Framework and Benchmark for Code-Driven Agentic Testing in Web Development

Bin Hong[1], **Zhenchao Zhang**[3], **Jiyuan He**[3], **Kai Zhang**[1], **Zhenya Huang**[1,2,†]
[1] State Key Laboratory of Cognitive Intelligence,
University of Science and Technology of China
[2] Institute of Artificial Intelligence,
Hefei Comprehensive National Science Center
[3] Meituan
hb2002@mail.ustc.edu.cn, huangzhy@ustc.edu.cn

Abstract

End-to-end GUI testing is essential for verifying web applications, yet existing evaluations rely on predefined checklists and are confined to the data and frameworks of web generation benchmarks, leaving the bug-discovery ability of vision-language models (VLMs) systematically untested. We introduce **C**ode-driven **A**gentic **T**esting (CAT), a paradigm in which the agent writes Playwright code to drive the browser, gathers feedback, and autonomously explores web applications to uncover bugs. We instantiate CAT with CATJudge, an agentic framework that unifies Browser-Use and Computer-Use tools within a single environment and CATTest, a benchmark of 102 AI-generated web applications with carefully annotated bugs, built through close human-AI collaboration to feature complex interactions and subtle defects. Experiments with mainstream VLMs show that all evaluated models perform poorly, revealing a clear gap between current VLM capabilities and the demands of real-world testing in AI web development. We release our code and data.[1]

1 Introduction

With the rapid advancement of large language models (LLMs) in code intelligence, LLM-powered coding agents have achieved remarkable progress on repository-level code generation tasks (Jimenez et al., 2024; Jiang et al., 2026). Collaborating with coding agents is increasingly becoming a programming paradigm that tangibly improves productivity (Ge et al., 2025; Jiang et al., 2024). While coding agents excel at conventional development, they still exhibit notable shortcomings in web application generation, which cannot be verified through textual unit tests or static analysis alone. Verifying web applications relies on visual and interactive feedback, as many bugs manifest only when the code is rendered in a browser (Zhou et al., 2025; Visconti et al., 2022). This makes end-to-end (E2E) testing involving graphical user interface (GUI) operations an essential measure for both quality assurance of AI-generated web applications and filtering of training data (Ye et al., 2025; Pezze et al., 2018).

E2E GUI testing incurs substantial human labor costs (Alégroth et al., 2015; Pezze et al., 2018). Benchmarks on web code generation typically employ vision-based rule scoring or LLM-as-Judge scoring (Sun et al., 2025; Wang et al., 2025b; Awal et al., 2025). More recently, benchmarking efforts have focused on using GUI agents based on vision-language models (VLMs) to automatically execute encapsulated GUI operations such as clicks and type to verify web code correctness (Lu et al., 2026; Wu et al., 2025; Bian et al., 2025). However, current LLMs/VLMs have limited GUI capabilities, and their GUI interaction environments typically assume fully functional web pages (Xie et al., 2024; Miyai et al., 2025). For testing scenarios that require identifying bugs based on anomalous feedback and demand more intensive web development knowledge, there is a lack of systematic evaluation of their E2E GUI testing abilities.

This gap manifests in two aspects. First, the current web testing paradigm relies on predefined checklists (GUI test cases described in natural language) (Lu et al., 2026; Wu et al., 2025). Yet AI-generated web pages, even when constrained by prompts, often produce features beyond what the checklist anticipates. The testing pipeline can only reflect the model's ability to execute predefined GUI test cases. More importantly, as code intelligence continues to advance and AI-generated web pages grow in complexity, the volume of checklist items will inevitably expand, leading to higher costs, decreased testing efficiency, and reduced reliability. Second, current evaluations of GUI testing capabilities are merely validations of automated

[†]Corresponding author
[1] https://github.com/SleepyWithoutCoffee/CATJudge

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 47250–47279
October 24-29, 2026 ©2026 Association for Computational Linguistics

evaluation methods within web generation benchmarks, constrained by the data and frameworks of those benchmarks themselves. The types and complexity of interactions required by the web pages are limited, and the subtlety of bugs is also limited.

To bridge this gap, we first model the pipeline from user-provided prompts to model-delivered code in AI web generation scenarios. Based on this, we define a different testing paradigm: the agent directly writes Playwright (Microsoft, 2025) code to interact with the browser, gather textual and visual feedback, and autonomously explore the web application to discover bugs. We term this paradigm as **C**ode-driven **A**gentic **T**esting (CAT). Then, we build CATJudge, an agentic framework that unifies the two mainstream families of GUI agent tools, Browser-Use Agent (BUA) (He et al., 2024) and Computer-Use Agent (CUA) (Xie et al., 2024) tools, within a single environment, allowing both toolsets to be invoked for web exploration. To evaluate the CAT capabilities of VLMs under the CAT paradigm, we introduce CATTest, a benchmark of 102 AI-generated web applications and carefully annotated bugs constructed through iterative, close human-AI collaboration. Task instances in CATTest include complex interactions and subtle bugs, demanding strong Playwright coding ability and long-horizon abilities.

We evaluate various VLMs on the CATTest dataset using CATJudge. Experimental results show that current mainstream VLMs achieve poor performance on the CATTest dataset (R-score below 43), revealing a substantial gap between the capabilities of existing VLMs and the practical requirements of real-world web development testing.

Our contributions are summarized as follows:

- We develop a theoretical workflow model for systematically analyzing the agentic web generation scenario, guiding the construction of related framework and benchmark.
- We propose CATJudge, an agentic framework along with a unified environment for executing CAT tasks for web applications. CATJudge integrates both BUA and CUA tools, and executes actions in real browsers.
- We construct CATTest, an agentic benchmark of 102 complex web applications for evaluating the CAT capabilities of VLMs in web development.
- We conduct experiments on CATTest with multiple mainstream VLMs using CATJudge, providing insights into improving their CAT capabilities in web development.

2 Related Works

Code Generation Benchmarks As the coding capabilities of LLMs have (Ye et al., 2026; Jiang et al., 2025; Zhao et al., 2024), research efforts in code intelligence have gradually shifted from function-level generation (e.g., HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2025)) toward file-level and repository-level tasks that align with real-world needs (e.g., SWE-Bench (Jimenez et al., 2024) and Terminal-Bench (Merrill et al., 2026)). These benchmarks generally rely on pre-written static test scripts and test cases to verify the generated code. Web development is a common repository-level code generation scenario in real-world software engineering. Numerous benchmarks have been proposed to evaluate LLMs/VLMs on generating web code. In terms of task formulation, these benchmarks fall into two categories: (1) generating web code from visual information (e.g., images (Sun et al., 2025; Si et al., 2025; Awal et al., 2025) and videos (Chen et al., 2025)), and (2) generating web code from textual prompts (Lu et al., 2026; Zhang et al., 2025; Sun et al., 2025; Wu et al., 2025). To verify AI-generated web code, these benchmarks employ a variety of approaches, including rule-based validation, LLM-as-Judge scoring, and using GUI agents to simulate human interactions.

GUI Agents and Why GUI Agents GUI agents are a class of agents equipped with specialized interaction tools that receive visual information or accessibility information (e.g., accessibility tree, a11y tree) and interact with real environments through these tools (Nguyen et al., 2025; Hu et al., 2025). They are primarily divided into two categories: Browser-Use Agents (BUA) (Browser Use, 2025; He et al., 2024) and Computer-Use Agents (CUA) (Xie et al., 2024; Wang et al., 2025a). The former is equipped with tools built on browser automation frameworks (e.g., Playwright (Microsoft, 2025) and Selenium (SeleniumHQ, 2025)), enabling LLMs to interact with browsers. The latter is equipped with system-level keyboard and mouse automation tools such as PyAutoGUI (Sweigart, 2025), enabling LLMs to localize targets on the screen and interact with computers using mouse and keyboard input just as humans do. For scenarios such as web development involving visual understanding and physical interaction, end-to-end GUI testing is a critical method for finding bugs that appear correct at the textual code level yet

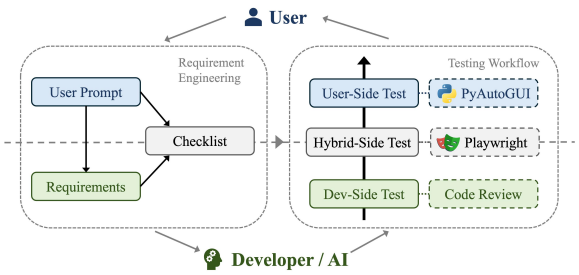


Figure 1: Abstracted workflow of AI-driven web development. Our evaluation focuses on the testing workflow.

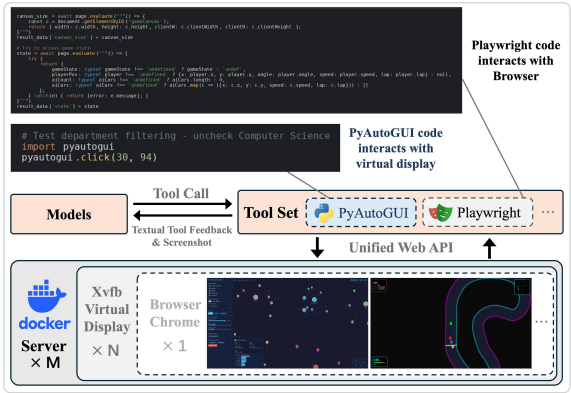


Figure 2: Overview of the CATJudge framework (set-up and tear-down procedures omitted).

manifest as errors when deployed in a browser. Rule-based validation and LLM-as-Judge scoring or comparison struggle to reflect the interactive functionality of web pages after actual deployment. Consequently, recent web generation benchmarks have increasingly adopted GUI agents to automate interaction with web pages for testing (Lu et al., 2026; Chen et al., 2025; Wu et al., 2025).

3 CAT in Web Development

In this section, we introduce **C**ode-Driven **A**gentic **T**esting in Web Development in detail. We begin with our research methodology, in which we model the workflow of AI web generation and define the CAT paradigm. Building on this, we design the CATJudge framework. Finally, we construct the CATTest benchmark for evaluation.

3.1 Methodology

Analogous to the human software development process, the AI-driven web development workflow can be abstracted as a closed loop: user → requirements → AI → testing → delivery, as shown in Figure 1. In this model, the user provides a prompt, which is then interpreted to produce a requirements document. Developers implement requirements, while QA engineers design checklists. Unlike the human development scenario, AI generates code faster and therefore demands a more efficient workflow with less communication and faster testing. To achieve this goal, the link between requirement engineering and the testing workflow in Figure 1 can be removed: the model needs to autonomously discover user-perceivable bugs through review or testing, without relying on a predefined checklist.

