WebTestBench: Can LLMs Perform End-to-End Web Testing?

Anonymous ACL submission

Abstract

The emergence of Large Language Models (LLMs) has driven automated web application development. However, verifying whether these generated applications reliably satisfy user requirements remains a major challenge. Unlike static code review, web testing must validate application behavior through real dynamic interactions. However, whether LLM-based agents can fulfill this testing requirement remains unexplored. To address this gap, we introduce WebTestBench, a benchmark for evaluating end-to-end automated web testing. We decompose the testing process into two cascaded sub-tasks, checklist generation and defect detection, and propose WebTester, a unified evaluation framework for this task. Evaluating representative LLMs reveals sobering results, as all models score below 30%, exposing a substantial gap between current LLMs capabilities and the demands of industrial-grade web testing deployment. Our dataset and code are available at `https://anonymous.4open.science/r/WebTestBench-Preview-E9D5`.

1 Introduction

Web applications serve as an indispensable pillar of modern society, supporting diverse daily activities by delivering rich content and interactive services (Common Crawl, 2026). Despite their ubiquity, developing web applications is labor-intensive and time-consuming, requiring developers to master diverse technical skills (Borges and de Souza, 2024) and coordinate across specialized roles such as coding, testing, and documentation (Liang et al., 2022). Recently, the rise of LLMs has alleviated this complexity by promoting efficient coding (Fried et al., 2023; Li et al., 2023; Jiang et al., 2024) and catalyzing the emergence of agentic coding systems (*e.g.*, Claude Code (Anthropic, 2026) and Codex (Openai, 2026)). These systems have established a mature *"generate → review → edit"* paradigm for general coding tasks, now extending to web development.

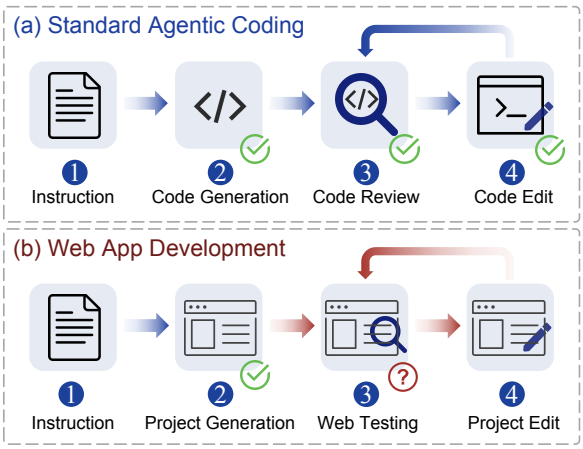


Figure 1: Agentic coding has evolved into an iterative "generate → review → edit" loop, while web development still lacks a mature testing stage. Our work studies this missing component by benchmarking web testing capability, aiming to facilitate future development cycle.

Recent advances have demonstrated remarkable progress in the "generate" stage of web development, enabling users to build complete applications from natural language instructions (Lu et al., 2025b; Wan et al., 2025). However, verifying whether these generated applications reliably satisfy user requirements remains a critical challenge. Unlike conventional code review, which statically examines source code, web testing needs to assess quality from user perspective by dynamic interactions, checking whether features behave as intended, and whether the rendered experience faithfully reflects the user intent. Consequently, as generation capabilities mature, the primary bottleneck has shifted to automated web testing (Le et al., 2025; Ye et al., 2025). Whether LLMs can fulfill this dynamic testing requirement remains insufficiently explored (Figure 1).

Despite this need, current research struggles to provide sufficient evaluation frameworks for automated web testing. Existing efforts can be catego-

Benchmark	#Sample	Env	M.P.	Checklist	Func.	Cons.	Inte.	L.H.	Agentic	Tool
Application Quality Evaluation Methods Originate from Web Generation Tasks										
Interaction2Code (Xiao et al., 2024)	127	Web	✗	✗	✓	✗	✓	✗	✗	✗
Design2Code (Si et al., 2025)	484	Web	✗	✗	✗	✗	✗	✗	✗	✗
WebGen-Bench (Lu et al., 2025b)	101	Web	✓	✗	✓	✗	✓	✓	✓	✗
ArtifactsBench (Zhang et al., 2025)	1,825	Web	✗	✗	✓	✗	✓	✓	✗	✗
WebCoderBench (Liu et al., 2026)	1,572	Web	✗	✗	✓	✗	✓	✗	✗	✗
Application Testing Benchmarks										
GTArena (Zhao et al., 2024a)	10,844	Mobile	✓	✓	✓	✗	✓	✗	✗	✗
GUITestBench (Gao et al., 2026)	143	Mobile	✓	✗	✓	✗	✓	✓	✓	✗
PlanATA (Chevrot et al., 2025)	113	Web	✓	✗	✓	✗	✓	✓	✓	✗
WebTestPilot (Teoh et al., 2026)	100	Web	✓	✗	✓	✗	✓	✓	✓	✗
WebTestBench	100	Web	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison with representative related benchmarks. **#Sample**=number of samples, **Env**=application environment, **M.P.**=project-level multi-page testing, **Checklist**=end-to-end testing without manually predefined checklist, **Func.**=functional correctness testing, **Cons.**=latent logical constraints testing, **Inte.**=dynamic interaction with the app, **L.H.**=long-horizon interactions, **Agentic**=agentic testing, **Tool**=testing conducted through tool calling.

rized into two approaches. The first relies on static evaluation methods, assessing generation quality through visual similarity (Si et al., 2025; Xiao et al., 2025) or isolated component-level checks (Xiao et al., 2024). A more realistic approach employs scripts or computer-use agents (CUAs) to simulate human behavior and perform functional verification (Lu et al., 2025b; Zhang et al., 2025; Zhao et al., 2024a). However, these methods face two critical limitations. First, they rely heavily on predefined, human-written test checklists. This reliance fundamentally contradicts the vision of automated development, where users should verify generated applications without manually authoring test cases. Second, existing works overlook implicit logical constraints beyond functional correctness, which are rarely specified explicitly in development instructions yet critical to application quality. For instance, in a meeting room reservation system, the same room cannot be double-booked for overlapping time slots. Consequently, whether LLMs and CUAs possess the capability to perform end-to-end web testing by autonomously generating checklists and detecting defects without human intervention remains an open question.

To address these gaps, we introduce **WebTestBench**, a benchmark for end-to-end automated web testing, requiring agents to generate checklist and perform defect detection. To ground evaluation in realistic AI-driven web development scenarios, we synthesize 100 web applications spanning seven categories. We evaluate web testing performance along four quality dimensions, including *functionality*, *constraint*, *interaction*, and *content*. Beyond functional verification and content relevance, WebTestBench also emphasizes on latent logical constraints, a critical yet overlooked dimension. To evaluate LLM performance on WebTestBench, we design **WebTester**, a two-stage framework for end-to-end web testing comprising a *Checklist Generation Agent* and a *Defect Detection Agent*. Building on this unified framework, we conduct a comprehensive evaluation of several powerful LLMs. The results are sobering, as all evaluated models score below 30% on WebTestBench, revealing a substantial gap between current LLMs capabilities and the demands of industrial-grade web testing deployment. Our analysis identifies three primary bottlenecks: (1) *Incomplete test coverage:* over 30% of test cases are omitted, leaving substantial defects entirely untested; (2) *Detection bias:* models default to either conservative pass judgments or aggressive false alarms; and (3) *Long-horizon instability:* comprehensive testing requires dozens of interaction turns and millions of tokens, posing demands on long-context memory and planning stability. In summary, our contributions are:

- We introduce WebTestBench, a meticulously annotated web testing benchmark that simulates end-to-end automated testing scenarios for modern AI-driven web development, accompanied by a suite of evaluation metrics.
- We design WebTester, a unified evaluation framework for end-to-end automated web testing built upon the WebTestBench evaluation protocol.
- We conduct a comprehensive evaluation of representative LLMs, uncovering their strengths and weaknesses across various dimensions and identifying consistent failure patterns. Hopefully, these findings provide valuable guidance for advancing automated web testing in the era of vibe coding.

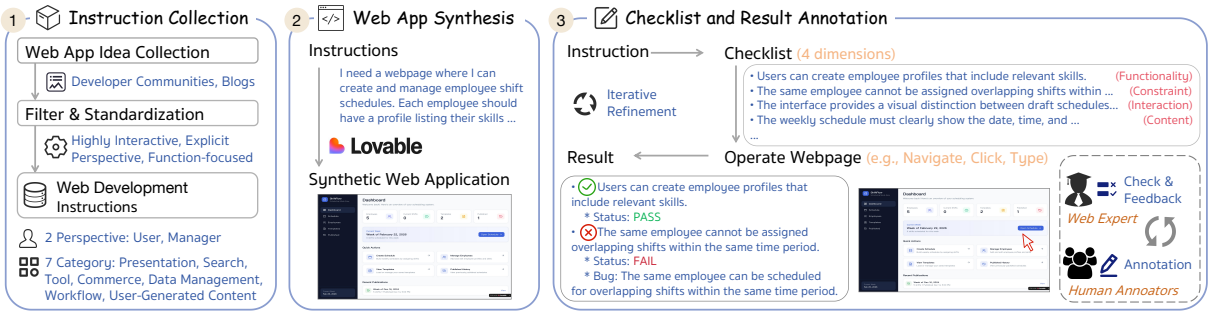


Figure 2: Overview of WebTestBench construction.

2 Related Work

Web Generation and Evaluation. LLMs have catalyzed a paradigm shift in programming, evolving from code completion (Fried et al., 2023; Li et al., 2023) to coding agents (Yang et al., 2024; Gao et al., 2025) and project synthesis (Zan et al., 2024; Zhao et al., 2024b). Two primary paradigms have emerged for automated web generation: (1) *UI-to-Code* (Yun et al., 2024; Wu et al., 2024) replicates designs from screenshots, and (2) *Language-to-Code* (Wan et al., 2025) generates applications from user instructions. Early benchmarks rely on static checks, such as visual similarity (Xiao et al., 2025; Lin et al., 2025b) or isolated component-level checks (Xiao et al., 2024), neglecting dynamic interactions. Recently, ArtifactsBench (Zhang et al., 2025) uses interaction screenshots, while WebGen-Bench (Lu et al., 2025b) employs agents for multi-step interactive tests.

Web Testing Benchmarks. As web generation capabilities mature, systematic evaluation of testing agents becomes critical to close the development loop. Motivated by the interactive capabilities of CUAs (Zhou et al., 2024; Xie et al., 2024), recent benchmarks have emerged to evaluate automated web testing. GTArena (Zhao et al., 2024a) decomposes testing into intention generation, execution, and detection, but limits detection to atomic-level bugs from single-step comparisons. GUITestBench (Gao et al., 2026) focuses on exploratory defect discovery. More recent works (Chevrot et al., 2025; Teoh et al., 2026) investigate whether agents can execute human-written test cases and accurately determine defect verdicts, advancing the evaluation of agent execution reliability.

Despite this progress, these benchmarks share a fundamental assumption that human-authored checklists are provided. While they evaluate whether agents can execute given test cases, they sidestep an equally important question of whether agents can generate comprehensive test cases. Furthermore, they focus on explicit functional requirements while overlooking implicit logical constraints. WebTestBench addresses these by introducing checklist-free end-to-end web testing, evaluating whether agents can generate complete checklists and detect defects across diverse dimensions.

