BookStack>.
- [9] Xiaoning Chang, Zheheng Liang, Yifei Zhang, Lei Cui, Zhenyue Long, Guoquan Wu, Yu Gao, Wei Chen, Jun Wei, and Tao Huang. 2023. A reinforcement learning approach to generating test cases for web applications. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 13–23.
- [10] Antoine Chevrot, Alexandre Vernotte, Jean-Rémy Falleri, Xavier Blanc, Bruno Legeard, and Aymeric Cretin. 2025. Are Autonomous Web Agents Good Testers? *Proceedings of the ACM on Software Engineering 2*, ISSTA (2025), 206–228.
- [11] Anna Corazza, Sergio Di Martino, Adriano Peron, and Luigi Libero Lucio Starace. 2021. Web application testing: Using tree kernels to detect near-duplicate states in automated model inference. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–6.
- [12] Cucumber 2014. <https://cucumber.io/>
- [13] Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. 2025. E2EGit: A Dataset of End-to-End Web Tests in Open Source Projects. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 836–840.
- [14] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 481–492.
- [15] Amin Milani Fard and Ali Mesbah. 2013. Feedback-directed exploration of web applications to derive test models.. In *ISSRE*, Vol. 13. 278–287.
- [16] Boni Garcia, Maurizio Leotta, Filippo Ricca, and Jim Whitehead. 2024. Use of chatgpt as an assistant in the end-to-end test script generation for android apps. In *Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation*. 5–11.

- [17] Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. 2025. Navigating the Digital World as Humans Do: Universal Visual Grounding for GUI Agents. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=kxnoqaicsCT>
- [18] gremlin.js 2014. <https://github.com/marmelab/gremlins.js/>.
- [19] Zhiyu Gu, Chenxu Liu, Guoquan Wu, Yifei Zhang, Chenxi Yang, Zheheng Liang, Wei Chen, and Jun Wei. 2025. Deep Reinforcement Learning for Automated Web GUI Testing. *arXiv preprint arXiv:2504.19237* (2025).
- [20] Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, Xingran Zhou, Shuheng Shen, Yunfei Liu, Beitong Zhou, Changhua Meng, Tianyu Xia, Weizhi Chen, et al. 2025. Ui-venus technical report: Building high-performance ui agents with rft. *arXiv preprint arXiv:2508.10833* (2025).
- [21] <https://www.qt.io/quality-assurance/squish> 2003. <https://www.qt.io/quality-assurance/squish>
- [22] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 269–282.
- [23] Yongxiang Hu, Xuan Wang, Yingchuan Wang, Yu Zhang, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. Auitestagent: Automatic requirements oriented gui function testing. *arXiv preprint arXiv:2407.09018* (2024).
- [24] Indico 2004. <https://github.com/indico/indico>.
- [25] Invoice Ninja 2018. <https://github.com/invoiceninja/invoiceninja>.
- [26] LaVague 2024. <https://github.com/lavague-ai/LaVague>.
- [27] Maurizio Leotta, Hafiz Zeeshan Yousaf, Filippo Ricca, and Boni Garcia. 2024. Ai-generated test scripts for web e2e testing with chatgpt and copilot: A preliminary study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 339–344.
- [28] Bobo Li, Yuheng Wang, Hao Fei, Juncheng Li, Wei Ji, Mong-Li Lee, and Wynne Hsu. 2025. FormFactory: An Interactive Benchmarking Suite for Multimodal Form-Filling Agents. *arXiv preprint arXiv:2506.01520* (2025).
- [29] Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. 2025. Screenspot-pro: Gui grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*. 8778–8786.
- [30] Chenxu Liu, Zhiyu Gu, Guoquan Wu, Ying Zhang, Jun Wei, and Tao Xie. 2025. Temac: Multi-Agent Collaboration for Automated Web GUI Testing. *arXiv preprint arXiv:2506.00520* (2025).
- [31] Chenxu Liu, Junheng Wang, Wei Yang, Ying Zhang, and Tao Xie. 2025. Judge: Effective State Abstraction for Guiding Automated Web GUI Testing. *ACM Transactions on Software Engineering and Methodology* (2025).
- [32] Ruofan Liu, Xiwen Teoh, Yun Lin, Guanjie Chen, Ruofei Ren, Denys Poshyvanyk, and Jin Song Dong. 2025. GUIPilot: A Consistency-Based Mobile GUI Testing Approach for Detecting Application-Specific Bugs. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 753–776.
- [33] Xinyue Liu, Zihong Song, Weike Fang, Wei Yang, and Weihang Wang. 2024. Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests. In *Proceedings of the ACM Web Conference 2024*. 3043–3052.
- [34] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367.
- [35] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. *arXiv preprint arXiv:2305.09434* (2023).
- [36] 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*. 1–13.
- [37] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Zhilin Tian, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *Proceedings of the IEEE/ACM 46th International conference on software engineering*. 1–12.
- [38] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Seeing is Believing: Vision-driven Non-crash Functional Bug Detection for Mobile Apps. *arXiv preprint arXiv:2407.03037* (2024).
- [39] Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. Omniparser for pure vision based gui agent. *arXiv preprint arXiv:2408.00203* (2024).
- [40] Yuxuan Lu, Bingsheng Yao, Hansu Gu, Jing Huang, Zheshe Jessie Wang, Yang Li, Jiri Gesi, Qi He, Toby Jia-Jun Li, and Dakuo Wang. 2025. Uxagent: An llm agent-based usability testing framework for web design. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–12.
- [41] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.