The testing workflow consumes information handed off directly from the development side and proceeds through layers of inspection. (1) *Developer-Side*: an inspection that requires professional expertise and is conducted from the developer's perspective, such as code review. (2) *Hybrid-Side*: an inspection that combines professional expertise with user-perspective experience, e.g., examining page source and element rendering with browser developer tools, such as Playwright-based testing. (3) *User-Side Test*: an inspection accessible even to users without specialized expertise, who can only rely on keyboard/mouse interactions and visual feedback, such as PyAutoGUI-based testing. Purely developer-side inspection lacks interaction and visual feedback, whereas purely user-side testing is constrained by the model's on-screen element-localization ability and cannot leverage the assistance provided by the browser developer tools. We therefore evaluate the VLM's capability for Hybrid-Side testing with Playwright. The VLM writes Playwright code and uses it to interact with the browser, gather textual and visual feedback, and autonomously explore the web application to discover bugs. We term this paradigm as **C**ode-driven **A**gentic **T**esting (CAT).

3.2 CATJudge Framework

Task Definition We formulate autonomous CAT as a POMDP $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \Omega, \mathcal{R}, H)$ with horizon H. A state $s_t \in \mathcal{S}$ is the latent environment state of the deployed web application (e.g., DOM and runtime state, together with its unknown defects). At each turn the agent receives an observation $o_t \sim \Omega(\cdot \mid s_t)$ exposing only part of s_t through its tools (§3.2.2), and selects a tool action conditioned on the task context $\mathcal{I}$ (the testing instruction together with a QA-oriented project document, see §3.2.2) and the history $h_t = (\mathcal{I}, o_{0:t}, a_{0:t-1})$.

To keep the toolkit GUI-only, the action space contains only interaction tools and a termination tool $\mathcal{A} = \mathcal{A}_{\text{int}} \sqcup \{a_{\text{term}}\}$: interaction tools in $\mathcal{A}_{\text{int}}$

drive s_t via $\mathcal{T}$, while the termination action a_{term} ends the episode and emits a verdict (PASS or NOT_PASS) together with a textual list of discovered bugs $\mathcal{B}$. The episode also ends when H turns are reached. Letting T denote the terminal turn, we adopt a sparse, episodic reward

$$\mathcal{R}(s_T, a_T) = \text{score}(\mathcal{B}, \mathcal{B}^*) \in [0, 1], \quad (1)$$

where $\mathcal{B}^*$ is the ground-truth bug set and $\text{score}(\cdot, \cdot)$ is defined in §3.3.2.

3.2.1 Environment

Remote Testing Service For extensibility and flexible deployment, CATJudge runs the web application under test on a remote server (e.g., a Docker container) that is decoupled from the local client. Before each testing episode, the server automatically executes a predefined set of deployment commands (e.g., `npm install` and `npm run dev`) to bring up the application, and tears the service down upon episode termination or interruption. The execution interfaces of all tools are exposed to the local tool interpreter through a unified web API, so the agent can invoke remote tools through the client as if they were local.

Hybrid Automation & Task Isolation The architectures of existing BUA and CUA differ from each other (He et al., 2024; Browser Use, 2025; Xie et al., 2024), and there is a need for a unified runtime framework. As shown in Figure 2, to accommodate both browser-only BUA tools via Playwright and OS-level CUA tools for keyboard/mouse automation via PyAutoGUI within a single environment, the server launches an Xvfb virtual display: CUA actions can run on this virtual screen, while the browser is started and rendered on the same virtual display so that BUA actions can be executed in a real browser. This co-located design allows CUA and GUI automation to coexist and to interact with a consistent rendering of the web page. Building on this design, the framework supports two complementary isolation modes: run actions on different servers yields strong isolation, whereas allocating different virtual displays on a single server provides lightweight, soft isolation.

3.2.2 Workflow

Initial State Recent studies suggest that LLMs exhibit clear limitations in exploring and learning unfamiliar environments from scratch (Foundation, 2026). To improve testing efficiency, we therefore inject a QA-oriented project document into the initial state. As is common practice in AI-assisted coding pipelines, the document is supplied by the development side and offers a minimal yet sufficient guide for testing: an overview of implemented features, the technical context (e.g., tech stack, core logic, and key state variables), and the interaction surface relevant to testing. We also provide an initial screenshot of the web page.

Feedback & Observation Each tool call returns textual feedback that aggregates the kinds of signals a human QA engineer would obtain from invoking the corresponding tool during testing. For example, a Playwright call may return execution status and error messages, browser state and return values from injected scripts and DevTools commands. After all tool calls in a turn have executed, the server waits for a short delay to let the page settle in response to the interactions, and then captures a screenshot of the resulting state and returns it to the agent as part of the observations. To bound the context length, only the most recent screenshot is retained in the history, which also matches the visual feedback a human tester would actually have access to at any moment.

3.2.3 Tools

GUI Tools Our framework exposes BUA and CUA tools for GUI interactions. Each contains: (1) a set of encapsulated atomic operations following prior works (He et al., 2024; Xie et al., 2024); and (2) a script-execution interface through which the agent can freely author and run code.

Core Tool Set Our experiments show that code-capable models prefer the more flexible script-execution tool over the atomic operations (see § 4.2). We argue that, for LLMs, interacting with the environment by writing code is more native and more efficient than imitating low-level human action sequences. We therefore include only the script-execution tool into the core tool set for GUI interactions. In addition, we provide a special tool for emitting the termination action. Departing from prior work that feeds the a11y tree to the model as part of the input (He et al., 2024), we expose the a11y tree as a standalone tool that the agent may invoke at will, while screenshots are instead pushed to the agent at every turn to mitigate the mirage reasoning (Asadi et al., 2026) issue. We also provide a wait tool that lets the agent intentionally idle, either to allow time-dependent operations to

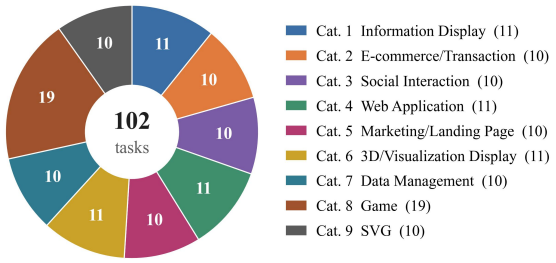


Figure 3: Distribution of the 102 tasks in CATTest across the 9 top-level categories.

take effect or to actively request a fresh screenshot.

3.3 CATTest Benchmark

To rigorously evaluate the code-driven agentic testing capability of VLMs, we construct CATTest, a benchmark dataset consisting of complex AI-generated web applications, paired with project documentations and carefully annotated ground-truth bugs (GT bugs).

3.3.1 Benchmark Construction

The construction of CATTest is a process of close human-AI collaboration, whose overall pipeline is illustrated in Figure 4. In total, CATTest contains 102 data points. Detailed statistics of CATTest are reported in Appendix A.1.

Web Code Generation To ensure broad coverage of interaction types, we first identify 9 popular usage scenarios for web applications. Within each scenario, drawing on representative real-world use cases and on category definitions on Wikipedia, we enumerate the interactions (e.g., element dragging, mouse-driven viewport rotation) required by each application, while keeping the overlap of required interactions among applications within the same top-level category as small as possible. This yields a batch of category metadata.

To ensure that GT bugs are sufficiently subtle and natrually, the generation tasks must be hard enough on one hand, and the code generator must be capable enough on the other, so that the elicited bugs reflect real defects that genuinely occur in AI coding workflows. To raise the difficulty of the generation tasks, we iteratively expand the requirement volume of each prompt following the requirement engineering workflow in Figure 1. Specifically, we first attach a concrete fictional usage context (e.g., an advertising site for a coffee product) to each task, and use the metadata to prompt Claude-Opus-4.6 (Anthropic, 2026a) for initial seed web development instructions. Following standard agile-development practice (Manifesto, 2001), we then perform a two-step decomposition of the seed instructions into feature lists that reflect coarse-grained requirements and user stories that reflect user paths. At each iteration, we leverage the user stories from the previous round to enrich the prompt with finer details and additional features, and redo the decomposition until the resulting user stories are sufficiently numerous, which in turn guarantees the difficulty of the task. The instructions are subsequently reviewed and cleaned by human annotators: (1) check whether the instruction targets a real-world website, and remove any related content to avoid copyright concerns; (2) strip over-specific technical constraints (e.g., self-containment requirements) so that the technology usage and distribution more closely match real-world AI coding scenarios; (3) remove requirements that CATJudge cannot support, e.g., replacing a real shopping transaction flow with a mock to avoid third-party services, or simulating a gyroscope with the mouse on a desktop environment; and (4) simplify tasks whose content volume exceeds the capacity of any available AI tool, reducing irrelevant textual content and repetitive technical points while preserving interaction richness (e.g., for a page that showcases product manuals, reducing the amount of product description text and the number of chapters).

To maximize the chance that bugs genuinely populate the coding-agent regime, we generate the web pages with Claude Code (Anthropic, 2025) backed by Claude-Opus-4.6, which represents the frontier of coding-agent capability, and obtain the corresponding project documents along the way. Appendix A.3 discusses this choice in detail. Finally, human annotators experience each generated web application to confirm the technical feasibility of the development instruction as well as the complexity and interactivity of the resulting project.