3 WebTestBench

The advent of automated web generation enables users to build web applications via natural language, yet the testing stage remains underdeveloped. To systematically evaluate whether agents can perform end-to-end web testing, we introduce WebTestBench. In this section, we define the end-to-end automated web testing task, followed by our data construction and evaluation protocol.

3.1 Task Definition

Given a user instruction and a corresponding web application, the target is to evaluate the end-to-end automated web testing capability of computer-use agents. Specifically, each agent is required to complete a two-stage task: **(1) Checklist Generation.** The agent decomposes the instruction into a set of verifiable test items, $\mathcal{T}_{\text{pred}} = \{p_1, \ldots, p_m\}$, where each test item p_j describes a concrete behavior or property of the web application that can be validated through interaction. **(2) Defect Detection.** The agent plans and executes all test items by simulating human interactions (*e.g.*, clicking, typing) within the web environment, and determines an execution status $s_j \in \{\text{Pass}, \text{Fail}\}$ for each item. The final output of the agent is a set of predicted test item-result pairs $\mathcal{P} = \{(p_j, s_j)\}_{j=1}^{m}$, which jointly represent the evaluation of the web application.

3.2 Data Construction

Figure 2 illustrates the workflow for WebTestBench construction, primarily consisting of instruction

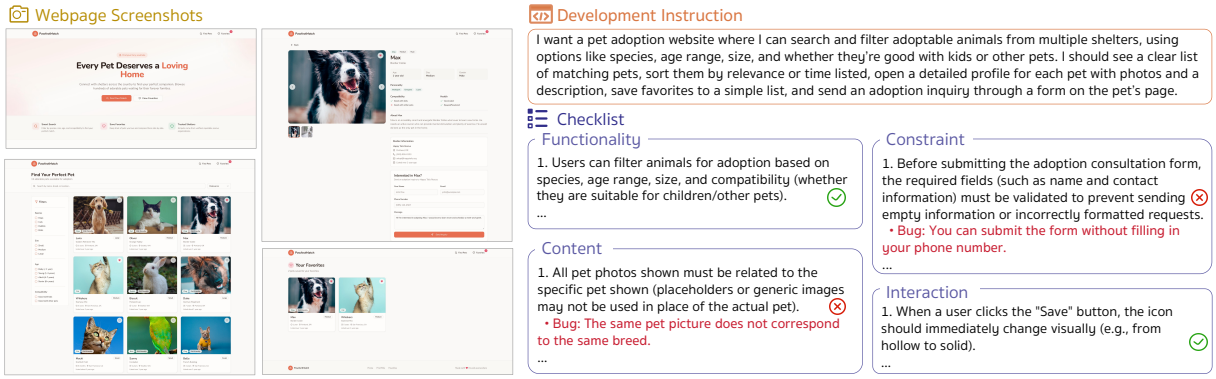


Figure 3: An illustrative example from WebTestBench. The web application is built from a given development instruction, typically comprising multiple pages with rich interactive functionality. The benchmark provides a gold checklist across four quality dimensions: Functionality, Constraint, Interaction, and Content. Each test case is annotated with a binary Pass/Fail verdict.

collection, web application synthesis, and checklist and result annotation.

Instruction Collection. To ensure the diversity and practical relevance of our benchmark, we initially collect 451 trending web application ideas from diverse developer communities and blogs. We eliminate highly similar duplicates and filter these raw ideas to isolate those aligned with our evaluation objectives, prioritizing attributes such as interactivity, practicality, and testability.

These ideas range from brief design sketches to overly verbose descriptions. Sketches are often too vague, while detailed descriptions include irrelevant content such as background narratives, references to commercial websites, and advertisements. We therefore rewrite each idea into a well-formed instruction using `GPT-5.1`. The model assigns each idea a specific perspective (*i.e.*, *User* or *Administrator*) to disambiguate permission logic, categorizes it into one of seven functional themes (*i.e.*, *Presentation*, *Search*, *Tool*, *Commerce*, *Data Management*, *Workflow*, and *User-Generated Content*), and produces a formal development instruction. Subsequently, human annotators review and refine these instructions to ensure quality and standardization. Finally, we sample 100 instructions and maintain a balanced distribution across all categories.

Web Application Synthesis. Evaluating web testing agents requires environments that are ecologically valid and contain diverse and non-trivial defects. However, standard web resources often present limitations. Commercial websites are typically well tested and continuously updated, which makes them unsuitable as benchmark samples. Open-source projects often feature simple designs with shallow structures or limited interactive functionality, and therefore fail to reflect realistic user interactions. To bridge this gap, we utilize `Lovable.dev`[1], an AI-powered web development platform that generates complete websites from user instructions, to synthesize web application projects. Through this process, we obtain an initial web application for each instruction, providing a realistic webpage for defect detection.

Gold Checklist and Result Annotation. Given a development instruction and its application, human annotators construct a testable checklist. Inspired by software quality and evaluation standards (*e.g.*, ISO/IEC 25010 (ISO/IEC, 2023)), and adapting them to the features of our benchmark, we categorize test cases into four dimensions: *Functionality*, *Constraint*, *Interaction*, and *Content*. Beyond isolated atomic interaction tests, we emphasize long-horizon interactions and latent logical constraints. For example, in a "meeting room booking" system, a functional test item would not merely be "click a button", but a complete sequence: "select a specific room, complete the reservation form, and verify the successful status update." Additionally, potential logical checks are also necessary, such as "the same meeting room cannot be reserved by two different users for the overlapping time slots."

Annotators derive a checklist from the instruction, and then interact with the actual application to align test cases with the implemented components and interaction flows. Finally, annotators execute the checklist by interacting with the website and document the Pass/Fail status of each item and provide concise bug reports for failures.

Iterative Refinement. Initial synthesis occasionally produces applications with relatively few de-

[1] https://lovable.dev/

fects, which limits the effectiveness in discriminatively evaluating web testing capabilities. To address this, annotators perform iterative refinement by revising the development instructions and regenerating the webpages, typically by introducing additional functional requirements. As instruction and webpage evolve, the checklist and defect annotations are synchronously updated until the samples contain sufficient defects for evaluation.

Quality Control. To ensure the quality of the dataset, all annotators undergo related training and conduct cross-validation during the annotation process. Finally, a senior annotation leader (non-authors) performs a final scan of the entire dataset, providing feedback and guiding annotators to optimize the annotations. This process ensures the standardization and high quality of the annotations.

3.3 Evaluation Metrics

Let $\mathcal{G} = \{(g_i, r_i)\}_{i=1}^{n}$ denote the gold outcome set, where g_i represents a gold test item and $r_i \in \{\text{Pass}, \text{Fail}\}$ is its ground-truth status. To quantify performance, we compare the agent's outcome set $\mathcal{P}$ against an annotated gold outcome set $\mathcal{G}$. Specifically, for a generated checklist $\mathcal{T}_{\text{pred}} = \{p_1, \ldots, p_m\}$ and a gold checklist $\mathcal{T}_{\text{gold}} = \{g_1, \ldots, g_n\}$, each predicted item p_j is mapped to g_i if they are semantically equivalent in testing intent. We define a matching function $\phi : \mathcal{T}_{\text{pred}} \rightarrow \mathcal{T}_{\text{gold}} \cup \{\varnothing\}$, where $\phi(p_j) = g_i$ indicates that p_j is matched to g_i. We define the matched set of a gold item g_i as $\Phi(g_i) = \{\, p_j \in \mathcal{T}_{\text{pred}} \mid \phi(p_j) = g_i \,\}$. We employ `Qwen3.5-27B` (Qwen Team, 2026b) as a semantic judge to perform this matching step.

Coverage. Coverage measures the completeness of the generated checklist, focusing on whether all necessary test items have been listed. It is defined as the proportion of gold test items successfully covered by the agent's predictions:

$$\text{Coverage} = \frac{\left|\{\, g_i \in \mathcal{T}_{\text{gold}} \mid \Phi(g_i) \neq \varnothing \,\}\right|}{|\mathcal{T}_{\text{gold}}|} \quad (1)$$

Defect Detection Performance. We treat defect detection as a binary classification task where "Fail" (a real defect) is the positive class. For each gold item g_i, we aggregate its matched predictions into a single verdict:

$$D(g_i) = \begin{cases} \text{Fail}, & \text{if } \exists\, p_j \in \Phi(g_i),\ s_j = \text{Fail} \\ \text{Pass}, & \text{otherwise} \end{cases} \quad (2)$$

where $D(g_i) = \text{Pass}$ when $\Phi(g_i) = \varnothing$. The confusion matrix entries are formalized as follows:

- *True Positive (TP):* The agent correctly identifies a ground-truth bug.

$$\text{TP} = \sum_{i=1}^{n} \mathbb{1}(r_i = \text{Fail} \wedge D(g_i) = \text{Fail}) \quad (3)$$

- *False Positive (FP):* The agent reports a defect for a gold item that is actually correct.

$$\text{FP} = \sum_{i=1}^{n} \mathbb{1}(r_i = \text{Pass} \wedge D(g_i) = \text{Fail}) \quad (4)$$

- *False Negative (FN):* The agent fails to detect a real bug, either due to misjudgment during detection or omission in checklist generation.

$$\text{FN} = \sum_{i=1}^{n} \mathbb{1}(r_i = \text{Fail} \wedge D(g_i) = \text{Pass}) \quad (5)$$

- *True Negative (TN):* The agent correctly raises no alarm for a passing gold item, either by correctly judging it as "Pass" or by not covering the item.

$$\text{TN} = \sum_{i=1}^{n} \mathbb{1}(r_i = \text{Pass} \wedge D(g_i) = \text{Pass}) \quad (6)$$

Based on these formal definitions, we can calculate Precision (P), Recall (R), and F1 score:

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}},\ R = \frac{\text{TP}}{\text{TP} + \text{FN}},\ \text{F1} = \frac{2 \times P \times R}{P + R} \quad (7)$$

3.4 WebTester

Given the current absence of an established framework for end-to-end automated web testing, we introduce WebTester, a minimal yet functional baseline designed for this task. Specifically, WebTester consists of two components: a checklist generation agent and a defect detection agent. This framework enables autonomous testing of a web application based on the user instruction and the web application, ultimately producing structured test results accompanied by bug reports.