- [42] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2011. AutoBlackTest: a tool for automatic black-box testing. In *Proceedings of the 33rd international conference on software engineering*. 1013–1015.
- [43] Ali Mesbah, Engin Bozdog, and Arie Van Deursen. 2008. Crawling Ajax by inferring user interface state changes. In *2008 eighth international conference on web engineering*. IEEE, 122–134.
- [44] Ali Mesbah, Arie Van Deursen, and Danny Roest. 2011. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering* 38, 1 (2011), 35–53.
- [45] Monkey 2023. <https://developer.android.com/studio/test/other-testing-tools/monkey>.
- [46] Dario Olianias, Maurizio Leotta, Filippo Ricca, Matteo Biagiola, and Paolo Tonella. 2021. STILE: a tool for parallel execution of E2E web test scripts. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 460–465.
- [47] Yu Pei, Jeongju Sohn, Sarra Habchi, and Mike Papadakis. 2025. Non-flaky and nearly optimal time-based treatment of asynchronous wait web tests. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–29.
- [48] Sven Peldszus, Noubar Akopian, and Thorsten Berger. 2023. RobotBT: Behavior-tree-based test-case specification for the robot framework. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1503–1506.
- [49] Chao Peng, Zhengwei Lv, Jiarong Fu, Jiayuan Liang, Zhao Zhang, Ajitha Rajan, and Ping Yang. 2024. Hawkeye: Change-targeted testing for android apps based on deep reinforcement learning. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 298–308.
- [50] Prestashop 2007. <https://github.com/saleor/saleor>.
- [51] Progressive Web Apps Market Size, Share & Trends Analysis Report, 2024–2030 2024. <https://www.grandviewresearch.com/industry-analysis/progressive-web-apps-pwa-market-report>
- [52] Dezhi Ran, Hao Wang, Zihong Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A runtime framework for LLM-based UI exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 958–970.
- [53] RSpec 2007. <https://rspec.info/>
- [54] Sabiha Salma, SM Hasan Mansur, Yule Zhang, and Kevin Moran. 2024. GuiEvo: Automated Evolution of Mobile App UIs. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 335–347.
- [55] Mobina Shahbandeh, Parsa Alian, Noor Nashid, and Ali Mesbah. 2024. Naviqate: Functionality-guided web application navigation. *arXiv preprint arXiv:2409.10741* (2024).
- [56] Fei Shao, Rui Xu, Wasif Haque, Jingwei Xu, Ying Zhang, Wei Yang, Yanfang Ye, and Xusheng Xiao. 2021. Webevo: taming web application evolution via detecting semantic structure changes. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 16–28.
- [57] Salman Sherin, Asmar Muqet, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. 2023. QExplore: An exploration strategy for dynamic web applications using guided search. *Journal of Systems and Software* 195 (2023), 111512.
- [58] State of Software Quality Report 2024. <https://katalon.com/reports/state-quality-2024>
- [59] Andrea Stocco, Alexandra Willi, Luigi Libero Lucio Starace, Matteo Biagiola, and Paolo Tonella. 2023. Neural embeddings for web testing. *arXiv preprint arXiv:2306.07400* (2023).
- [60] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1115–1137.
- [61] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 245–256.
- [62] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 119–130.
- [63] Maryam Taeb, Amanda Swearngin, Eldon Schoop, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2024. Axnav: Replaying accessibility tests from natural language. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.
- [64] The Failed Launch Of www.HealthCare.gov 2016. <https://d3.harvard.edu/platform-rctom/submission/the-failed-launch-of-www-healthcare-gov/>
- [65] The Payroll System That Cost Queensland Health AU1.25 Billion [n. d.]. <https://www.henricodolfig.com/2019/12/project-failure-case-study-queensland-health.html>
- [66] Siyi Wang, Sinan Wang, Yujia Fan, Xiaolei Li, and Yepang Liu. 2024. Leveraging large vision-language model for better automatic web GUI testing. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 125–137.