Bug Annotation This step is the most labor-intensive and the most challenging stage of the construction pipeline, as no single tool or expert can uncover every bug (Su et al., 2021). After manual testing, CATJudge can still discover a substantial number of bugs that human testers miss. We therefore undertake a heavy iterative pipeline of annotation and verification, following the testing workflow in Figure 1. First, the human experience step in data

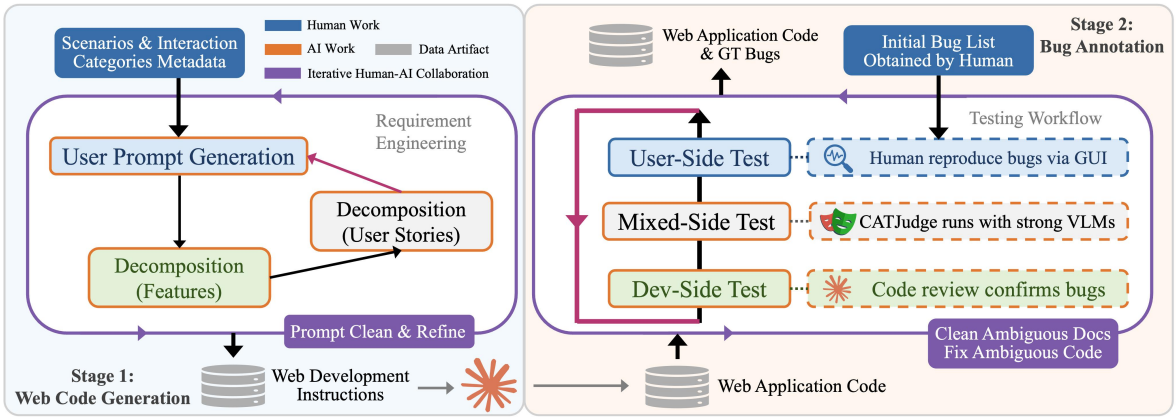


Figure 4: Overview of the CATTest benchmark construction. Dark blue denotes human work steps, orange denotes AI work steps, purple denotes iterative human-AI collaborative steps, and grey denotes data artifacts.

construction yields an initial bug list. For each bug on the list, a human annotator attempts to reproduce it through manual GUI actions to confirm the described phenomenon truly occurs. Claude Code backed by Claude-Opus-4.6 then analyzes the source code and the project document with respect to the bug report to determine whether the source code is indeed defective, with humans collaboratively working with it throughout the process. To reduce subjective judgment during evaluation and improve its stability, ambiguous wording in the generation instruction and the project document are simultaneously cleaned and rewritten to handle false positives caused by unclear requirements or misleading documentation (e.g., the drag-to-zoom direction is counter-intuitive). For bugs remaining ambiguous (e.g., pixel-level minor UI offsets), we instead modify the source code directly to eliminate them. To maximize bug discovery coverage, we adopt a tool-competition pooling strategy (Metzman et al., 2021): run CATJudge with multiple strong code-capable models (see Appendix A.2 for details), contribute additional bugs to the list, after which the bug confirming procedure is repeated. We iterate the loop until human annotators and Claude Code agree that every newly reported bug is either a false positive or a duplicate of a confirmed one (describing the same phenomenon, or textually being a cause of another). This yields the final GT bug set for evaluation.

3.3.2 Evaluation Metrics

To ensure that the evaluation metrics faithfully reflect the practical needs of AI coding, we design two complementary sets of metrics, at the coarse-grained and fine-grained levels respectively.

Coarse-grained Metrics We treat the verdict of whether a project passes testing as a binary classification task. Taking NOT_PASS as the positive class, we report the four classical metrics: Accuracy, Recall, Precision, and F1. Coarse-grained metrics do not fully reflect a model's true bug-finding ability, but they can serve as a reference for filtering training data for web code generation tasks.

Fine-grained Metrics For each project, we assign a score in $[0, 1]$ based on how many of the GT bugs are reported. We first use Gemini-3-Flash (Google, 2025) to match the model reported bugs $\mathcal{B}$ against the GT bugs $\mathcal{B}^*$. The matching criterion is the same as in annotation: two bugs are considered matched if they describe the same phenomenon, or if one is a cause or a direct consequence of the other. The full criterion, the matching procedure, and worked examples are given in Appendix A.4. We performed manual bug matching on the test outputs of Claude-Opus-4.7 and compared it with the automatic matching; as shown in Table 2 (§4.2), the automatic matching closely aligns with human judgment. Let $M \subseteq \mathcal{B} \times \mathcal{B}^*$ denote the set of matched pairs produced by the matcher, and write $\mathcal{B}^*_M = \{b^* \in \mathcal{B}^* \mid \exists b, (b, b^*) \in M\}$ and $\mathcal{B}_M = \{b \in \mathcal{B} \mid \exists b^*, (b, b^*) \in M\}$. We compute the scores as follows:

- **Recall score (R-score)**, the primary metric of CATTest, which measures how comprehensively the model uncovers bugs. A higher score indicates that more GT bugs are caught:

$$R = \begin{cases} 1, & |\mathcal{B}^*| = 0 \text{ and } |\mathcal{B}| = 0, \\ 0, & |\mathcal{B}^*| = 0 \text{ and } |\mathcal{B}| > 0, \\ \dfrac{|\mathcal{B}^*_M|}{|\mathcal{B}^*|}, & |\mathcal{B}^*| > 0, \end{cases} \tag{2}$$

Model	R-score↑	P-score↑	Acc↑	Recall↑	Precision↑	F1↑
Proprietary Models						
Claude-Opus-4.7	**42.57**	38.30	**83.33**	93.67	86.05	**89.70**
Claude-Opus-4.6	41.55	48.20	81.37	89.87	86.59	88.20
Claude-Sonnet-4.6	37.68	41.08	82.47	**98.65**	82.02	89.57
Gemini-3.1-Pro	36.49	**48.56**	73.53	82.28	83.33	82.80
Gemini-3-Flash	24.22	22.99	73.53	83.54	82.50	83.02
Gemini-3.1-Flash-Lite	20.92	23.37	48.04	40.51	84.21	54.70
GPT-5.5	39.41	46.88	76.47	86.08	83.95	85.00
GPT-5.4	17.58	22.37	76.47	93.67	79.57	86.05
GPT-5.4-mini	19.07	25.00	58.82	62.03	80.33	70.00
GLM-5V-Turbo	24.67	33.50	61.78	58.23	86.79	69.70
Doubao-Seed-2.0-Pro	22.79	26.55	50.98	44.30	85.37	58.33
Doubao-Seed-2.0-Lite	21.98	23.20	42.16	32.91	81.25	46.85
Open Models						
Qwen3.5-Plus	24.71	32.62	68.63	72.15	85.07	78.08
Qwen3.6-27B	33.56	37.92	71.57	72.15	**89.06**	79.72
Qwen3.5-27B	23.24	26.72	69.61	78.48	81.58	80.00
Qwen3.5-9B	14.71	14.71	43.14	25.37	68.00	36.96
Kimi-K2.6	30.78	29.43	69.61	75.95	83.33	79.47

Table 1: Main results of mainstream VLMs on CATTest under the CATJudge framework. R-score and P-score are the fine-grained metrics measuring bug-discovery ability. Acc, Recall, Precision, and F1 are the coarse-grained pass/not-pass classification metrics. **Bold** marks the best result per column, and underline marks the second best.

- **Precision score (P-score)**. Since the R-score does not penalize over-reporting, we additionally compute a P-score that measures the model's tendency to report bugs. A higher score indicates more accurate reporting, and a lower score suggests that the model tends to over-report:

$$P = \begin{cases} 1, & |\mathcal{B}^*| = 0 \text{ and } |\mathcal{B}| = 0, \\ 0, & |\mathcal{B}^*| > 0 \text{ and } |\mathcal{B}| = 0, \\ \dfrac{|\mathcal{B}_M|}{|\mathcal{B}|}, & |\mathcal{B}| > 0. \end{cases} \tag{3}$$

The per-project R-scores and P-scores are summed across the benchmark and normalized to $[0, 100]$, yielding the overall R-score and P-score.

4 Experiments

Experimental Setup We conduct experiments with the CATJudge framework on CATTest, using the core tool set (§3.2.3) that includes only the Playwright script-execution tool as the GUI tool. The maximum number of interaction turns is set to $H = 250$, and both the inter-tool interval within a turn and the post-execution screenshot delay are set to 0.1 s. For each API request, the maximum number of generated tokens is 16,384 and the temperature is 1.0. All the other parameters are kept at the provider's default values.

4.1 Main Results

Table 1 reports the performance of mainstream VLMs (see Appendix B for the full list) on CATTest under the CATJudge framework. We can draw the following observations.

Even the strongest models are far from solving the CAT tasks. The best R-score is only 42.57 (Claude-Opus-4.7) and the best P-score is only 48.56 (Gemini-3.1-Pro), meaning that no model can reliably uncover even half of the GT bugs while keeping its reports trustworthy. Claude-Sonnet-4.6 reaches a near-perfect Recall of 98.65 yet only obtains an R-score of 37.68. GPT-5.4 exhibits the same pattern (Recall 93.67 vs. R-score 17.58). These models are good at flagging that "something is wrong" with a project, but fail to identify the specific defects that triggered the failure.

A clear capability gap separates frontier models from the rest. Five models (Claude-Opus-4.7, Claude-Opus-4.6, GPT-5.5, Claude-Sonnet-4.6, and Gemini-3.1-Pro) form a leading tier with R-score above 36. Two open models follow at an intermediate level (Qwen3.6-27B at 33.56 and Kimi-K2.6 at 30.78), after which the remaining models fall below 25, leaving a gap of roughly six points between the intermediate group and the rest. Low-capacity and lightweight variants degrade further on the coarse-grained metrics: Gemini-3.1-Flash-Lite, Doubao-Seed-2.0-Lite, and Qwen3.5-9B are

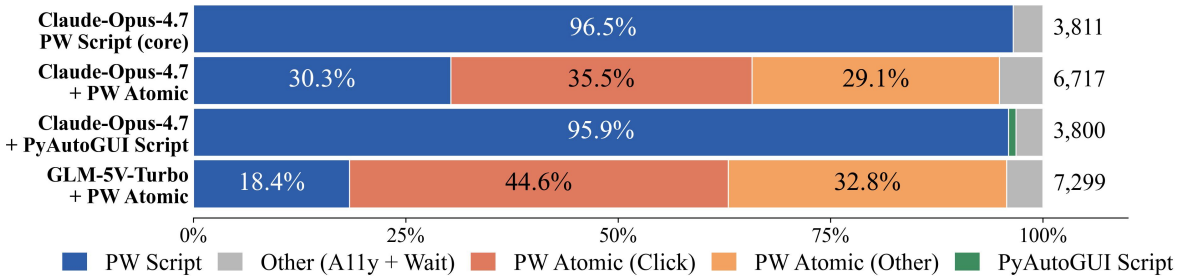


Figure 5: Tool-call distribution across the three tool-set configurations on Claude-Opus-4.7 (top three rows), together with the + *PW Atomic* configuration on the weaker GLM-5V-Turbo (bottom row) for comparison. Each horizontal bar shows the proportion of calls made to each available tool, and the number to the right of each bar reports the total number of tool calls under that configuration. *PW Atomic (Click)* denotes the Playwright atomic click operation, which is reported separately because it dominates the rest of the atomic operations; *PW Atomic (Other)* aggregates all remaining Playwright atomic operations.