Checklist Generation Agent. Given a development instruction I used to synthesize the target web application, the checklist generation Agent A_C is responsible for decomposing I into a structured, executable test checklist $\mathcal{T}_{\text{pred}}$. Formally, we define a test item p as a triple:

$$p = (d,\ a,\ e) \quad (8)$$

where d is a textual description of what is being tested, a is the action to be performed on the web interface (*e.g.*, clicking the "Edit" button), and e is

Model	#Turns	#Tokens	Functionality		Constraint		Interaction		Content		Overall			
			Cov.	F1	Cov.	F1	Cov.	F1	Cov.	F1	Cov.	P	R	F1
Open-Source LLMs														
Minimax-M2.1	41.5	3.57M	76.7	12.6	41.4	15.1	43.5	18.2	45.6	7.5	59.8	21.6	14.2	14.7
Qwen3-Coder-Next	63.7	6.28M	77.5	14.4	47.8	23.8	42.2	12.3	37.0	4.0	60.2	27.8	16.0	17.4
GLM-4.7	42.0	3.54M	79.7	16.7	48.1	21.0	38.1	18.9	47.7	4.9	61.5	27.5	17.2	18.6
GLM-5	41.1	3.68M	79.8	12.2	48.9	27.4	39.1	20.1	50.2	2.8	62.9	30.7	15.6	18.8
Step-3.5-Flash	57.1	3.38M	79.9	20.9	**53.8**	27.6	**49.8**	21.5	**60.4**	2.6	**66.2**	**34.9**	21.1	23.6
MiMo-V2-Flash	59.4	7.24M	79.1	21.5	48.2	**28.9**	48.5	20.6	49.6	8.2	63.2	33.8	23.5	24.1
Closed-Source LLMs														
Claude Opus 4.5	42.3	2.58M	**82.4**	17.6	49.9	20.4	41.4	15.7	42.4	5.4	62.9	32.1	15.6	19.2
Claude Sonnet 4.5	37.8	1.93M	81.0	22.2	48.7	22.0	46.5	19.4	53.1	1.6	64.0	32.8	19.3	21.8
GPT-5.2	69.6	7.45M	77.2	25.5	52.0	22.6	43.4	**24.5**	45.7	4.5	61.4	25.6	25.7	23.4
GPT-5.1	30.7	0.91M	75.6	**31.5**	49.9	25.2	48.2	19.7	59.1	**16.8**	63.0	25.4	**32.5**	**26.0**

Table 2: Web testing performance of representative LLM on WebTestBench under the WebTester framework. We report the Coverage metric (Cov.) for checklist generation, and the average number of iteration turns (#Turns), average context tokens (#Tokens) per instance, and Precision/Recall/F1 metrics for defect detection. All results are averaged over three independent runs. Results in **bold** and underline denote the best and second-best performances.

the expected outcome that determines the Pass/Fail verdict. The full checklist generated by A_C is a typed collection of such triples:

$$\mathcal{T}_{\text{pred}} = A_C(I) = \{\, p_j \}_{j=1}^{m} \tag{9}$$

Defect Detection Agent. Given the development instruction I, the generated checklist $\mathcal{T}_{\text{pred}}$, and the target web application W, the defect detection agent A_D is responsible for executing each test item $p_j \in \mathcal{T}_{\text{pred}}$ by simulating human interactions with the webpages, and determining a binary execution status for each item. Formally, $\mathcal{A}_D$ maps each test item together with the instruction and the web application to a verdict:

$$s_j = A_D(p_j,\ I,\ W) \in \{\text{Pass},\ \text{Fail}\} \tag{10}$$

For each failed item, $\mathcal{A}_D$ additionally produces a bug report b_j describing the observed erroneous behavior. The final output of WebTester is thus a set of predicted outcome triples:

$$\mathcal{S} = A_D(\mathcal{T}_{\text{pred}}, I, W) = \{(p_j, s_j[, b_j])\}_{j=1}^{m} \tag{11}$$

4 Experiments

4.1 Settings

We build WebTester on Claude Code as the basic agent framework. Specifically, we adopt the Claude Agent SDK[2] to provide a unified evaluation interface across different LLMs. To interact with the web application, we leverage Playwright MCP[3], a Model Context Protocol server that provides browser automation capabilities (*e.g.*, navigation, clicking, typing) to the agent.

[2] https://github.com/anthropics/claude-agent-sdk-python

[3] https://github.com/microsoft/playwright-mcp

We evaluate several closed-source models and open-source models, including: GPT-5.2/5.1 (Openai, 2025b,a), Claude Opus/Sonnet 4.5 (Anthropic, 2025a,b), GLM-5/4.7 (Zeng et al., 2026, 2025), Step-3.5-Flash (Huang et al., 2026), Qwen3-Coder-Next (Qwen Team, 2026a), MiMo-V2-Flash (Xiao et al., 2026), and Minimax-M2.1 (MiniMax, 2025). More details are available in Appendix B.1.

4.2 Main Results and Analysis

Overall Performance. Table 2 demonstrates the performance of the tested LLMs on end-to-end web testing. Despite advanced LLMs have made progress in agentic coding tasks and general web browsing tasks, they still exhibit notable performance bottlenecks when confronted with highly dynamic web testing scenarios involving complex interactions and constraints. All tested models failed to achieve an F1 score exceeding 30%. Among all evaluated models, GPT-5.1 attains the highest F1 score of 26.0%, a standing driven primarily by its leading recall of 32.5%. MiMo-V2-Flash ranks second with an F1 of 24.1%, achieving higher precision (33.8%).

Insufficient Test Completeness. Reliable defect detection relies on a comprehensive test checklist. However, across all tested models, their coverage consistently remains below 70%. This indicates that a substantial portion of test items are excluded from subsequent detection and introduces false negatives. Even the two strongest models, GPT-5.1 and MiMo-V2-Flash, omit at least one-third of the test cases. This reflects a fundamental limitation in how LLMs decompose development instructions into verifiable test cases, particularly for implicit cases

Model	False Negative Miss / Detect (%)		True Negative Miss / Detect (%)	
Minimax M2.1	**57.6**	42.4	**43.0**	57.0
Qwen3 Coder Next	**54.7**	45.3	**42.5**	57.5
GLM 4.7	**61.1**	38.9	**42.0**	58.0
GLM 5	**54.6**	45.4	**39.7**	60.3
Step 3.5 Flash	**53.1**	46.9	**35.6**	64.4
MiMo-V2-Flash	**60.6**	39.4	**39.2**	60.8
Claude Opus 4.5	**57.1**	42.9	**37.2**	62.8
Claude Sonnet 4.5	**58.2**	41.8	**37.9**	62.1
GPT 5.2	**67.0**	33.0	**47.5**	52.5
GPT 5.1	**73.8**	26.2	**48.5**	51.5

Table 3: Negative prediction attribution. **Miss** (%) denotes the proportion attributable to checklist coverage gaps, and **Detect** (%) denotes the proportion where the model generated and executed the corresponding test case and rendered a negative verdict.

not explicitly stated in the instructions.

Defect Detection Failures Attribution. We further analyze the sources of detection errors to better characterize failure modes. Precision reflects the model's propensity for false alarms. As shown in Table 2, precision remains consistently low across all models (21.6–34.9%), indicating that models frequently misclassify normal behaviors as defects, such as transient UI rendering delays and asynchronous state updates. In contrast, recall is jointly influenced by both checklist generation and defect detection: a real defect goes undetected either because the agent fails to generate the corresponding test item, resulting in the defect entirely unexamined, or because the agent generates the item but incorrectly judges it as non-defective.

To disentangle these two failure modes, Table 3 decomposes the causes of false-negative and true-negative predictions. Test case omission accounts for the dominant share of false negatives (53.1–73.8%), establishing that incomplete checklist generation is the primary factor leading to failed defect detection. Therefore, improving the completeness of the checklist is the most effective strategy for reducing false negatives. For true negatives, some of them also arise from the omission. The Detect ratio consistently exceeds the Miss ratio across all models, indicating that models more often identify passing behaviors through active judgment rather than by omission. This indicates that while models struggle to detect real defects, their judgment of non-defective behaviors is comparatively reliable.

Long-horizon Interaction Challenge. Completing a comprehensive web defect detection process typically requires dozens of interaction turns and millions of tokens. For example, Step-3.5-Flash requires an average of 57.1 turns and 3.38M tokens per sample. Such long-horizon tasks demand rigorous long-context memory and planning stability. Specifically, as the interaction history accumulates, models become increasingly susceptible to tracking failures, resulting in the loss of prior states or the execution of redundant operations.

Quality Dimensions. As shown in Table 2, *Functionality* achieves the highest coverage across all models, as functional items are typically explicit and extractable from the development instruction. In contrast, coverage drops substantially for the remaining three types, which involve implicit requirements demanding deeper reasoning. Notably, *Constraint* achieves relatively higher F1 despite lower coverage, since constraint violations tend to produce clear, observable signals once tested. *Content* performs worst by a marked margin, as models struggle both to generate defect-targeting items and to judge semantic alignment between displayed content and the development instruction. Detailed analysis is provided in Appendix B.4.

Correlation with Human Judgments. To validate the effectiveness and robustness of our automatic evaluation pipeline, we perform a correlation analysis between the automated evaluation results and human judgments, reporting Kendall's τ, Spearman's ρ, and Pearson r correlation coefficients. As shown in Table 4, all judge models achieve strong correlation with human judgments and Qwen3.5-27B attains the best correlation. Moreover, as an open-source model, it is not subject to deprecation or silent updates over time, ensuring long-term reproducibility and stability. We therefore adopt Qwen3.5-27B as the judge model in this work. More details are available in Appendix B.3.

Model	Kendall's τ	Spearman's ρ	Pearson r
GPT-5-Mini	68.8	75.0	78.9
GPT-5	76.6	82.7	85.7
Qwen3.5-27B	**78.5**	**83.2**	**87.1**

Table 4: Human judgment correlation scores (%) for our automatic evaluation. All p-values < 0.05.

4.3 Oracle Setting

In the end-to-end setting, checklist generation and defect detection are executed sequentially, so final performance is jointly influenced by both sub-tasks. To isolate detection capability, we evaluate models in an oracle setting, where the defect detection agent receives the annotated gold checklist directly, thereby decoupling the detection phase from the upstream checklist generation. Beyond standard

Model	Setting	Precision	Recall	F1	SPC
Step-3.5-Flash	E2E	34.9	21.1	23.6	-
	Oracle	53.4 (+18.5)	38.7 (+17.6)	40.9 (+17.3)	90.4
MiMo-V2-Flash	E2E	33.8	23.5	24.1	-
	Oracle	44.3 (+10.5)	30.4 (+6.9)	31.9 (+7.8)	91.8
Claude Sonnet 4.5	E2E	32.8	19.3	21.8	-
	Oracle	61.0 (+28.2)	47.4 (+28.1)	49.2 (+27.4)	87.6
GPT-5.1	E2E	25.4	32.5	26.0	-
	Oracle	47.0 (+21.6)	63.4 (+30.9)	46.8 (+20.8)	68.8

Table 5: Comparison of perfermance between the end-to-end (E2E) setting and the oracle setting for representative models. In the oracle setting, the gold checklist is provided to the defect detection agent, decoupling detection performance from checklist generation quality.

Precision, Recall, and F1, we additionally report Specificity (SPC $= \mathrm{TN}/(\mathrm{TN}+\mathrm{FP})$), which measures the ability to correctly identify non-defective items. More details are available in Appendix B.7.

As shown in Table 5, all models achieve notable performance improvements in the oracle setting, confirming that test incompleteness is one of the primary bottlenecks for end-to-end web testing. Beyond this trend, two notable patterns emerge.

Default-to-Pass Bias. Across most models, Precision exceeds Recall by $\sim$10% in the E2E setting (Table 2), while this gap widens to $\sim$15% in the oracle setting (Table 5). Most models adopt a *conservative* detection strategy, defaulting to a "Pass" judgment when evidence for a defect is ambiguous or insufficient. The GPT series instead pursues an *aggressive* strategy, exhibiting markedly higher Recall. However, this aggressiveness comes at a cost, as GPT-5.1 achieves the lowest Specificity, indicating a pronounced tendency to misclassify non-defective behaviors as bugs. Taken together, these results reveal a fundamental tension between defect sensitivity and false-positive suppression that no model resolves satisfactorily.