- [67] Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, et al. 2025. MMBench-GUI: Hierarchical Multi-Platform Evaluation Framework for GUI Agents. *arXiv preprint arXiv:2507.19478* (2025).
- [68] Zhitao Wang, Wei Wang, Zirao Li, Long Wang, Can Yi, Xinjie Xu, Luyang Cao, Hanjing Su, Shouzhi Chen, and Jun Zhou. 2024. Xuat-copilot: Multi-agent collaborative system for automated user acceptance testing with large language model. *arXiv preprint arXiv:2401.02705* (2024).
- [69] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 307–317.
- [70] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and practical property-based testing for android apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 53–64.
- [71] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 186–197.
- [72] Rahul Krishna Yandrapally and Ali Mesbah. 2022. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1086–1101.
- [73] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2024. Intent-driven mobile gui testing with autonomous large language model agents. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 129–139.
- [74] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Practical non-intrusive GUI exploration testing with visual-based robotic arms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [75] Shengcheng Yu, Chunrong Fang, Xin Li, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Effective, platform-independent gui testing via image embedding and reinforcement learning. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–27.
- [76] Shengcheng Yu, Chunrong Fang, Yulei Liu, Ziqian Zhang, Yexiao Yun, Xin Li, and Zhenyu Chen. 2022. Universally Adaptive Cross-Platform Reinforcement Learning Testing via GUI Image Understanding. *arXiv preprint arXiv:2208.09116* (2022).
- [77] Haonan Zhang, Lizhi Liao, Zishuo Ding, Weiyi Shang, Nidhi Narula, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2024. Towards a Robust Waiting Strategy for Web GUI Testing for an Industrial Software System. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2065–2076.
- [78] Yuxin Zhang, Sen Chen, Xiaofei Xie, Zibo Liu, and Lingling Fan. 2025. Scenario-Driven and Context-Aware Automated Accessibility Testing for Android Apps. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 630–630.
- [79] Kangjia Zhao, Jiahui Song, Leigang Sha, Haozhan Shen, Zhi Chen, Tiancheng Zhao, Xiubo Liang, and Jianwei Yin. 2024. Gui testing arena: A unified benchmark for advancing autonomous gui testing agent. *arXiv preprint arXiv:2412.18426* (2024).
- [80] Shanhui Zhao, Hao Wen, Wenjie Du, Cheng Liang, Yunxin Liu, Xiaozhou Ye, Ye Ouyang, and Yuanchun Li. 2025. LLM-Explorer: Towards Efficient and Affordable LLM-based Exploration for Mobile Apps. *arXiv preprint arXiv:2505.10593* (2025).
- [81] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 423–435.
- [82] Daniel Zimmermann and Anne Koziolk. 2023. Gui-based software testing: An automated approach using gpt-4 and selenium webdriver. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 171–174.

A DETECTION OF REAL-WORLD BUGS

Table 7. Real-world bugs detected by WEBTESTPILOT, replicated from GitHub issue trackers.

#	App	Bug Description	Issue
1	Bookstack	Importing books over file size limit fails silently without error	#5612
2	Bookstack	Blank lines disappearing after saving	#5344
3	Bookstack	Sorting pages	#5074
4	Bookstack	Internal server error when creating more than one new user	#4862
5	Bookstack	Page does not scroll to section when clicking on title in navigation	#4330
6	Indico	Empty calendar when using back button	#3499
7	Indico	Duplicate results in search	#5287
8	Indico	Error when trying to send an email notification about a survey	#6667
9	Indico	Category search results list contains categories that have been previously deleted	#5197
10	Invoiceninja	Invoice preview does not update after changing "surcharge" fields	#4072
11	Invoiceninja	Clicking on a hyperlink opens a test installation page in the "Invoice Design" page	#4896
12	Invoiceninja	Customer documents filter is not working	#11188
13	Invoiceninja	The "Last Year" option in reports uses the current year instead of the last year	#10876
14	Invoiceninja	Issue viewing or downloading documents on invoices	#10317
15	Invoiceninja	Can not edit clients	#9809
16	Invoiceninja	Incorrect payment total in statements	#10769
17	Prestashop	"Configure" button is missing in Catalog module	#22170
18	Prestashop	Missing reset button in "Theme & Logo"	#18893
19	Prestashop	The "Close" button is not working in the "Upload" module modal	#33629
20	Prestashop	The "Upgrade" button is not visible even though a new version of the module is available	#32497
21	Prestashop	Can not use the back-office, redirect to login page	#14796
22	Prestashop	Incorrectly calculated prices in the cart due to rounding errors	#25788

Received 2025-09-12; accepted 2025-12-22