Model	GT	Reported
Claude-Opus-4.7	97.38%	96.59%
GLM-5V-Turbo	98.43%	97.10%

Table 2: Agreement between Gemini-3-Flash-based automatic bug matching and human matching on the test outputs of Claude-Opus-4.7 and GLM-5V-Turbo. **GT** reports the fraction of GT bugs whose match label agrees with the human, and **Reported** reports the same on the model-reported bugs.

Config	R	P	Acc	Rec.	Prec.	F1
Hybrid	**42.57**	**38.30**	**83.33**	93.67	**86.05**	**89.70**
User	32.94	27.83	82.35	94.94	84.27	89.29
Developer	35.87	10.60	76.47	**98.73**	77.23	86.67

Table 3: Comparison of the three different testing workflow configurations evaluated on Claude-Opus-4.7. Here **R** and **P** denote the overall R-score and P-score, while **Rec.** and **Prec.** denote Recall and Precision. **Bold** marks the best result per column.

the only models whose Recall falls below 41, making them unreliable for practical GUI testing. The gap is also pronounced within the same family: GPT-5.5 more than doubles GPT-5.4 in R-score (39.41 vs. 17.58), suggesting that the CAT task is highly sensitive to the underlying coding capability and benefits disproportionately from the latest generation of frontier models.

4.2 Analysis

Hybrid-Side testing matters. To validate our choice of Hybrid-Side testing (§3), we compare three configurations in the testing workflow on the strongest model in our pool, Claude-Opus-4.7: (1) *Hybrid-Side*, the default CATJudge setup with the Playwright script-execution tool; (2) *User-Side*, CATJudge with the Playwright tool replaced by PyAutoGUI; and (3) *Developer-Side*, where Claude Code performs static code review without any browser interaction. As shown in Table 3, Hybrid-Side delivers the best R-score and P-score by a clear margin. Switching to PyAutoGUI drops the R-score, since on-screen element localization becomes the bottleneck. Hybrid-Side is also cheaper than User-Side, as reported in Table 8 (§C). Pure code review yields a competitive R-score but its P-score collapses, confirming that without interactive and visual feedback the model massively over-reports speculative defects that do not actually manifest at runtime.

Tool preference varies with model capability. Starting from the core tool set (§3.2.3), we additionally expose the Playwright atomic operations and the PyAutoGUI script-execution tool in two separate configurations, and record the resulting tool-call distributions on both Claude-Opus-4.7 and GLM-5V-Turbo (Figure 5, full metrics in Table 6). The strong model concentrates its calls on Playwright script-execution and seldom invokes either alternative, whereas the weaker model spreads most of its calls across the atomic operations. Beyond the relative distribution, the totals on the right of Figure 5 show that script-execution also substantially reduces the absolute number of tool calls and makes the testing more efficient.

5 Conclusion

This paper studies whether VLMs can act as autonomous QA engineers for AI-generated web applications. Starting from a workflow model of

agentic web generation, we address the lack of an evaluation paradigm with three contributions: a code-driven testing paradigm in which the agent treats Playwright code as its primary interface to the browser, an agentic framework that unifies BUA and CUA tools inside a single isolated environment, and a benchmark of 102 web projects with carefully annotated ground-truth bugs. Across mainstream VLMs, the highest score we measure remains well below the level required for unattended deployment, and a sizable performance cliff separates frontier models from the rest, indicating that bug discovery in real web projects is a strong probe of coding intelligence than benchmark scores on conventional code generation.

Limitations

Our work aims to lay a foundation for code-driven agentic testing in web development. Nevertheless, several limitations remain: (1) Completeness of GT Bugs: As discussed in §3.3.1, bug annotation is a highly labor-intensive process and is unlikely to be exhausted by any single effort. Despite our best efforts to identify GT bugs as comprehensively as possible, some defects inevitably remain undiscovered. Therefore, R-score does not penalize a model for reporting additional bugs on projects with bugs, and is complemented by the P-score. We expect the GT bug set to be continuously expanded through future use. (2) Evaluation of Subjective Metrics: During GT bug annotation, we clean and revise the project documents and source code to minimize subjectivity and stabilize evaluation. However, we still observe that models sometimes report minor, barely user-perceivable issues as bugs. For example, in project 99, the background animation continues to play after gameover, which is in fact common and acceptable behavior in many real-world games. In addition, to keep the evaluation as objective as possible, we focus only on functional correctness (e.g., interactive functionality, layout and rendering correctness, and performance) and do not evaluate subjective dimensions specified in web development standards, such as accessibility, user experience, and visual style. This is a deliberate scoping decision rather than a framework restriction: covering such dimensions would only require supplying additional criteria to the CATJudge agent, but it would come at the cost of evaluation stability, since models differ in how they interpret and apply subjective scoring criteria, and the resulting scores are correspondingly harder to reproduce. We therefore concentrate on functional correctness, which is the more fundamental requirement and one that current models already fail to guarantee, and we regard the evaluation of subjective dimensions as complementary work best carried out by dedicated benchmarks, so that a healthy evaluation ecosystem is built jointly from multiple efforts rather than from any single one.

Acknowledgments

This research was supported by the National Natural Science Foundation of China (Grants No.62477044), the Key Technologies R&D Program of Anhui Province (No.202423k09020039), the Young Elite Scientists Sponsorship Program by CAST (No. 2024QNRC001), the Fundamental Research Funds for the Central Universities (No.WK2150110038).

References

Emil Alégroth, Robert Feldt, and Lisa Ryrholm. 2015. Visual gui testing in practice: challenges, problemsand limitations. *Empirical Software Engineering*, 20(3):694–744.

Anthropic. 2025. Overview - claude code docs. https://code.claude.com/docs/en/overview. Accessed: 2026-05-25.

Anthropic. 2026a. Claude opus 4.6 anthropic. https://www.anthropic.com/news/claude-opus-4-6. Accessed: 2026-05-25.

Anthropic. 2026b. Introducing claude opus 4.7 anthropic. https://www.anthropic.com/news/claude-opus-4-7. Accessed: 2026-05-25.

Anthropic. 2026c. Introducing sonnet 4.6 anthropic. https://www.anthropic.com/news/claude-sonnet-4-6. Accessed: 2026-05-25.

Mohammad Asadi, Jack W O'Sullivan, Fang Cao, Tahoura Nedaee, Kamyar Rajabalifardi, Fei-Fei Li, Ehsan Adeli, and Euan Ashley. 2026. Mirage: The illusion of visual understanding. *arXiv preprint arXiv:2603.21687*.

Rabiul Awal, Mahsa Massoud, Aarash Feizi, Zichao Li, Suyuchen Wang, Christopher Pal, Aishwarya Agrawal, David Vazquez, Siva Reddy, Juan A Rodriguez, and 1 others. 2025. Webmmu: A benchmark for multimodal multilingual website understanding and code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25129–25156.

Yutong Bian, Xianhao Lin, Yupeng Xie, Tianyang Liu, Mingchen Zhuge, Siyuan Lu, Haoming Tang, Jinlin

Wang, Jiayi Zhang, Jiaqi Chen, and 1 others. 2025. You don't know until you click: Automated gui testing for production-ready software evaluation. *arXiv preprint arXiv:2508.14104*.

Browser Use. 2025. Browser use - the way ai uses the internet. https://browser-use.com/. Accessed: 2026-05-22.

ByteDance Seed. 2026. Bytedance seed. https://seed.bytedance.com/en/seed2. Accessed: 2026-05-26.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yang Chen, Minghao Liu, Yufan Shen, Yunwen Li, Tianyuan Huang, Xinyu Fang, Tianyu Zheng, Wenxuan Huang, Cheng Yang, Daocheng Fu, and 1 others. 2025. Iwr-bench: Can lvlms reconstruct interactive webpage from a user interaction video? *arXiv preprint arXiv:2509.24709*.

ARC Foundation. 2026. Arc-agi-3: A new challenge for frontier agentic intelligence. *arXiv preprint arXiv:2603.24621*.

Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, and 1 others. 2025. A survey of vibe coding with large language models. *arXiv preprint arXiv:2510.12399*.

Google. 2025. Gemini 3 flash | gemini enterprise agent platform | google cloud documentation. https://docs.cloud.google.com/gemini-enterprise-agent-platform/models/gemini/3-flash. Accessed: 2026-05-25.

Google. 2026. Gemini 3.1 pro | gemini enterprise agent platform | google cloud documentation. https://docs.cloud.google.com/gemini-enterprise-agent-platform/models/gemini/3-1-pro. Accessed: 2026-05-25.

Google DeepMind. 2026. Gemini 3.1 flash-lite | gemini enterprise agent platform | google cloud documentation. https://docs.cloud.google.com/gemini-enterprise-agent-platform/models/gemini/3-1-flash-lite. Accessed: 2026-05-26.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890.

Wenyi Hong, Xiaotao Gu, Ziyang Pan, Zhen Yang, Yuting Wang, Yue Wang, Yuanchang Yue, Yu Wang, Yanling Wang, Yan Wang, and 1 others. 2026. Glm-5v-turbo: Toward a native foundation model for multimodal agents. *arXiv preprint arXiv:2604.26752*.

Xueyu Hu, Tao Xiong, Biao Yi, Zishu Wei, Ruixuan Xiao, Yurun Chen, Jiasheng Ye, Meiling Tao, Xiangxin Zhou, Ziyu Zhao, and 1 others. 2025. Os agents: A survey on mllm-based agents for computer, phone and browser use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7436–7465.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pages 58791–58831.

Hao Jiang, Qi Liu, Rui Li, Shengyu Ye, and Shijin Wang. 2024. Cursorcore: Assist programming through aligning anything. *arXiv preprint arXiv:2410.07002*.

Hao Jiang, Qi Liu, Rui Li, Yuze Zhao, Yixiao Ma, Shengyu Ye, Junyu Lu, and Yu Su. 2025. Verse: Verification-based self-play for code instructions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24276–24284.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2026. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch. *Advances in Neural Information Processing Systems*, 38.

Agile Manifesto. 2001. Manifesto for agile software development.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1393–1403.

Microsoft. 2025. microsoft/playwright: Playwright is a framework for web testing and automation. it allows testing chromium, firefox and webkit with a single api. https://github.com/microsoft/playwright. Accessed: 2026-05-22.

MiniMax. 2025. Vibe: Visual & interactive benchmark for execution in application development.