Open-Source vs. Closed-Source Models. Closed-source models not only outperform open-source models in the oracle setting, but also exhibit substantially larger gains when transitioning from E2E to oracle. For instance, Claude Sonnet 4.5 gains 28.2% in Precision and GPT-5.1 gains 30.9% in Recall, whereas MiMo-V2-Flash and Step-3.5-Flash improve by less than 20% across all metrics. This asymmetry suggests a qualitative difference in bottleneck structure: for closed-source models, checklist generation is the dominant limiting factor, and a comprehensive checklist yields substantially greater detection gains compared to open-source models. In contrast, open-source models face compounding difficulties in both sub-tasks, indicating that advancing them requires parallel progress in checklist coverage and detection reliability.

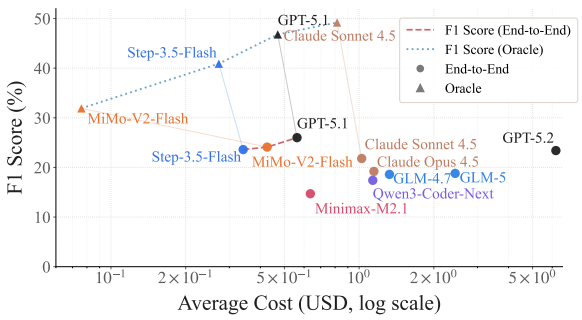


Figure 4: Cost-performance trade-off of evaluated models under End-to-End and Oracle settings.

4.4 Cost-Performance Trade-off

Figure 4 illustrates the cost-performance trade-off for all evaluated models. In the end-to-end setting, cost does not reliably predict performance. Although GPT-5.2 has the highest cost, its F1 score is only moderate. In contrast, Step-3.5-Flash and MiMo-V2-Flash achieve competitive performance at notably lower costs, making them the most cost-effective choices in practice. In the oracle setting, models with stronger detection capabilities achieve remarkable performance improvements at only a moderate additional cost. These results indicate that the current performance ceiling depends primarily not on the inference budget, but on the quality of the generated checklists and the reliability of detection, as discussed in Sections 4.2 and 4.3.

5 Conclusion

In this paper, we introduce WebTestBench, a challenging benchmark for end-to-end web testing without relying on human-written test cases, grounded in the context of automated web development. Beyond standard functional testing, WebTestBench incorporates latent logical constraints, enabling a comprehensive assessment of application quality. We further propose WebTester, a unified framework for end-to-end web testing, and evaluate representative LLMs on WebTestBench. The results reveal a substantial gap between current LLMs capabilities and industrial-grade requirements. Through comprehensive analysis, we identify three fundamental bottlenecks, including incomplete test coverage, detection bias, and long-horizon instability. We hope WebTestBench serves as a foundation for future research in web testing and enabling a complete agentic web development cycle.

Limitations

Although WebTestBench provides fine-grained annotations across four quality dimensions, the construction process is inherently labor-intensive and demands domain expertise. Annotators must reason beyond explicit requirements to identify latent logical constraints. This specialized nature makes it costly to scale or replicate our methodology for building similar benchmarks. Second, our evaluation protocol also carries an inherent limitation. The scoring system relies on aligning predicted test items against the gold checklist. Thus, it cannot assess the validity of generated test cases that fall outside the gold checklist coverage, even if such cases reflect genuine quality concerns. Finally, while WebTestBench spans seven web application categories, it does not fully capture the diversity of real-world applications. Certain application types, such as "games" and "complex multi-user real-time interactive systems", are deliberately excluded, as they demand high-frequency capture and interpretation of highly dynamic content that exceeds the perceptual latency of both DOM-based and screenshot-based approaches, and thus lies beyond the current capability boundary of CUAs.

Ethics Policy

The web development instructions collected for WebTestBench may contain background narratives or promotional content originating from real-world sources. Following our collection procedure, we remove or rewrite all such material, with human annotators verifying that no real-world entities remain. To support defect detection, we use `Lovable.dev` to populate each synthesized web application with sample data such as placeholder accounts and content. This data is not intended to represent actual individuals or organizations, and any resemblance is coincidental. Despite these precautions, the dataset may reflect latent social biases or stereotypes inherent in generative models, including those related to gender, race, age, and socioeconomic status. We therefore advise users to exercise caution when interpreting results or building upon this content.

References

Anthropic. 2025a. Introducing claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5.

Anthropic. 2025b. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5.

Anthropic. 2026. Claude code - ai coding agent for terminal & ide. https://claude.com/product/claude-code.

Giovana Giardini Borges and Rogéria Cristiane Gratão de Souza. 2024. Skills development for software engineers: Systematic literature review. *Information and Software Technology*, 168:107395.

Sacha Brisset, Romain Rouvoy, Lionel Seinturier, and Renaud Pawlak. 2022. Erratum: Leveraging flexible tree matching to repair broken locators in web automation scripts. *Information and Software Technology*, 144:106754.

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):206–228.

Common Crawl. 2026. Common crawl - open repository of web crawl data. https://commoncrawl.org/.

Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Lacoste, Massimo Caccia, Alexandre Drouin, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Graham Neubig, Quentin Cappart, Russ Salakhutdinov, and Nicolas Chapados. 2025. The browsergym ecosystem for web agent research. *Transactions on Machine Learning Research*. Expert Certification.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations (ICLR)*.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, and 1 others. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*.

Yifei Gao, Jiang Wu, Xiaoyi Chen, Yifan Yang, Zhe Cui, Tianyi Ma, Jiaming Zhang, and Jitao Sang. 2026. Guitester: Enabling gui agents for exploratory defect discovery. *arXiv preprint arXiv:2601.04500*.

Yongxiang Hu, Yu Zhang, Xuan Wang, Yingjie Liu, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2025. Kuitest: Leveraging knowledge in the

wild as gui testing oracle for mobile apps. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

Ailin Huang, Ang Li, Aobo Kong, Bin Wang, Binxing Jiao, Bo Dong, Bojun Wang, Boyu Chen, Brian Li, Buyun Ma, and 1 others. 2026. Step 3.5 flash: Open frontier-level intelligence with 11b active parameters. *arXiv preprint arXiv:2602.10604*.

ISO/IEC. 2023. ISO/IEC 25010:2023 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. https://www.iso.org/standard/78176.html.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply reinforcing android gui testing with deep reinforcement learning. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*.

Nguyen-Khang Le, Quan Minh Bui, Minh Ngoc Nguyen, Hiep Nguyen, Trung Vo, Son T Luu, Shoshin Nomura, and Minh Le Nguyen. 2025. Automated web application testing: End-to-end test case generation with large language models and screen transition graphs. *arXiv preprint arXiv:2506.02529*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Jenny T Liang, Thomas Zimmermann, and Denae Ford. 2022. Understanding skills for oss communities on github. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 170–182.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. 2025a. Showui: One vision-language-action model for gui visual agent. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR)*.

Zhiyu Lin, Zhengda Zhou, Zhiyuan Zhao, Tianrui Wan, Yilun Ma, Junyu Gao, and Xuelong Li. 2025b. Webuibench: A comprehensive benchmark for evaluating multimodal large language models in webui-to-code. *arXiv preprint arXiv:2506.07818*.

Chenxu Liu, Yingjie Fu, Wei Yang, Ying Zhang, and Tao Xie. 2026. Webcoderbench: Benchmarking web application generation with comprehensive and interpretable evaluation metrics. *arXiv preprint arXiv:2601.02430*.

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 International Conference on Software Engineering (ICSE)*.

Yuxuan Lu, Bingsheng Yao, Hansu Gu, Jing Huang, Zheshen Jessie Wang, Yang Li, Jiri Gesi, Qi He, Toby Jia-Jun Li, and Dakuo Wang. 2025a. 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*.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025b. Webgenbench: Evaluating llms on generating interactive and functional websites from scratch. *arXiv preprint arXiv:2505.03733*.

MiniMax. 2025. Minimax m2.1: Significantly enhanced multi-language programming, built for real-world complex tasks. https://www.minimax.io/news/minimax-m21.

Atsuyuki Miyai, Zaiying Zhao, Kazuki Egashira, Atsuki Sato, Tatsumi Sunada, Shota Onohara, Hiromasa Yamanishi, Mashiro Toyooka, Kunato Nishina, Ryoma Maeda, Kiyoharu Aizawa, and Toshihiko Yamasaki. 2025. Webchorearena: Evaluating web browsing agents on realistic tedious web tasks. *arXiv preprint arXiv:2506.01952*.

Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Juan A Rodriguez, Montek Kalsi, Nicolas Chapados, M Tamer Özsu, Aishwarya Agrawal, David Vazquez, Christopher Pal, and 1 others. 2025. Ui-vision: A desktop-centric gui benchmark for visual perception and interaction. In *International Conference on Machine Learning (ICML)*.

Openai. 2025a. Gpt-5.1: A smarter, more conversational chatgpt. https://openai.com/index/gpt-5-1/.

Openai. 2025b. Introducing gpt-5.2. https://openai.com/index/introducing-gpt-5-2/.

Openai. 2026. Codex | openai. https://openai.com/codex.

Qwen Team. 2026a. Qwen3-coder-next technical report. Technical report. Accessed: 2026-02-03.

Qwen Team. 2026b. Qwen3.5: Towards native multimodal agents.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3956–3974, Albuquerque, New Mexico.

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. *arXiv preprint arXiv:2602.11724*.

Yuxuan Wan, Tingshuo Liang, Jiakai Xu, Jingyu Xiao, Yintong Huo, and Michael R Lyu. 2025. Automatically generating web applications from requirements via multi-agent test-driven development. *arXiv preprint arXiv:2509.25297*.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, and 1 others. 2025. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*.

Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey Bigham, and Jeffrey Nichols. 2024. UICoder: Finetuning large language models to generate user interface code through automated feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7511–7525, Mexico City, Mexico.

Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, Bowen Shen, Chen Zhang, Chenhong He, Chiheng Lou, Fuli Luo, Gang Wang, and 1 others. 2026. Mimo-v2-flash technical report. *arXiv preprint arXiv:2601.02780*.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. 2024. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. *arXiv preprint arXiv:2411.03292*.

Jingyu Xiao, Ming Wang, Man Ho Lam, Yuxuan Wan, Junliang Liu, Yintong Huo, and Michael R Lyu. 2025. Designbench: A comprehensive benchmark for mllm-based front-end code generation. *arXiv preprint arXiv:2506.06251*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering.

Naimeng Ye, Xiao Yu, Ruize Xu, Tianyi Peng, and Zhou Yu. 2025. Ai agents for web testing: A case study in the wild. *arXiv preprint arXiv:2509.05197*.

Sukmin Yun, Rusiru Thushara, Mohammad Bhat, Yongxin Wang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, and 1 others. 2024. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Yafen Yao, Wei Li, Xiaolin Chen, Yongshun Gong, Bei Guan, and 1 others. 2024. Codes: Natural language to code repository via multi-layer sketch. *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chengxing Xie, Cunxiang Wang, and 1 others. 2026. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Changzhi Zhou, Ken Deng, Dengpeng Wu, Guanhua Huang, Kejiao Li, and 1 others. 2025. Artifactsbench: Bridging the visual-interactive gap in llm code generation evaluation. *arXiv preprint arXiv:2507.04952*.