Atsuyuki Miyai, Zaiying Zhao, Kazuki Egashira, Atsuki Sato, Tatsumi Sunada, Shota Onohara, Hiromasa Yamanishi, Mashiro Toyooka, Kunato Nishina, Ryoma Maeda, and 1 others. 2025. Webchorearena: Evaluating web browsing agents on realistic tedious web tasks. *arXiv preprint arXiv:2506.01952*.

Moonshot AI. 2026. Kimi k2.6 | leading open-source model in coding & agent. https://www.kimi.com/ai-models/kimi-k2-6. Accessed: 2026-05-26.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, and 1 others. 2025. Gui agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22522–22538.

OpenAI. 2026a. Introducing gpt-5.4 | openai. https://openai.com/index/introducing-gpt-5-4/. Accessed: 2026-05-25.

OpenAI. 2026b. Introducing gpt-5.4 mini and nano | openai. https://openai.com/index/introducing-gpt-5-4-mini-and-nano/. Accessed: 2026-05-25.

OpenAI. 2026c. Introducing gpt-5.5 | openai. https://openai.com/index/introducing-gpt-5-5/. Accessed: 2026-05-26.

Mauro Pezze, Paolo Rondena, and Daniele Zuddas. 2018. Automatic gui testing of desktop applications: an empirical assessment of the state of the art. In *Companion proceedings for the issta/ecoop 2018 workshops*, pages 54–62.

Qwen Team. 2026a. Qwen. https://qwen.ai/blog?id=qwen3.5. Accessed: 2026-05-26.

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

Qwen Team. 2026c. Qwen3.6-27B: Flagship-level coding in a 27B dense model.

SeleniumHQ. 2025. Seleniumhq/selenium: A browser automation framework and ecosystem. https://github.com/SeleniumHQ/selenium. Accessed: 2026-05-22.

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.

Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 119–130.

Haoyu Sun, Huichen Will Wang, Jiawei Gu, Linjie Li, and Yu Cheng. 2025. Fullfront: Benchmarking mllms across the full front-end engineering workflow. *arXiv preprint arXiv:2505.17399*.

Al Sweigart. 2025. asweigart/pyautogui: A cross-platform gui automation python module for human beings. used to programmatically control the mouse & keyboard. https://github.com/asweigart/pyautogui. Accessed: 2026-05-22.

Ennio Visconti, Christos Tsigkanos, and Laura Nenzi. 2022. Webmonitor: Verification of web user interfaces. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4.

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

Kuang-Da Wang, Zhao Wang, Yotaro Shimose, Wei-Yao Wang, and Shingo Takamatsu. 2025b. Webgen-v bench: Structured representation for enhancing visual design in llm-based web generation and evaluation. *arXiv preprint arXiv:2510.15306*.

Xueqing Wu, Zihan Xue, Da Yin, Shuyan Zhou, Kai-Wei Chang, Nanyun Peng, and Yeming Wen. 2025. Frontalk: Benchmarking front-end development as conversational code generation with multi-modal feedback. *arXiv preprint arXiv:2601.04203*.

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. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Naimeng Ye, Xiao Yu, Ruize Xu, Tianyi Peng, and Zhou Yu. 2025. Ai agents for web testing: A case study in the wild. In *NeurIPS 2025 Workshop on Bridging Language, Agent, and World Models for Reasoning and Planning*.

Shengyu Ye, Qi Liu, Hao Jiang, Zheng Zhang, Heng Yu, and Zhenya Huang. 2026. Themis: Automated constraint-aware test synthesis framework for code reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 34432–34440.

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*.

Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. 2024. Repair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 16415–16429.

Chijin Zhou, Quan Zhang, Bingzhou Qian, and Yu Jiang. 2025. Janus: Detecting rendering bugs in web browsers via visual delta consistency. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 2702–2713.

A CATTest Details

A.1 Statistics

CATTest contains 102 AI-generated web applications, organized into 9 top-level categories (Cat. 1–9) that are further divided into fine-grained subcategories. Each subcategory is associated with a *core interaction* that characterizes its primary input modality and feedback loop. Figure 3 summarizes the distribution of tasks across the 9 top-level categories. To give a concrete sense of how the subcategory taxonomy is designed, Table 9 shows the full subcategory list for Cat. 8 (Game), which is the largest category in CATTest, together with the core interaction of each subcategory.

Prompt Complexity To quantify the difficulty implicitly carried by the development prompts in CATTest, we apply the same two-stage decomposition pipeline used during CATTest construction (see §3.3.1) to the prompts of two open-source web generation benchmarks, WebGen-Bench (Lu et al., 2026) and the web subset of VIBE (MiniMax, 2025), and count the resulting user stories per item. A larger story count indicates a richer set of latent requirements and, in turn, a more demanding generation task. We use the same backbone model, system prompts, and decoding configuration across all three datasets, so the per-item story count is directly comparable. Table 4 reports the resulting statistics. CATTest yields the highest average number of stories per item (43.40), 1.43× that of WebGen-Bench and 1.31× that of VIBE-web. Even the prompts annotated as *hard* in VIBE-web (35.69 stories per item) fall short of the CATTest average. Within CATTest, the most complex categories are Game (Cat. 8, 53.63) and Web Application (Cat. 4, 52.45), both of which involve high interaction density and rich state.

Subset	#Items	#Stories	Avg.
CATTest (Ours)			
All	102	4,427	**43.40**
Information Display	11	341	31.00
E-commerce/Transaction	10	410	41.00
Social Interaction	10	437	43.70
Web Application	11	577	52.45
Marketing/Landing Page	10	359	35.90
3D/Visualization Display	11	477	43.36
Data Management	10	423	42.30
Game	19	1,019	53.63
SVG (DOM-Interactive)	10	384	38.40
WebGen-Bench			
All	101	3,072	**30.42**
Content Presentation	28	714	25.50
Data Management	24	793	33.04
User Interaction	49	1,565	31.94
VIBE (web subset)			
All	40	1,327	**33.17**
Easy	13	397	30.54
Medium	14	466	33.29
Hard	13	464	35.69

Table 4: Prompt complexity comparison across CATTest and two open-source web generation benchmarks. **#Items** is the number of prompts; **#Stories** is the total number of user stories obtained from the two-stage decomposition pipeline; **Avg.** is the average number of stories per item. Within each dataset, items are further grouped by their native taxonomy (top-level category for CATTest, primary category for WebGen-Bench, and difficulty level for VIBE-web).

GT Bug Statistics Unlike debug-oriented benchmarks that manually inject defects into otherwise correct code, we obtain GT bugs directly from AI-coding artifacts by raising the difficulty of the generation prompts. Although we generate the web applications with Claude Code backed by Claude-Opus-4.6, which represents the frontier of AI-coding capability, and run the generation pipeline only once without iterative regeneration or self-repair, the resulting projects still expose a substantial number of bugs. Table 5 summarizes the GT bug statistics. Treated as a web generation benchmark, CATTest yields a project-level pass rate (i.e., the fraction of projects whose GT bug set is empty) of only 22.55% for Claude Code + Claude-Opus-4.6, with 1.86 bugs on average per project. The bug-bearing rate is high across nearly all categories, reaching 100% for Marketing/Landing Page (Cat. 5), 90.91% for 3D/Visualization Dis-

Category	#Items	#B	B%	Avg.
All	102	190	77.45	1.86
Information Display	11	8	45.45	0.73
E-commerce/Transaction	10	20	70.00	2.00
Social Interaction	10	24	80.00	2.40
Web Application	11	32	72.73	2.91
Marketing/Landing Page	10	20	100.00	2.00
3D/Visualization Display	11	20	90.91	1.82
Data Management	10	14	60.00	1.40
Game	19	37	89.47	1.95
SVG (DOM-Interactive)	10	15	80.00	1.50

Table 5: GT bug statistics on CATTest, broken down by top-level category. **#B** is the total number of GT bugs in the category; **B%** is the fraction of projects in the category that contain at least one GT bug (i.e., $1-$ pass rate); **Avg.** is the average number of GT bugs per project.

Tool Set	R	P	Acc	Rec.	Prec.	F1
PW Script	**42.57**	38.30	**83.33**	**93.67**	86.05	**89.70**
+ PW Atomic	35.07	30.20	82.35	89.87	**87.65**	88.75
+ PA Script	40.13	**43.24**	81.37	91.14	85.71	88.34

Table 6: Tool set ablation evaluated on Claude-Opus-4.7. *PW Script (core)* is the core tool set used in the main experiments. + *PW Atomic* additionally exposes the Playwright atomic operations; + *PA Script* additionally exposes the PyAutoGUI script-execution tool. **R** and **P** denote the overall R-score and P-score, while **Rec.** and **Prec.** denote Recall and Precision. **Bold** marks the best result per column.

play (Cat. 6), and 89.47% for Game (Cat. 8). This confirms the effectiveness of our difficulty-control strategy: prompts complex enough to overwhelm a state-of-the-art coding agent naturally elicit a rich pool of real defects, removing the need for manual bug injection.

A.2 Models Involved in Bug Annotation

During the bug annotation stage of the benchmark construction, we adopt a tool-competition pooling strategy. To this end, we run several strong VLMs on CATJudge to discover bugs. The models involved in this process are: Claude-Opus-4.7, Claude-Opus-4.6, Gemini-3.1-Pro, Gemini-3-Flash, Gemini-3.1-Flash-Lite, GPT-5.5, GLM-5V-Turbo, and Qwen3.5-Plus.

A.3 On the Choice of Code Generator

The evaluation scope of CATTest is deliberately restricted to *subtle, naturally occurring* bugs in coding-agent scenarios (§3.3.1), and our use of a frontier coding agent as the generator is a design decision aligned with that scope.

The purpose of the generation stage is to raise the difficulty of the *debugging* task. A weaker generator would work against this purpose in two ways. First, it lowers the ceiling of bug subtlety: the defects it leaves behind tend to be coarse and immediately visible, so every tester scores high and the benchmark loses its power to discriminate among testers of different strength. Second, and more damaging, weaker generators frequently fail before the interactive layer is ever reached. In our preliminary trials with less capable generator configurations, a large fraction of the resulting projects did not compile, crashed on start-up, or failed to launch at all because of syntax and configuration errors. Such projects are trivially rejected by any tester without a single meaningful interaction, and they are therefore useless for evaluating interaction-driven bug discovery, which is precisely the ability CATTest is designed to measure. Using a frontier generator instead yields projects that run correctly, look plausible, and fail only under specific interaction sequences, which is the regime of interest.