Kangjia Zhao, Jiahui Song, Leigang Sha, Haozhan Shen, Zhi Chen, Tiancheng Zhao, Xiubo Liang, and Jianwei Yin. 2024a. Gui testing arena: A unified benchmark for advancing autonomous gui testing agent. *arXiv preprint arXiv:2412.18426*.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. 2024b. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*.

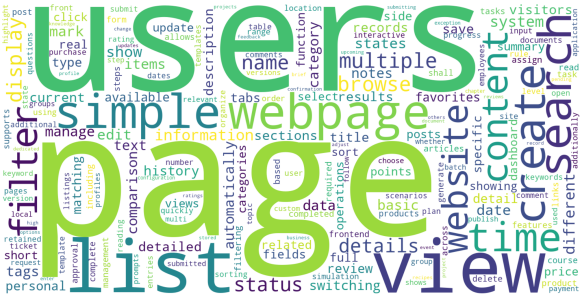
Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations (ICLR)*.

A WebTestBench

A.1 Dataset Statistics

Table 6 presents the statistics of WebTestBench. The benchmark comprises 100 carefully curated development instructions spanning seven categories, each paired with an AI-generated web application that exhibits naturally occurring defects. The applications are structurally complex and highly interactive, reflecting the characteristics of real-world web development. On average, each instance is associated with 17.5 gold test items distributed across four dimensions: functionality, constraint, interaction, and content. The relevant descriptions and typical examples for the seven web application categories and four test case types are provided in Tables 12 and 13, respectively.

Figures 5 and 6 present word cloud visualizations for development instructions and gold test checklists, respectively, illustrating the lexical characteristics of WebTestBench.


users
page
simple
webpage
list
view
search
content
create
time
website
filter
display
information
records
multiple
browse
name
status
data
details
comparison
history
text
manage
edit
sections
title
tabs
select
results
favorites
current
available
description
category
function
interactive
states
notes
save
system
visitors
update
show
items
click
mark
real
type
tags
personal
short
detailed
fields
related
review
switching
simulation
automatically
categories
points
basic
date
showing
different
price
course
payment
product
delete
shows
frontend
completed
sorting
filtering
template
approval
complete
prompts
entries
submitted
side
post
form
change
rating
purchase
profile
steps
questions
highlight
supports
version
pages
local
retained
ticket
request
matching
listings
profiles
keyword
comment
dashboard
assign
documents
summary
input
read
task
tasks
location
projects
submitting
interaction
implement
specific
site
posts
articles
scenarios
products
plan
frontend
completed
across
priority
links
keywords
comment
features
publish
time
date
detail
showing
basic
points
choose
sort
sorted
required
function
interactive
states
notes
save
progress
summary
documents
assign

cases. The annotation document provides descriptions, evaluation focus, and typical examples for each dimension. Detailed definitions are provided in Table 13.

Among the four dimensions, *Constraint* represents implicit logical rules that are rarely explicitly stated in development instructions yet critical to application correctness. To provide concrete guidance for identifying constraint violations, Table 14 presents a taxonomy of common constraint types encountered in web applications, along with guiding questions that help identify potential violations.

Annotation Procedure. Human annotators participate in the entire annotation pipeline. During instruction collection, annotators review and refine the development instructions rewritten by `GPT-5.1`, ensuring that the descriptions focus on application implementation and avoid references to real individuals, companies, or brands. They also verify and correct the category label assigned to each instruction. During web application synthesis, annotators use `Lovable.dev` to generate a web application from each instruction. During gold checklist and result annotation, annotators construct the required test cases by combining direct information from the instruction with additional reasoning about potential behaviors. Each test case is then assigned to one of the four dimensions. Annotators interact with the synthesized application to align test cases with the corresponding interface components and interaction flows, and perform defect detection for each item. This process produces a tuple consisting of the test case description, a verdict (Pass or Fail), and an optional bug report for failed items. If an instance contains fewer than three defects, annotators either continue interacting with `Lovable.dev` to introduce additional features or modify the instruction directly. The checklist and results are updated accordingly until the instance contains a sufficient number of defects, ensuring that each example is informative for defect detection evaluation. Finally, all annotation results are cross-validated among annotators.

Expected Output. Following this pipeline, each instance in the WebTestBench includes a development instruction, an web application containing naturally occurring defects, a manually written checklist, a Pass/Fail verdict for each test item, and an optional bug report for failed items.

A.2.2 Human Annotators

We employ crowdsourcing for data annotation. All annotators have prior experience evaluating the quality of AI-generated web applications in similar annotation tasks. Before the main annotation stage, candidates are required to study the annotation document and complete a trial annotation on three example samples. This stage serve both as training and as a qualification assessment. Only candidates who demonstrate high consistency, accuracy, and a clear understanding of the guidelines are selected for the main annotation task. A senior annotation lead, familiar with the task requirements and expected data quality, review all annotations and resolved disagreements identified during cross-validation.

A.2.3 Reproducibility

The code for each web application is available and can be deployed locally with a minimal setup command (`npm install && npm run dev`) and does not rely on `Lovable.dev` during inference. This design ensures that the benchmark remains fully accessible and that experimental results can be reproduced independently of any future changes to the `Lovable.dev` platform.

B More Experimental Details and Results

B.1 More Experimental Settings

In this work, we employ Claude Code v2.1.25 and the Claude Agent SDK v0.1.0, with the maximum iteration turns limited to 150. To enable Claude Code to support non-Claude models, we integrate OpenRouter[4] with Claude Code as a unified invocation interface. For the checklist generation agent, Playwright MCP is not employed, as the target task involves test case generation rather than web interaction. For the defect detection agent, we use Playwright MCP v0.0.41 with the viewport fixed at 1280×720 during evaluation. The prompt templates for the two agents are shown in Figures 8 and 9. When scoring, we set the temperature to 0.1 to ensure reproducibility. All other configurations follow the default settings.

While various LLMs can be deployed within Claude Code, the high complexity of the task presents substantial practical challenges. Models that fail to complete the testing process, such as inability to utilize Playwright MCP, output format

[4] https://openrouter.ai/

Model	Overall		Functionality		Constraint		Interaction		Content	
	Coverage	F1	Coverage	F1	Coverage	F1	Coverage	F1	Coverage	F1
Step-3.5-Flash	66.2 ±0.3	23.6 ±1.1	79.9 ±0.3	20.9 ±1.6	53.8 ±0.7	27.6 ±0.4	49.8 ±1.4	21.5 ±1.0	60.4 ±0.8	2.6 ±0.0
MiMo-V2-Flash	63.2 ±1.1	24.1 ±1.0	79.1 ±1.9	21.5 ±0.4	48.2 ±0.7	28.9 ±0.7	48.5 ±0.9	20.6 ±2.0	49.6 ±1.0	8.2 ±0.8
Claude Sonnet 4.5	64.0 ±0.4	21.8 ±0.2	80.6 ±0.6	20.3 ±3.3	48.5 ±2.1	21.3 ±0.7	42.9 ±6.5	18.5 ±1.1	52.2 ±2.9	2.4 ±1.6
GPT-5.1	63.0 ±0.3	26.0 ±0.8	75.6 ±0.9	31.5 ±1.2	49.9 ±1.4	25.2 ±2.3	48.2 ±1.3	19.7 ±3.7	59.1 ±1.4	16.8 ±2.8

Table 7: Variance of repeated runs in the end-to-end setting. Scores are reported as Mean ± Standard Deviation (%) across three independent runs. The small standard deviations indicate stable performance across trials, demonstrating the reliability of the evaluation framework.

Model	Functionality				Constraint				Interaction				Content			
	Cov.	P	R	F1	Cov.	P	R	F1	Cov.	P	R	F1	Cov.	P	R	F1
Minimax M2.1	76.7	16.0	14.4	12.6	41.4	18.8	14.2	15.1	43.5	21.3	17.6	18.2	45.6	8.3	7.3	7.5
Qwen3 Coder Next	77.5	17.7	15.2	14.4	47.8	27.7	23.1	23.8	42.2	13.1	14.1	12.3	37.0	4.3	3.9	4.0
GLM 4.7	79.7	21.1	17.3	16.7	48.1	27.2	18.9	21.0	38.1	20.7	19.6	18.9	47.7	5.5	4.7	4.9
GLM 5	79.8	17.4	11.2	12.2	48.9	32.3	27.2	27.4	39.1	23.0	19.6	20.1	50.2	4.3	2.2	2.8
Step 3.5 Flash	79.9	**29.1**	19.4	20.9	**53.8**	32.1	27.5	27.6	**49.8**	24.5	21.6	21.5	**60.4**	2.6	2.6	2.6
MiMo-V2-Flash	79.1	24.9	24.0	21.5	48.2	**34.2**	**28.2**	**28.9**	48.5	23.0	19.8	20.6	49.6	9.1	9.6	8.2
Claude Opus 4.5	**82.4**	22.1	16.9	17.6	49.9	26.3	17.5	20.4	41.4	19.6	14.6	15.7	42.4	6.0	5.1	5.4
Claude Sonnet 4.5	80.6	24.4	20.9	20.3	48.5	27.4	19.0	21.3	42.9	21.0	18.4	18.5	52.2	3.4	2.0	2.4
GPT 5.2	77.2	25.6	32.7	25.5	52.0	25.1	23.7	22.6	43.4	**29.4**	**22.5**	**24.5**	45.7	6.0	4.1	4.5
GPT 5.1	75.6	**29.1**	**43.0**	**31.5**	49.9	28.7	25.1	25.2	48.2	20.9	21.5	19.7	59.1	**16.4**	**18.8**	**16.8**

Table 8: Detailed quality dimension-wise evaluation results on WebTestBench.

errors, or unexpected termination, will be excluded from our evaluation.

The matching between predicted and gold test items described in Section 3.3 is performed by `Qwen3.5-27B` as a semantic judge. The prompt template used for this step is provided in Figure 10.

B.2 Variance across Repeated Runs

To assess the stability of our evaluation framework, we conduct three independent runs for four representative models on WebTestBench. Table 7 reports the mean and standard deviation across these runs. The results indicate that the variance is generally small, with standard deviations typically below 1.5% for Coverage and below 2.0% for F1. This demonstrates the stability and robustness of both the evaluation framework and the models' execution trajectories, ensuring that our reported performance metrics are reliable and reproducible.

B.3 Correlation with Human Judgments

As described in Section 3.3, our evaluation protocol incorporates an LLM only for the relatively simple task of matching predicted test items to gold test items, rather than asking the LLM to directly assign scores. This design choice makes the evaluation more reliable compared to direct LLM-based scoring approaches. To further validate the alignment between our automated evaluation protocol and human judgment, we conduct a correlation analysis on results aggregated from three models, including GLM-4.7, Step-3.5-Flash, and Claude Sonnet 4.5. Given the high cost of manual annotation, we randomly sample a subset of 20 instances with a balanced category distribution, comprising 352 gold test items and 1,144 predicted test items in total across the three models (GLM-4.7: 379, Step-3.5-Flash: 398, Claude Sonnet 4.5: 367). The per-category sample counts are 2/3/3/3/3/3/3 across the seven categories. Human annotators are asked to perform the item matching task manually, and the resulting evaluation metrics are compared against those produced by our automated pipeline. We report Kendall's τ, Spearman's ρ, and Pearson r measures. The results demonstrate that our automatic evaluation achieves strong alignment with human judgments across all correlation metrics.

B.4 Detailed Dimension-wise Analysis

As shown in Table 8, we present the detailed dimension-wise results and provide the following analysis.

Functionality. Functional test items are typically explicit and directly extracted from the development instruction, which explains why *Functionality* consistently achieves the highest coverage across all models. However, high coverage does not translate to high F1 score, reflecting persistent difficul-

Model	Presentation				Search				Tool				Commerce			
	Cov.	P	R	F1	Cov.	P	R	F1	Cov.	P	R	F1	Cov.	P	R	F1
Minimax-M2.1	58.6	25.6	10.5	14.5	53.1	18.9	6.9	9.4	59.7	15.2	14.4	13.7	56.8	10.5	9.2	8.0
Qwen3-Coder-Next	59.6	17.3	16.7	14.8	63.3	21.5	11.1	13.7	57.5	34.3	12.9	17.0	65.4	28.1	17.4	19.9
GLM-4.7	58.4	23.4	13.3	16.5	63.5	19.4	26.5	20.0	62.8	18.2	9.2	10.4	64.5	36.8	20.1	25.1
GLM-5	62.9	30.3	16.7	20.9	62.2	14.8	5.8	8.3	**63.9**	26.2	11.9	14.6	65.8	38.7	15.2	19.7
Step-3.5-Flash	**65.8**	39.5	14.2	19.9	**69.6**	**25.9**	14.9	18.6	61.0	37.6	22.7	25.9	**72.4**	35.2	22.5	**25.7**
MiMo-V2-Flash	60.1	24.8	15.1	17.4	59.5	24.8	**31.5**	**24.1**	59.7	**44.2**	27.8	30.6	64.3	34.2	17.1	19.9
Claude Opus 4.5	63.6	37.3	22.9	**28.0**	60.5	11.1	5.4	6.8	61.6	28.4	16.7	17.4	64.0	**54.2**	16.2	23.9
Claude Sonnet 4.5	62.8	**44.0**	19.1	24.1	65.9	21.1	20.9	19.7	62.0	32.6	20.4	23.2	63.7	22.0	12.5	13.5
GPT-5.2	57.9	34.7	21.7	25.3	57.9	15.5	14.2	14.4	61.2	29.9	**35.0**	**30.9**	60.3	22.7	18.4	18.0
GPT-5.1	57.7	27.2	**25.6**	24.8	63.4	19.8	29.1	20.8	59.5	25.1	33.5	26.6	69.8	24.9	**26.6**	24.3
Avg.	60.7	30.4	17.6	20.6	61.9	19.3	16.6	15.6	60.9	29.2	20.4	21.0	64.7	30.7	17.5	19.8
Std.	2.8	8.3	4.7	4.8	4.5	4.5	9.8	5.9	1.8	8.7	9.1	7.4	4.3	11.9	4.9	5.6

Model	Data Management				Workflow				User-Generated Content				Overall			
	Cov.	P	R	F1	Cov.	P	R	F1	Cov.	P	R	F1	Cov.	P	R	F1
Minimax-M2.1	63.0	34.5	22.1	23.7	62.8	16.7	10.4	11.0	60.2	27.6	21.3	18.3	59.8	21.6	14.2	14.7
Qwen3-Coder-Next	59.7	38.8	18.8	21.6	58.8	18.4	16.5	15.3	59.9	**30.1**	16.8	17.8	60.2	27.8	16.0	17.4
GLM-4.7	61.3	43.1	21.8	25.1	61.1	22.0	16.2	16.3	59.7	24.5	16.5	17.5	61.5	27.5	17.2	18.6
GLM-5	61.5	35.5	19.7	23.2	65.6	37.7	23.9	26.7	57.6	23.6	8.8	11.4	62.9	30.7	15.6	18.8
Step-3.5-Flash	**67.7**	**46.3**	33.5	**34.5**	65.2	27.1	18.4	18.3	64.6	25.8	13.3	15.6	**66.2**	**34.9**	21.1	23.6
MiMo-V2-Flash	66.6	40.8	29.4	30.5	65.3	**39.5**	24.2	27.5	**64.8**	16.0	17.5	12.0	63.2	33.8	23.5	24.1
Claude Opus 4.5	61.7	27.4	11.3	15.2	65.7	39.0	21.9	26.1	62.3	20.8	10.7	13.1	62.9	32.1	15.6	19.2
Claude Sonnet 4.5	64.6	44.0	23.1	26.8	**67.6**	33.5	22.4	24.8	61.3	23.0	13.3	15.5	64.0	32.8	19.3	21.8
GPT-5.2	62.8	29.2	25.5	24.9	64.6	23.3	29.7	24.3	62.0	18.2	28.3	19.4	61.4	25.6	25.7	23.4
GPT-5.1	65.4	29.9	**37.7**	30.0	64.4	28.4	**39.8**	**30.0**	60.7	16.9	**28.8**	**20.4**	63.0	25.4	**32.5**	**26.0**
Avg.	63.4	37.0	24.3	25.5	64.1	28.6	22.3	22.0	61.3	22.7	17.5	16.1	62.5	29.2	20.1	20.8
Std.	2.5	6.7	7.6	5.4	2.5	8.5	8.1	6.3	2.2	4.7	6.8	3.1	1.9	4.3	5.8	3.6

Table 9: Detailed category-wise evaluation results on WebTestBench.

ties in executing multi-step interaction flows and reasoning over execution outcomes.

Constraint. Despite substantially lower coverage than Functionality, Constraint achieves relatively higher F1 score. This is because constraint failures typically manifest as clear, observable state changes (e.g., a double-booking being accepted when it should be rejected), making defect detection more straightforward when the relevant test case is present. The primary bottleneck lies in generating constraint cases, as they require reasoning about implicit logic constraint not explicitly stated in the instruction.

Interaction. Interaction items focus on dynamic UI feedback (e.g., toast notifications, button state changes). Coverage is similarly low due to their implicit nature, and detection is non-trivial as it requires observing transient state changes.

Content. Content performs worst by a marked margin, with two primary failure modes: (1) models often fail to generate items targeting the actual defects, resulting in zero true positives and consequently zero F1 on many samples despite non-zero coverage, and (2) models struggle to judge whether the displayed content (text, images, icons) is semantically aligned with the theme of the development instruction. These findings indicate that the primary bottleneck in checklist generation lies in the insufficient surfacing of implicit requirements, while content-level semantic alignment remains a formidable challenge.

B.5 Category-wise Analysis

As shown in Table 9, we present detailed category-wise results, along with the cross-model mean and standard deviation for each category and metric, to reflect both the average performance and the variability across models. We further analyze these results in conjunction with the category definitions provided in Table 12.

Data Management achieves the highest overall performance, with an average F1 score of 25.5%. Applications in this category involve explicit state transitions, such as record creation, deletion, and numerical updates, where Pass/Fail judgments can be directly inferred from observable changes in the interface state before and after an operation. *Workflow* applications share a similar state-transition structure but present additional challenges due to

GPT-5.1	
Avg. Playwright Calls: 23.23	
Tool	**Calls**
browser_click	14.69
browser_fill_form	2.14
browser_snapshot	1.81
browser_wait_for	1.41
browser_navigate	1.23
browser_evaluate	0.68
browser_type	0.55
browser_press_key	0.20

GPT-5.2	
Avg. Playwright Calls: 61.96	
Tool	**Calls**
browser_click	35.71
browser_snapshot	6.47
browser_evaluate	5.86
browser_fill_form	4.72
browser_navigate	2.13
browser_wait_for	2.08
browser_type	1.56
browser_press_key	1.36

Claude Sonnet 4.5	
Avg. Playwright Calls: 34.74	
Tool	**Calls**
browser_click	18.74
browser_snapshot	4.73
browser_type	3.39
browser_evaluate	2.24
browser_navigate	2.02
browser_fill_form	1.48
browser_wait_for	1.13
browser_press_key	0.58

MiMo-V2-Flash	
Avg. Playwright Calls: 49.37	
Tool	**Calls**
browser_click	25.23
browser_evaluate	8.29
browser_navigate	4.00
browser_snapshot	3.57
browser_fill_form	2.96
browser_type	2.47
browser_wait_for	0.86
browser_console_messages	0.46

Step-3.5-Flash	
Avg. Playwright Calls: 48.83	
Tool	**Calls**
browser_click	26.01
browser_snapshot	6.98
browser_type	4.89
browser_evaluate	3.30
browser_navigate	2.59
browser_wait_for	1.78
browser_fill_form	1.47
browser_press_key	0.73

GLM-5	
Avg. Playwright Calls: 37.73	
Tool	**Calls**
browser_click	23.84
browser_snapshot	4.33
browser_type	3.90
browser_navigate	1.74
browser_fill_form	1.63
browser_evaluate	0.69
browser_press_key	0.62
browser_wait_for	0.37

Table 10: Statistics of Playwright tool calls per instance for representative models.

multi-step processes and the need for cross-stage context tracking.

Commerce and *Tool* applications occupy the middle tier, with average F1 scores of 19.8% and 21.0%, respectively. Commerce applications involve structured transaction flows that can generally be verified, though complex business logic, such as inventory constraints and payment validation, introduces additional reasoning requirements. Tool applications follow an input-processing-output paradigm, where correctness verification requires models to infer expected outputs based on specific inputs. Notably, *Tool* shows the largest cross-model variance in F1 (7.4%), suggesting that the reasoning demands in this category are unevenly handled across models.

Presentation performance is dominated by closed-source models, with Claude Opus 4.5 achieving the highest F1 score of 28.0%, while open-source models generally perform much worse. This gap suggests that evaluating content layout and semantic structure benefits disproportionately from stronger model capacity. *Search* shows similar large cross-model variance (5.9%), with performance ranging from 6.8% for Claude Opus 4.5 to 24.1% for MiMo-V2-Flash. Both categories require assessing semantic alignment between rendered content and user intent, rather than simply observing discrete component states, which presents a verification challenge that no model currently handles consistently.

User-Generated Content has the worst performance, with an average F1 score of only 16.1% and the smallest cross-model variance (3.1%), indicating that all models struggle equally in this domain. UGC applications involve unstructured, dynamically generated content, whose correctness cannot be determined from fixed component states, necessitating abstract semantic judgment that consistently exceeds current CUA capabilities.

Together, these disparities emphasize the fundamental distinction between operation-driven verification, where correctness is grounded in observable state transitions, and understanding-driven verification, where correctness relies on semantic alignment with user intent. Overcoming this gap presents a central challenge for the future development of automated web testing.

B.6 Analysis of Tool Usage

To better understand the interaction workflows of CUAs during automated web testing, we analyze the playwright tool usage of representative models, as shown in Table 10.