Two further points bound the influence of this choice on the validity of the evaluation. First, the evaluation is anchored on the GT bug set rather than on the generator: the generation prompts are never exposed to the models under test, which only observe a fixed, already-running web project, and the GT bug set is produced by a tool-competition pooling strategy that combines human verification with several strong models from different families (§A.2). Second, a tester that misses a GT bug reveals a genuine capability deficiency, independently of whether the same defect would also arise in code produced by some other generator. Finally, the construction pipeline (Figure 4) is fully specified and extensible: alternative generators can be substituted and additional projects annotated under the same protocol, so CATTest can be extended along whichever axis a future evaluation goal requires.

A.4 Bug Matching Criterion

Both the annotation stage (§3.3.1) and the fine-grained metrics (§3.3.2) rely on deciding whether a reported bug and a GT bug refer to the same underlying defect. Because a single defect can surface as several distinct user-visible symptoms, matching cannot be reduced to textual similarity. We therefore adopt the following criterion. A pair

of bugs is considered a match if and only if at least one of the following conditions holds:

1. the two descriptions refer to essentially the same functional problem, even when worded differently;
2. the first describes a symptom or behavior that is directly caused by the same underlying defect as the second;
3. the first describes a consequence of the second, i.e. it would not occur if the second were fixed.

Matching is performed as an all-pairs comparison between the reported bug list and the GT bug list, with the criterion applied to each pair independently; each reported bug is accompanied by a description, reproduction steps, and expected versus actual behavior (Figure 20), which supplies the evidence needed to decide the relation. The resulting matched-pair set is the set M used in §3.3.2.

Partial matches and supersets. The two lists need not describe a defect at the same granularity, and the resulting containment can run in either direction: a single reported bug may cover several GT bugs, or a single GT bug may be split across several reported bugs. Either way, condition (1) is applied part-wise, so a bug that captures only a component of its counterpart still matches it. Project 38, a cloud-server terminal emulator, illustrates the first direction. Two GT bugs, *"cursor position deviates from the command-line prompt"* and *"after input, the text position shifts upward relative to the cursor position"*, are both judged to match the single reported bug *"input line broken: typed text and cursor render far away from the prompt and on separate lines"*, since each GT bug describes one component of the broken layout that the report states as a whole. The converse case, in which one GT bug is only jointly covered by several reported bugs, is handled symmetrically.

Causal relationships. The two lists may also describe a defect at different points along the same causal chain, and again either side may be the deeper one. GT bugs are first recorded from the user's perspective during the human experience step (§3.3.1), whereas a tester that inspects the page source is free to report the underlying root cause directly; conversely, a tester may report only a downstream symptom of a GT bug stated at its root. Conditions (2) and (3) cover these two directions symmetrically, so a match does not require the two descriptions to sit at the same depth. Project 58, a 3D building-model visualization, illustrates the case where the report is the deeper of the two: the GT bug *"measurement numbers can only be cleared using the Clear All button"* is matched by the reported bug *"per-measurement delete (×) button is unreachable due to* `pointer-events: none` *on the label"*, because the GT symptom is a direct consequence of the reported root cause. Had the report instead noted only that individual measurements cannot be removed, it would match the same GT bug in the opposite direction.

A counterexample. Neither direction may be granted on superficial plausibility alone: the causal link has to actually hold. In the *Insufficient Interaction Coverage* case (§D.2), the reported *"Delete Node does not work"* reads as a downstream symptom of the GT bug *"single-click cannot select a node"*, since a node that cannot be selected cannot be deleted, which would make it a match under condition (3). The deletion functionality is nevertheless correct: a node can still be selected by a box-select drag and then deleted successfully. The apparent dependency is thus an artifact of the report being written from an incomplete exploration rather than a real causal relation, so the two bugs are *not* matched, and the GT bug counts as missed.

A.5 Example

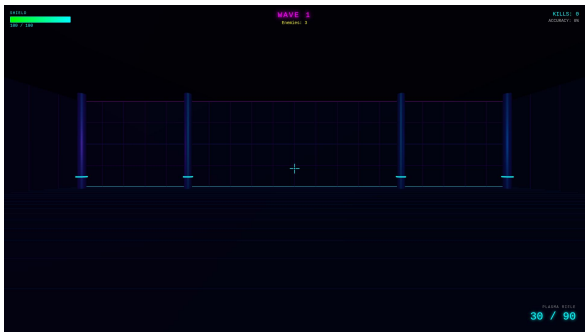
To give a concrete sense of what a CATTest data point looks like, we use Project 71 (*Neon Purge*) as a running example. This project belongs to top-level category Cat. 8 (Game) and the subcategory *FPS / Shooter*, whose core interaction is *Mouse aim + left-click shoot, WASD move, crosshair feedback, ammo management* (See Table 9). The example consists of four parts: (i) the development prompt sent to Claude Code (Figure 7); (ii) the QA-oriented project document handed off to the agent under test (Figure 8); (iii) screenshots of the resulting web application (Figure 6); and (iv) the GT bugs list.

GT Bugs. Project 71 contains a single GT bug:

> *Pausing during the countdown between waves does not interrupt the countdown, and the pause overlay gets stuck on screen and does not disappear.*

This defect manifests only at the boundary between two game states (WAVE_TRANSITION → COUNTDOWN → PLAYING). Reproducing it requires the

(a) Start screen with the title "NEON PURGE".



(b) In-game state during a wave, showing the cyberpunk arena.

Figure 6: Screenshots of the web application generated for Project 71 (*Neon Purge*, Cat. 8 / FPS-Shooter).

agent to actively drive the game into the wave-transition window and press `Esc` at the right moment, illustrating the kind of subtle, interaction-dependent defects that motivate CAT.

B Experiment Details

VLMs involved in the experiments We aim to cover the strongest model released by each major provider. To be eligible for our evaluation, a model must possess image-understanding capability. The full list of models is as follows:

- Claude series: Claude-Opus-4.7 (Anthropic, 2026b), Claude-Opus-4.6 (Anthropic, 2026a), Claude-Sonnet-4.6 (Anthropic, 2026c).
- Gemini series: Gemini-3.1-Pro (Google, 2026), Gemini-3-Flash (Google, 2025), Gemini-3.1-Flash-Lite (Google DeepMind, 2026).
- GPT series: GPT-5.5 (OpenAI, 2026c), GPT-5.4 (OpenAI, 2026a), GPT-5.4-mini (OpenAI, 2026b).
- Qwen series: Qwen3.5-Plus (Qwen Team, 2026a), Qwen3.6-27B (Qwen Team, 2026c), Qwen3.5-27B (Qwen Team, 2026b), Qwen3.5-9B (Qwen Team, 2026b).
- Others: GLM-5V-Turbo (Hong et al., 2026), Doubao-Seed-2.0-Pro (ByteDance Seed, 2026), Doubao-Seed-2.0-Lite (ByteDance Seed, 2026), Kimi-K2.6 (Moonshot AI, 2026).

C Additional Experiments

Evaluation is Stable on CATTest. To assess the run-to-run stability of CATTest under our default setting, we repeat the full evaluation of Claude-Opus-4.7 five times and report the mean and standard deviation of every metric in the *w/ doc* rows of Table 7. These numbers indicate that CATTest yields reproducible measurements across independent runs of the same model.

Setting	R	P	Acc
w/ doc	$40.64_{\pm 2.19}$	$38.79_{\pm 2.73}$	$80.78_{\pm 1.91}$
w/o doc	27.17	23.25	78.43
	Rec.	**Prec.**	**F1**
w/ doc	$89.87_{\pm 4.29}$	$86.02_{\pm 1.43}$	$87.84_{\pm 1.52}$
w/o doc	93.67	81.32	87.06

Table 7: Evaluation on Claude-Opus-4.7. The *w/ doc* rows report mean$_{\pm \text{std}}$ over five runs of the default setting; the *w/o doc* rows report a single run with the in-context project documentation removed. **R** and **P** denote the overall R-score and P-score; **Rec.** and **Prec.** denote Recall and Precision.

Basic documental guidance is important. We ablate the project documentation on Claude-Opus-4.7. Comparing the two rows of Table 7, the coarse-grained metrics move only modestly. The fine-grained R-score and P-score, however, fall sharply from 40.64 to 27.17 and from 38.79 to 23.25, a gap of more than five times the corresponding standard deviations. This pattern supports our design choice of providing minimal documental guidance.

Hybrid-Side testing is also cheaper. Beyond the accuracy gap reported in Table 3, we compare the Hybrid-Side and User-Side configurations in terms of testing cost on Claude-Opus-4.7. Table 8 reports the average wall-clock time to complete the testing of one project, together with the total number of API calls, the total number of tokens consumed, and the total inference cost aggregated over the whole benchmark. Hybrid-Side testing is not only more accurate but also uniformly cheaper along every axis: it finishes a project in 9.1 minutes on average against 13.9 minutes for User-Side, and consumes roughly 65% of the API calls, 45% of the tokens, and 58% of the monetary cost. The gap follows directly from the interaction granularity of

Config	Time	Calls	Tokens	Cost
Hybrid	**9.1**	**3,912**	**0.89**	**181**
User	13.9	5,982	1.96	311

Table 8: Testing cost of the Hybrid-Side and User-Side configurations on Claude-Opus-4.7. **Time** is the average wall-clock time spent on one project, in minutes; **Calls** is the total number of API calls over the benchmark; **Tokens** is the total number of tokens consumed, in billions; **Cost** is the total inference cost in USD. **Bold** marks the better result per column.

the two paradigms. A single Playwright script can express a compound interaction and return an aggregated observation in one turn, whereas driving the same interaction through keyboard and mouse primitives requires many turns, each paying the cost of a full model call and each carrying an additional screenshot into the context. Code as an interface is thus the more economical modality as well as the more capable one.

D Case Study

D.1 Good Cases

A useful way to see why a code-driven testing paradigm matters is to consider the projects that are **nearly impossible** to test through atomic mouse and keyboard Playwright operations alone. Browser games rendered on a single `canvas` surface, or on WebGL, are the clearest example: from the outside, the page exposes essentially one large opaque rectangle, with no individually addressable buttons, lists, or text fields for an atomic testing tool to target (for example, a single `<canvas>` node in the a11y tree, without any semantic information). A bug that only appears at the two-minute mark of a shoot-em-up, or in a transition between game modes, demands that the tester actually *play* the game to that point, and that often means surviving randomized enemy waves, collecting the right power-ups, and reaching specific boss states. For an AI agent, this is both slow and unreliable: a single attempt at a late-game state can take many minutes of real-time play and depends on stochastic events the agent has no stable way to control with GUI actions alone. Sometimes a human can have trouble playing a game. The natural alternative is to have the agent write code that interacts with the application's internal logic, rather than chase the moving target through the rendered surface.