We observe substantial variation in interaction volume across models. GPT-5.2 produces the highest number of tool calls (61.96 per instance), indicating a preference for aggressive exploration

Model	Functionality			Constraint			Interaction			Content			Overall		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
MiMo-V2-Flash	22.2	20.1	19.3	50.5	49.7	46.6	33.3	31.1	30.3	16.2	16.7	15.8	44.3	30.4	31.9
Step-3.5-Flash	33.5	25.6	26.4	61.4	56.9	55.6	39.2	36.9	35.9	30.3	33.3	31.2	53.4	38.7	40.9
GPT-5.1	42.8	54.8	42.0	57.0	76.5	59.2	55.9	65.7	56.0	36.5	46.2	38.2	47.0	63.4	46.8
Claude Sonnet 4.5	45.0	32.3	34.5	69.1	72.6	66.7	44.8	42.8	41.4	32.3	32.7	30.7	61.0	47.4	49.2

Table 11: Detailed results on WebTestBench in the oracle setting.

through frequent web interactions. In contrast, GPT-5.1 uses fewer calls (23.23 per instance), suggesting a more efficiency-oriented strategy, likely supported by stronger planning capabilities.

Across all models, `browser_click` is the most frequently used tool, accounting for approximately 50% to 63% of total calls. This is consistent with real-world web interaction patterns, where click-based actions constitute the most fundamental and pervasive form of engagement. In addition, notable differences emerge in how models leverage auxiliary tools. MiMo-V2-Flash and GPT-5.2 rely more heavily on `browser_evaluate` (8.29 and 5.86 per instance respectively), indicating a tendency to inspect underlying page structures such as the DOM structure or JavaScript execution results. While this approach can enable more precise state verification, it also incurs considerable computational overhead, as reflected in the significantly higher context token consumption of both models (7.26M and 7.43M respectively). Notably, these models exhibit relatively low usage of `browser_wait_for`, suggesting that asynchronous behaviors are not adequately handled. This may cause models to misinterpret transient UI rendering delays and asynchronous state updates as genuine defects, contributing to elevated false-positive rates.

These findings highlight that future CUA systems would benefit from more deliberate interaction planning and more selective use of tools, both of which are important for improving testing reliability in dynamic web environments.

In addition, Figure 7 illustrates the web interaction process of CUAs during defect detection. We present the invoked Playwright tools, the corresponding actions, and the resulting webpage screenshots after each interaction. This visualization is based on the output of GPT-5.1.

B.7 Detailed Results and Analysis in the Oracle Setting

Table 11 reports detailed results under the oracle setting, where the gold checklist is directly provided to the defect detection agent. Compared to the end-to-end setting, all models show clear improvements, indicating that incomplete checklists are the major bottleneck in end-to-end testing. In this setting, the gap between closed-source and open-source models also becomes more pronounced. Claude Sonnet 4.5 and GPT-5.1 achieve overall F1 scores of 49.2% and 46.8%, respectively, clearly outperforming MiMo-V2-Flash (31.9%) and Step-3.5-Flash (40.9%). This pattern indicates that checklist generation is the primary limiting factor for closed-source models in end-to-end setting, while open-source models still face difficulties in both checklist generation and detection.

Across all models, *Constraint* consistently achieves the highest F1 scores, with Claude Sonnet 4.5 reaching 66.7% and GPT-5.1 reaching 59.2%. This is consistent with the nature of constraint violations, which often appear as explicit and observable state changes that are easy to verify once the relevant test item is given. In contrast, *Functionality* remains difficult for open-source models even in the oracle setting, suggesting limitations in handling multi-step interactions and reasoning over execution processes. Notably, *Content* shows the lowest performance across all models. This indicates that verifying semantic alignment between displayed content and development intent or test query remains challenging.

C More Related Work

Computer-Use Agents. Beyond pure code generation, CUAs have emerged as a tool for interacting with web environments. Through screenshots (Lin et al., 2025a; Nayak et al., 2025; Wang et al., 2025) or accessible code (*e.g.*, DOM trees) (Deng et al., 2023; de Chezelles et al., 2025), CUAs can execute multi-step tasks involving navigation, clicking, and typing. Current benchmarks (Zhou et al., 2024; Xie et al., 2024; Miyai et al., 2025) primarily focus on operational success in realistic web scenarios, such as online shopping and discussion forums. While CUAs have demonstrated potential in general web

activities, their effectiveness in software quality assurance, particularly defect detection, remains insufficiently explored.

Automated Web Testing. Web application testing is a critical yet time-consuming phase in software development, motivating the pursuit of automated solutions. Early automation methods like Selenium use script tools to execute user operations. While simplifying manual tasks, they tightly bind to the website's DOM structure and event triggers, leading to limited scalability and high maintenance costs (Brisset et al., 2022). Training-based methods (Li et al., 2019; Lan et al., 2024) can learn testing strategies, but they suffer from high training costs and limited generalization to emerging web frameworks. Recent works (Liu et al., 2024; Hu et al., 2025) investigate multimodal LLMs for testing mobile applications. For web applications, Le et al. (2025) focus on automating site navigation and form filling. UXAgent (Lu et al., 2025a) introduces personalized agents for testing web applications. In this paper, we focus on the automated web application testing. Beyond basic functionality, we explore agents' performance on long-horizon tasks and their ability to detect latent bugs.

Category	Details
Presentation	*Description:* Presents information in a clear, structured form, , enabling users to browse, read, and access content efficiently. *Core Functionalities:* Content layout and typography, navigation structure, pagination and indexing, media rendering, basic search and filtering. *Typical Examples:* Blog and article websites, personal homepages, corporate websites, documentation and wikis, portfolios, and news portals.
Search	*Description:* Supports search, filtering, ranking, and recommendation over heterogeneous datasets, emphasizing information discovery and matching. *Core Functionalities:* Search input and query syntax, filter and sort controls, result list presentation, recommendation-based content delivery, result detail pages. *Typical Examples:* Job platforms, real estate listing websites, academic literature search engines, content recommendation systems.
Tool	*Description:* Provides a structured input–process–output workflow to help users perform transformation, analysis, or generation tasks, typically emphasizing accuracy and clear feedback. *Core Functionalities:* Input collection (forms), algorithmic or logical processing, result presentation and visualization, and editors for text, graphics, or tables. *Typical Examples:* Data calculation and conversion tools, visualization generators, online document editors, format converters, simulators and configurators.
Commerce	*Description:* Supports complete transaction flows, from product or service selection, often involving business logic related to inventory, pricing, orders, and payment verification. *Core Functionalities:* Product and service browsing, shopping cart and checkout processes, payment and confirmation mechanisms, order management, inventory and pricing management, reservation and scheduling features. *Typical Examples:* E-commerce websites, booking systems for hotels, flights, restaurants, or meeting rooms, subscription platforms, second-hand marketplaces.
Data Management	*Description:* Provides management, monitoring, and operational capabilities over structured data, typically in internal or back-office settings, emphasizing organization, editing, and visualization. *Core Functionalities:* Data tables with CRUD operations, access control mechanisms, dashboards and audit logs, statistical charts, and system configuration. *Typical Examples:* User, order, and inventory management systems, enterprise back-office platforms, visual analytics dashboards.
Workflow	*Description:* Organizes user tasks through predefined steps or stages, emphasizing process control, state management, approval flows, and cross-role collaboration. *Core Functionalities:* Multi-step forms, tracking and state transitions, task and ticket management, approval workflows, content management modules, cross-role collaboration. *Typical Examples:* Content management systems (CMS), project management systems, task and approval systems, complex registration or data collection pipelines.
User-Generated Content	*Description:* Core content is created, edited, shared, and interacted with by users, emphasizing community dynamics, creation toolchains, and social networks. *Core Functionalities:* Content publishing and editors, comments and reactions, user profiles and social graphs, content moderation, notifications and real-time updates. *Typical Examples:* Online forums, commenting systems, social platforms, creative platforms for writing, images, or video content.

Table 12: Taxonomy of web application categories used in WebTestBench.

Dimension	Details
Functionality	*Description:* Focuses on whether the application can successfully complete the primary tasks and business flows specified in the development instruction under normal usage conditions. *Evaluation Focus:* Successful completion of core operations (create, view, edit, delete, submit); correct responses to normal user requests (search, filter, navigate, form submission); coherent cross-module flows and reachable navigation. *Typical Examples:* A user can submit a search query and view the returned results; a user can create a new project and see it reflected in the list.
Constraint	*Description:* Focuses on whether the application is robust under abnormal or boundary usage, including input validation, error handling, and prevention of invalid or conflicting states. *Evaluation Focus:* Effective validation of required fields with clear error messages; blocking of conflicting or illegal operations (e.g., double-booking, editing non-existent records); graceful handling of boundary conditions (empty lists, excessively long inputs, invalid formats); consistency between interface state and underlying data. *Typical Examples:* An already-reserved meeting room cannot be booked again; submitting a form with empty required fields is blocked with a clear prompt.
Interaction	*Description:* Focuses on the dynamic feedback and visual or state changes the application provides in response to user actions, reflecting the experiential quality of functionality rather than its correctness. *Evaluation Focus:* Clear feedback after click, input, or toggle actions (e.g., button state changes, toast notifications, success or failure messages); smooth state transitions such as tab switching, expand/collapse, and toggle; visible and interpretable action outcomes (e.g., redirect or confirmation after submission). *Typical Examples:* A success notification appears after a booking is created; hovering over a button triggers an appropriate visual change.
Content	*Description:* Focuses on whether the information displayed by the application is accurate, complete, and semantically consistent with the development instruction, covering all front-end content elements including text, data, images, and icons. *Evaluation Focus:* Thematic consistency of text, field labels; complete and well-formatted content in lists, detail pages, and cards; media assets (images, icons) rendered correctly and relevant to the theme. *Typical Examples:* All images on an iPhone showcase page are iPhone-related; each blog card includes the title, summary, and date fields required by the instruction.

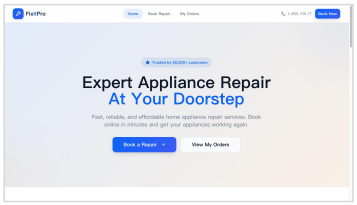
Table 13: Taxonomy of test case dimensions used in WebTestBench.

Constraint Type	Question to Ask
Uniqueness	Can the same entity be created or registered twice?
Conflict / Overlap	Can two operations produce an impossible state? (e.g., double-booking, overlapping assignments)
Required Fields	Can a form be submitted with empty required fields?
Format Validation	Can invalid formats (email, phone, date) pass through?
Temporal Validity	Is a time-related input (date, time range) logically valid? (e.g., end time after start time, no past dates for future bookings)
Numerical Integrity	Do multiple numeric fields satisfy an aggregate constraint? (e.g., weights sum to exactly 100%)
State Prerequisite	Does an action require a prior state? (e.g., cannot checkout without items in cart)
Access Control	Can a user view or modify resources that belong to another user?

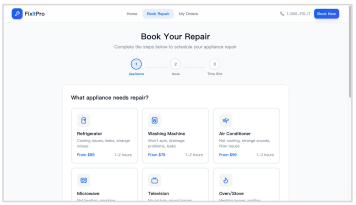
Table 14: Representative constraint types covered in WebTestBench.

Development Instruction:
I want a webpage where users can book home appliance repair services. Users should be able to select the type of appliance that needs repair (like refrigerator, washing machine, air conditioner, or microwave), describe the issue they're experiencing, and choose from available appointment time slots. After selecting a time, they can proceed to a payment page where they enter payment details and confirm the booking. Once the booking is confirmed, users can view their appointments in an order history page and have the option to cancel an appointment if needed.

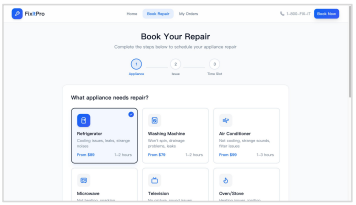
Tool: browser_navigate
Action: Navigate to web_url
1



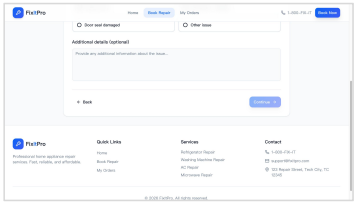
Tool: browser_click
Action: Click “Book Repair” button
2



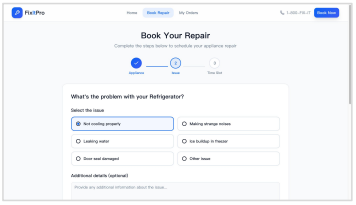
Tool: browser_click
Action: Click “Refrigerator Cooling issues,” button
3



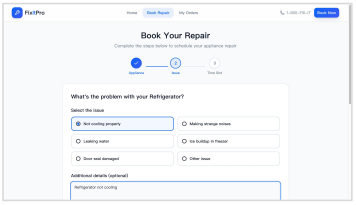
Tool: browser_click
Action: Click “Continue” button
4



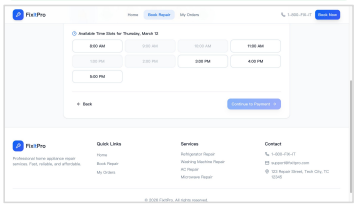
Tool: browser_click
Action: Click “Not cooling properly” button
5



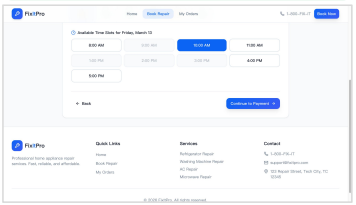
Tool: browser_type
Action: Type “Refrigerator not ...” on “Additional details ...” textbox
6



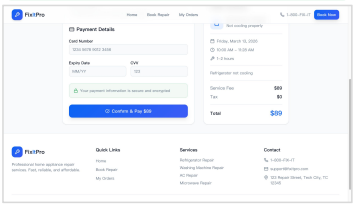
Tool: browser_click
Action: Click “Continue” button
7



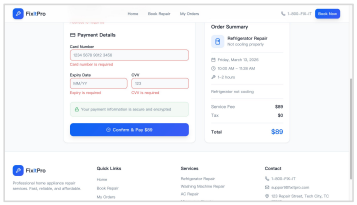
Tool: browser_click
Action: Click “Not cooling properly” button
8



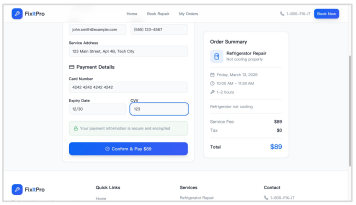
Tool: browser_click
Action: Click “Continue to Payment” button
9



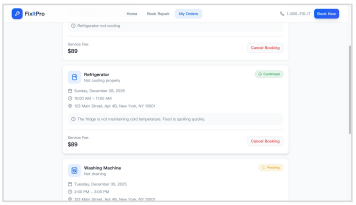
Tool: browser_click
Action: Click “Confirm & Pay $” button
10



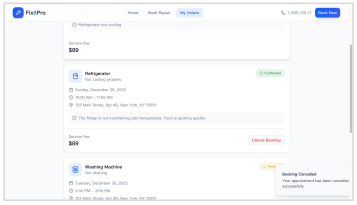
Tool: browser_fill_form
Action: Click “Not cooling properly” button
11



Tool: browser_click
Action: Click “Confirm & Pay $” button
12



Tool: browser_click
Action: Click “Cancel Booking” button
13



Tool: browser_click
Action: Click “Book Repair” button
14

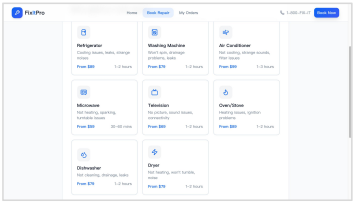


Figure 7: An example illustrating how CUA interacts with a web application during defect detection.

Prompt for Checklist Generation Agent

Role
You are a Senior Software Quality Assurance Engineer who can read a user instruction and immediately produce a complete, executable UI/UX test checklist. Your focus is strictly on 'what' the application should do (the features), not 'how' it should be built (the technical implementation).

Task
Directly generate executable test checklist. Decompose the user instruction into structured, testable items.

Each item must be:
- **Specific**: Clear action and expected outcome.
- **Binary**: PASS or FAIL (no ambiguity).
- **Debuggable**: Failure indicates exactly what's missing.

Your checklist will be used to:
1. Test web applications. Produce PASS/FAIL results for each test item.
2. Generate detailed bug reports identifying which requirement failed and why.

Checklist Item Category
1. Functionality (FT)
   * Focus: Core user tasks and workflows that must succeed when inputs are valid.
   * Scope: What happens when everything goes right?
   * Example: "User can submit a search query", "User can add an item to the cart".
2. Constraint (CS)
   * Focus: Rules, validations, state invariants, and conflict-prevention logic that prevent the system from entering invalid or contradictory states.
   * Scope: What prevents the user from doing the wrong thing? What happens with conflicting data?
   * Examples: "Meeting room cannot be booked if already occupied.", "Cannot submit form with empty required fields."
3. Interaction (IX)
   * Focus: Dynamic behaviors and system responses to user actions (non-functional visual/state changes, user experience).
   * Scope: How does the interface respond to events like clicks, hover?
   * Examples: "Show success toast after reservation is created."
4. Content (CT)
   * Focus: The relevance and integrity of text, data, and media (images, icons, videos). Content must strictly align with the instruction's theme/purpose.
   * Scope: Is the displayed information relevant, and fully functional?
   * Examples: "All displayed images must be directly relevant to the theme of 'iPhone'."

Default Data
Assume the application has default data (e.g., pre-existing products in a store). Do not create new data for testing; use the default data already present in the application.

Unified Checklist Item Template

```markdown
- [ ] [ID]: [Test description]
   - Action: [What to do]
   - Expected: [What should happen]
```

Output Format (Markdown)

```markdown
# Test Checklist

## Functionality
- [ ] FT-01: [use unified template]
- [ ] FT-02: [use unified template]

## Constraint
- [ ] CS-01: [use unified template]

## Interaction
- [ ] IX-01: [use unified template]
```

Content
- [ ] CT-01: [use unified template]
```

# Rules
1. Testable: Every item must produce a clear Pass/Fail result.
2. Executable: Quality assurance tester should know exactly what to do.
3. Specific for action/expected: Include exact element names, button text, expected messages, etc.
4. Concise for description: Test description should be 1-2 lines, action/expected should be brief.
5. No Implementation: Specify what the app does, not how it's built (no framework details).
6. Desktop Only: Ignore responsive design requirements.
7. Max 20 items total: Prioritize core requirements. Keep only what is necessary to satisfy the instruction.
8. No Redundancy: Avoid duplicating content or behavior that is covered by other categories (e.g., "success messages" should be included only once). Each checklist item MUST be assigned exactly one primary category (FT / CS / IX / CT), even if it has secondary implications.

# Input

## User Instruction
$instruction

# Output (Markdown)

Figure 8: Prompt for Checklist Generation Agent.

Prompt for Defect Detection Agent

# Role
You are an expert Quality Assurance Test Engineer specializing in automated UI/UX testing. Your task is to validate a web application against a provided checklist. You must systematically execute actions, verify results, and update the checklist status.

# Execution Standards

## 1. Interaction Strategy
- Tool Use: Use **Playwright tools** to interact with the DOM. Disallow the use of `Bash`, `Read`, and `Write` tools to operate web pages.
- DOM-Only: Do NOT use screenshots or visual validation. Rely on DOM attributes (text, id, class, accessibility roles) for verification.
- Integrity: Execute all items; never skip. If an item cannot be done, mark FAIL with a concrete reason (no hallucination).
- Batching: For pure data entry (e.g., filling a form), you may combine multiple `fill/select` actions into a single code block to save time.
- Limited Budget: The entire execution process must operate within a limited budget of turn/tool-call (max 150 times total). Plan first, and execute with as few operations as possible.
- Navigation: Only navigate if the checklist item explicitly requires it. Disable page refresh operations unless the page crashes.

## 2. Verification Logic
- Strict Verification: Compare the `'Actual` behavior of the page against the `Expected` field in the checklist.
- Pass: The feature works exactly as described.
- Fail: Any deviation (missing element, wrong text, no response, error message) is a FAIL.

## 3. Workflow
1. Initialize: Navigate to the Target URL.
2. Iterate: Go through the Checklist items.
3. Execute: Perform the 'Action' defined in the item.
4. Verify: Check if the 'Expected' result is met.
5. Record: Update the item's status immediately in your internal memory.

# Output Format (Markdown)
You must output the Full Checklist with updated statuses. Do not summarize; return the complete list.

## Unified Result Item Template

If PASS: Change `- [ ]` to `- [X]` to mark the test as passed.
```

```markdown
- [X] TEST-ID: [original Description]
    - Action: [original Action]
    - Expected: [original Expected]
```

If FAIL: Keep `- [ ]` and append a ‘Bug Report‘ block immediately after the test item.

```markdown
- [ ] TEST-ID: [original Description]
    - Action: [original Action]
    - Expected: [original Expected]
    - Bug Report:
        - Issue: [Specific problem type: e.g., Unresponsive Button, Incorrect Form Submission, Element Occlusion]
        - Actual: [Quote the observed deviation: e.g., Button does not trigger the expected modal, Button text overlaps with icon]
```

Output Template

```markdown
# Test Result

## Functionality
[use unified result item template for each FT-xx]
[use unified result item template for each FT-xx]

## Constraint
[use unified result item template for each CS-xx]

## Interaction
[use unified result item template for each IX-xx]

## Content
[use unified result item template for each CT-xx]
```

Input

User Instruction
$instruction

Application URL
$server_url

Test Checklist
```markdown
$checklist
```

Output

Figure 9: Prompt for Defect Detection Agent.

Prompt for Test Item Matching

Given a list of predicted checklist items and a list of gold checklist items. You are required to align predicted (model-generated) test items to gold (human-labeled) test items for the same web instruction.

Instruction:
```
$instruction
```

Gold Test Items (`"gold_id": "description"`):
```
$gold_items
```

Predicted Test Items (`"pred_id": "description"`):
```
$pred_items
```

Goal:
For each predicted item, decide if it corresponds to exactly one gold item describing the same requirement/behavior. Produce a one-to-one mapping; unmatched predictions should map to None.

Matching rules:
1. Mapping constraint: each predicted item maps to AT MOST ONE gold item; each gold item MAY be assigned to MULTIPLE predicted items.
2. Prioritize intent over wording: if a predicted item is more specific/less specific but clearly covers the same user requirement, match it; otherwise, leave it unmatched.
3. Do NOT force matches: if no gold item cleanly aligns, use None.
4. Preserve predicted order: output tuples follow the input predicted sequence; length of output list equals number of predicted items.

Output Format (Markdown)
[("pred_id_1", "gold_id" or None), ("pred_id_2", "gold_id" or None), ...]

DO NOT PROVIDE ANY OTHER OUTPUT TEXT OR EXPLANATION. Only output the List. Output:

Figure 10: Prompt for Test Item Matching.