The good case below illustrates exactly this. We walk through a representative trajectory produced by GPT-5.5 on Project 99 (*Cosmic Surge*, a 2D shoot-em-up). Two of the project's documented behaviors, a boss that spawns around the two-minute mark and a higher-difficulty loop that begins after the boss is defeated, are practically out of reach for play-only exploration. Instead of playing, the agent treats the page itself as a programmable artifact. Powered with free-access to Playwright script-execution, it first downloads the page's source code, then writes a small piece of JavaScript that installs a backdoor into the game's internal state and reloads the page with that backdoor in place. Once the backdoor is installed, the agent no longer has to play: it can directly read out variables that are normally private (the current wave timer, the boss HP bar, the active enemies on screen) and overwrite them at will (jumping the timer forward, spawning enemies at chosen positions, draining boss HP to one shot). All bug detections in this run, namely an inflated first-kill combo counter and a boss bullet that incorrectly carries over into the next loop, are obtained through this approach rather than by normally playing the game. We walk through the key turns of the trajectory in Figures 9–13. For each turn we show, side-by-side, the screenshot captured after the action, the agent's verbatim reasoning, the script it executed, a natural-language summary of the script's purpose, and the structured return values it observed.

D.2 Error Patterns

Mirage Reasoning. A recurring failure mode is *mirage reasoning* (e.g., the gent tends to ignore visual information or ignore textual information (Asadi et al., 2026)): the agent talks itself into a coherent story about why a strange observation is fine, and then treats that story as ground truth, instead of returning to the visual evidence. Figure 14 illustrates the pattern on Project 45, a flipping-card countdown timer. The screenshot shows the digits visibly malformed: each cell, which is meant to hold a single digit, is showing two digits stacked on top of each other. After clearing routine checks elsewhere on the page, the agent fast-forwards the timer and reads back the contents of the digit cells, finding roughly twice as many digits as the design calls for. Rather than weigh that count against the static screenshot in front of it, the agent decides the extra digits must be a fleeting frame of an in-progress flip animation; a follow-up check that merely confirms an animation is happening is then taken as proof that the timer is fine. The

verdict is PASS with zero bugs. The failure is not perceptual, the broken digits are plainly visible, but a willingness to prefer a fluent explanation ("a mid-flip frame") to the simpler reading that the page is showing twice as many digits as it should.

Insufficient Interaction Coverage. A second pattern is what we call *insufficient interaction coverage*: the agent reaches for high-level shortcuts such as "click this" or "press that" by default, and never constructs the more elaborate gestures that the project's edge-case bugs actually require, or would help to reveal the true bug. Figure 15 illustrates the pattern on Project 34 (*FlowWeave*, a node editor). Three of the four documented bugs are gesture-shaped: dragging a node and releasing the mouse over the right-hand panel makes the node keep following the cursor; with snap enabled the drag visibly lags behind the cursor; and a single click cannot select a node, only the box-select drag does. Across the run the agent issues a long string of high-level interactions but only a handful of true mouse-press-move-release gestures, all of them safely contained inside the canvas; it never tries finishing a drag outside the canvas, never reruns a drag with snap turned on, and never compares plain clicking against the alternative selection paths. As a result the report flags five surface anomalies, but only one of them lines up with a real bug (the right-click "Delete Connection" being inert). The agent even writes that "no node has visual selection highlight," a single follow-up step away from discovering that single-click does not select at all, which is the true bug leading to the observed "Delete Node does not work" phenomenon (the "Delete Node" function itself works as expected).

Spec Hallucination. A third pattern, complementary to mirage reasoning, is what we call *spec hallucination*: the agent imports its prior beliefs about how a class of applications "ought to" behave and treats those beliefs as binding requirements, even when the project's documentation says nothing of the kind, and even when the agent's own follow-up observations contradict the invented rule. Figure 16 illustrates the pattern on Project 13 (*BloomBox*, a gardening e-commerce site), a project that carries no documented bugs at all. After completing a normal checkout, the agent notices that the cart counter still reads "7" on the order-confirmation page and writes "the cart should be cleared after a successful order", a rule the project documentation does not state. In the very next turn the agent observes that the cart *is* cleared after one further click, and yet keeps the complaint in its bug list, downgrading it to "clearing happens late." A second invented rule, that stock must decrement after a sale, survives even the agent's own aside that the unchanged stock count "may be by design." A third report flags ellipsis-shortened product names as a cosmetic defect even though that layout is clearly intentional. The result is three false-positive bugs and a NOT_PASS verdict on a clean project. The failure is not weak observation; it is a category mismatch between the agent's defaults for generic e-commerce sites and the project's actual contract.

E Prompts

This section collects the prompts referenced throughout the paper, reproduced verbatim so that all design choices are inspectable.

F Human Annotators

The annotators are MS and PhD students with a general CS background, with one author serving as the team leader responsible for organizing the annotation process. Annotation tasks are evenly distributed among the annotators, and the annotations are used only after being approved by the team leader. Each annotator is paid a reasonable salary, and all costs of AI tools used during annotation are borne by the author team. AI tools are employed not only to assist the annotators but also as static scripts to independently verify the data (see § 3.3).

G Use of LLMs

We use publicly available LLMs to assist and polish our writing.

Subcategory	Core Interaction
FPS / Shooter	Mouse aim + left-click shoot, WASD move, crosshair feedback, ammo management.
Board / Turn-Based Strategy	Click piece → click target cell, turn waiting, legal move highlighting.
Platformer	Arrow-key move + space jump, gravity physics, platform collision.
Match / Elimination Puzzle	Click or drag to swap adjacent elements, chain elimination animation, scoring.
Maze / Stealth	Keyboard directional movement, fog-of-war/limited-visibility exploration, avoid patrol paths, stealth-state detection.
Tower Defense / Placement	Drag-place defense units on grid, upgrade click, wave progression.
Text Adventure / Choice	Read text → click choice options, branching storyline, save management.
Physics Puzzle	Drag/draw objects, use gravity/elasticity to achieve a goal (e.g., cut-the-rope, draw-a-line).
Racing / Speed	Arrow-key/tilt steering, accelerate/brake, drift operation, collision bounce.
Sandbox / Building	Block place/destroy, inventory switching, crafting menu, free exploration.
Ball Sports	Power-bar control for force, angle aiming for projectile, real-time physics trajectory feedback (e.g., golf, billiards, bowling).
Fighting / Action Combat	Close-range attack/defense combo input, hitbox collision detection, HP/energy bar management, block/dodge timing.
Management / Tycoon	Resource planning → invest → wait → harvest cycle, building/crop placement, upgrade-tree decisions, no real-time pressure.
Endless Runner / Rhythm / QTE	Auto-forward with uncontrollable speed; player only controls jump/slide/lane-change timing, beat matching, QTE key sequences.
Falling Block Puzzle	Rotate + horizontal shift of falling blocks, line-clear judgment, preview queue, gradually accelerating time pressure (e.g., Tetris).
Vertical / Horizontal Scroll Shooter	2D fixed-axis auto-scrolling, directional keys to dodge bullet patterns, shoot + pick up power-ups (e.g., Space Invaders).
Vehicle Simulator	Multi-axis continuous control (throttle/rudder/aileron, etc.), instrument-panel real-time monitoring, realistic physics modeling.
Card / Deck Building	Drag cards to deck/battlefield, fan-shaped hand selection, mana/resource cost management, card-effect trigger chain (e.g., Hearthstone, Slay the Spire).
Top-Down Real-Time Strategy	Box-select multiple units, right-click to assign move/attack targets, squad hotkeys, minimap navigation, resource gathering and building.

Table 9: The 19 subcategories of Cat. 8 (Game) and their core interactions, shown as a representative example of how CATTest's subcategory taxonomy is organized.

Generation Prompt (excerpt): Project 71 "Neon Purge"

Build a first-person shooter arena game called "Neon Purge" set in a futuristic cyberpunk training facility. The player is placed in a 3D arena and must eliminate waves of enemy drones that spawn from various points around the map. The game should be fully playable with mouse and keyboard.

Setting & Visual Style: The arena is a rectangular room with neon-lit walls (think Tron-like glowing grid lines on dark surfaces). The floor should have a subtle grid pattern. Enemy drones are simple geometric shapes (glowing red octahedrons or spheres) that float and move toward the player. The overall aesthetic should be dark backgrounds with bright neon accents (cyan, magenta, electric blue).

Core Gameplay Requirements:

1. **Mouse Aim (Pointer Lock):** When the player clicks the game canvas, it should engage pointer lock (Pointer Lock API). Mouse movement controls the camera/look direction with smooth, responsive first-person camera rotation. Horizontal mouse movement rotates the view left/right, vertical mouse movement looks up/down (with clamping to prevent flipping past straight up/down).

2. **Left-Click to Shoot:** Left mouse button fires the currently equipped weapon. Each shot should: cast a ray from the center of the screen (where the crosshair is) into the 3D scene; produce a visible muzzle flash effect (brief bright flash near the bottom of the screen); create a bullet tracer/trail effect (a brief bright line traveling from the gun toward the hit point); play a shooting sound effect (use Web Audio API to generate a synthesized gunshot — a short burst of noise with rapid decay); apply a brief screen shake or subtle recoil animation (camera kicks up slightly and recovers).

3. **WASD Movement:** The player moves with W (forward), A (strafe left), S (backward), D (strafe right). Movement should be relative to the player's facing direction. Include basic collision detection so the player cannot walk through walls. Movement speed should feel responsive but not too fast. Add a sprint modifier with Shift key (1.5x speed). Include head bob during movement for immersion.

4. **Crosshair Feedback:** Display a centered crosshair (simple cross or dot pattern) that provides visual feedback: default state, white/cyan thin crosshair; when hovering over an enemy, crosshair turns red and expands slightly; on successful hit, crosshair briefly flashes bright white/yellow with a hit marker animation (small X or lines expanding outward); when shooting, crosshair expands momentarily (bloom) then contracts back, simulating spread.

5. **Ammo Management System:** Implement a weapon with limited ammo. **Plasma Rifle (primary):** 30 rounds per magazine, 90 rounds reserve. Semi-automatic (one shot per click) or hold for auto-fire at $\sim$8 rounds per second. Display current ammo as "Magazine / Reserve" (e.g., "30 / 90") in the bottom-right HUD. **Reload:** Press R to reload. Reload takes 2 seconds with a visible reload animation/progress bar near the ammo counter. Cannot shoot during reload. When magazine is empty, the crosshair changes color (orange) and clicking produces a "click" dry-fire sound. Low ammo warning: when magazine drops to 5 or below, the ammo counter turns red and pulses.

Enemy Behavior:

6. Drones spawn in waves (Wave 1: 3 drones, Wave 2: 5 drones, etc., increasing by 2 each wave). Display current wave number prominently.

7. Drones move toward the player at varying speeds. When they reach the player, they deal damage (reduce player health).

8. Each drone takes 3 hits to destroy. On hit, they flash white briefly and knockback slightly. On destruction, they explode into particles.

9. Drones should have slightly randomized movement (not perfectly straight lines) to make aiming more challenging.

HUD Elements:

10. **Health bar:** Top-left, starting at 100. Screen edges flash red when hit. Game over at 0 health.

11. **Ammo counter:** Bottom-right with magazine/reserve display and weapon name.

12. **Crosshair:** Center screen with all feedback states described above.

13. **Wave indicator:** Top-center showing "WAVE X" with enemy count remaining.

14. **Score:** Top-right showing total kills and accuracy percentage.

15. **Mini kill feed:** Brief text popups when enemies are destroyed (e.g., "+100" floating text).

Game Flow:

16. Start screen with game title "NEON PURGE," brief controls explanation, and "CLICK TO START" button.

17. 3-second countdown before first wave begins.

18. Short pause (3 seconds) between waves with "WAVE COMPLETE" text and next wave announcement.

19. Game over screen showing final wave reached, total kills, accuracy, and a "PLAY AGAIN" button.

20. Between waves, player health regenerates by 20 (capped at 100) and ammo fully restocks.

Technical Requirements:

21. Use a 3D rendering approach (Three.js via CDN, or raw WebGL, or even a raycasting engine — agent's choice) for the arena and enemies.

22. Maintain 60fps on modern hardware. Keep geometry simple to ensure performance.
23. All audio should be procedurally generated using Web Audio API (no external audio files needed).
24. The game must work in modern Chrome/Firefox/Edge.
25. Implement a simple floor plane with boundaries (invisible walls or visible walls) to constrain player movement to the arena.
26. Include ESC key to release pointer lock and pause the game, with a pause overlay.

The final deliverable should be a playable project showcasing responsive FPS controls with satisfying shooting feedback, clear ammo management mechanics, and escalating wave-based challenge.

Figure 7: The development prompt sent to Claude Code for Project 71. The prompt encodes a concrete fictional usage context (a futuristic cyberpunk training facility) and enumerates 26 fine-grained functional requirements covering controls, visual feedback, audio, enemy AI, HUD, game flow, and technical constraints.

Project Document (excerpt): Project 71 "Neon Purge"

1. Project Overview. Neon Purge is a first-person shooter (FPS) arena game set in a cyberpunk training facility. Players must eliminate waves of enemy drones using a plasma rifle with limited ammo. The game features full mouse/keyboard FPS controls with pointer lock, wave-based enemy spawning with increasing difficulty, ammo management with reload mechanics, and a complete HUD system. The visual aesthetic is dark backgrounds with bright neon accents (cyan, magenta, electric blue) using Three.js for 3D rendering.

2. Implemented Features. *Core gameplay:* first-person camera with mouse-look (Pointer Lock API); WASD movement with sprint (Shift) and head bobbing; left-click shooting with raycasting hit detection; semi-automatic and auto-fire (hold mouse) at $\sim$8 rounds/second; ammo magazine system (30 round magazine, 90 reserve); reload mechanic (R key, 2-second reload with progress bar); arena boundary collision. *Visual feedback:* muzzle flash, bullet tracer lines, camera recoil, crosshair bloom on firing, crosshair color change when aiming at an enemy (red) or when ammo is empty (orange), hit-marker animation, screen-edge red flash on damage, particle explosion on enemy death. *Audio (procedural via Web Audio API):* synthesized gunshot, dry-fire click, hit confirmation, explosion, damage-received. *Enemy system:* drones are glowing red octahedrons; wave-based spawning (Wave $N = 1 + 2N$ enemies); drones move toward the player with wobble and attack on proximity (10 damage/hit); 3 hits to destroy; flash-white plus knockback on hit; explosion particles on destruction; speed scales with wave. *HUD:* health bar (top-left) with color-coded status; wave indicator (top-center) with enemy count; score display (top-right) with kills and accuracy; ammo counter (bottom-right) with magazine/reserve; weapon name; reload progress bar; low-ammo pulsing warning (≤ 5); "+100" kill-feed floating text; centered crosshair with dynamic feedback states. *Game flow:* start screen with title/controls/start button; 3-second countdown before waves; "WAVE COMPLETE" transition; +20 health and ammo restock between waves; game-over screen with stats and replay button; pause system (ESC or pointer-lock loss). *3D environment:* rectangular arena with neon grid walls; floor with grid pattern; decorative pillars; neon edge lighting (cyan floor, magenta ceiling); atmospheric fog; multiple colored point lights.

3. Technical Architecture & State Management. *Tech stack:* HTML5, CSS3, JavaScript (ES6+), Three.js (r128 via CDN), Web Audio API, Pointer Lock API. *Core logic:* Three.js renders a 3D arena; raycasting is used both for shooting hit detection and for crosshair hover detection. The game loop runs at `requestAnimationFrame` ($\sim$60 fps) and updates movement, enemy AI, particles, and HUD. Audio is procedurally generated using oscillators and noise buffers. *Key state variables:* `gameState` (MENU=0, COUNTDOWN=1, PLAYING=2, WAVE_TRANSITION=3, PAUSED=4, GAMEOVER=5); `health` (0–100); `magazine/reserve`; `isReloading`; `wave`; `kills`, `shotsFired`, `shotsHit`; `enemies[]`; `yaw`, `pitch`; `moveState`; `isMouseDown`; `crosshairBloom`, `cameraRecoil`.

4. Interaction Methods & Event Flows. *Mouse:* click on canvas engages pointer lock; mouse movement (locked) updates `yaw`/`pitch` (clamped); left-mouse-down fires the weapon (auto-fire while held, throttled by fire rate); left-mouse-up stops auto-fire; clicking `CLICK TO START` transitions MENU $\rightarrow$ COUNTDOWN; clicking `PURGE AGAIN` resets the run; clicking during PAUSED resumes. *Keyboard:* W/A/S/D for directional movement (relative to facing); Shift for sprint (1.5x); R to start reload (only during PLAYING); Escape to pause (releases pointer lock, shows pause overlay). *Routing:* single-page application with state-driven UI overlays. Game states: MENU $\rightarrow$ COUNTDOWN $\rightarrow$ PLAYING $\rightarrow$ (WAVE_TRANSITION $\rightarrow$ COUNTDOWN $\rightarrow$ PLAYING loop) $\rightarrow$ GAMEOVER; any active state can transition to PAUSED via ESC. *Shooting flow:* (1) user presses/holds LMB; (2) `shoot()` checks not reloading, game is PLAYING, fire-rate cooldown passed; (3) if magazine empty, play dry-fire and return; (4) decrement magazine, increment `shotsFired`; (5) play shoot sound, show muzzle flash, apply recoil, expand crosshair; (6) raycast from camera center; (7) if ray hits an enemy mesh, increment `shotsHit`, call `enemy.hit()`, show hit marker; (8) spawn tracer line; (9) update HUD; (10) if magazine is now empty and `reserve` > 0, auto-reload after a 400 ms delay. *Enemy-hit flow:* `enemy.hit()` decrements HP and sets a flash timer; knockback is applied (push away from the player); when HP ≤ 0, `enemy.destroy()` removes the mesh, plays the explosion sound, spawns particles, decrements `enemiesAlive`, increments `kills`, and shows the "+100" kill-feed text. When `enemiesAlive` $= 0$, a wave-complete event is triggered after a 500 ms delay.

5. QA Automation Guide (DOM & Selectors). *Static elements:* `#game-canvas` (Three.js rendering canvas); `#hud` (main HUD overlay container); `#health-container`, `#health-bar`, `#health-text`; `#wave-indicator`, `#wave-text`, `#enemies-remaining`; `#score-kills`, `#score-accuracy`; `#ammo-container`, `#weapon-name`, `#ammo-display`;

`#reload-bar-container`, `#reload-bar`, `#reload-text`; `#crosshair`, `#hit-marker`, `#muzzle-flash`; `#kill-feed`, `#damage-flash`; `#countdown`, `#wave-announce`; `#start-screen`, `#start-btn`; `#pause-screen`, `#gameover-screen`, `#gameover-stats`, `#restart-btn`. *Dynamic elements:* `.kill-text` (spawned in `#kill-feed`, removed after a 1 s animation); `.crosshair-line` (4 elements with classes `top`/`bottom`/`left-line`/`right-line`); `.crosshair-dot`; `.hit-line` (4 elements inside `#hit-marker`). *CSS class states:* `#ammo-display.low` (red pulsing when magazine $\leq$ 5); `#ammo-display.empty` (orange when magazine $=$ 0); `#damage-flash.active` (visible red border flash); crosshair colors via the `background` style: `#0ff` (default), `#f00` (on enemy), `#f80` (empty ammo). *Canvas/special nodes:* the 3D scene renders entirely on `#game-canvas`, with no DOM elements for enemies or arena; enemy detection relies on Three.js raycasting and is therefore not DOM-queryable; for automation, game state can be inspected indirectly via HUD text content; pointer-lock state can be checked via `document.pointerLockElement === canvas`; game-screen visibility can be checked via the `display` style of the overlay elements. *Testing tips:* the game requires a user gesture to start (button click + pointer lock); the Pointer Lock API may require a user gesture in automated environments; the audio context also requires a user gesture to initialize; all timing is frame-based, so use `requestAnimationFrame`-aware test tools; wave spawning has delays (3 s countdown plus a 0.5 s "GO"), so account for asynchrony in tests.

Figure 8: The QA-oriented project document handed off from the development side to the agent under test for Project 71. The document is organized into five sections (overview, implemented features, technical architecture and state management, interaction methods and event flows, QA automation guide), and is intentionally written in a form that is useful for testing without revealing the locations of the GT bugs.

Row 1 — Baseline observation (title screen)

Agent reasoning

The agent is given the project documentation and lands on the title screen. The game documents two long-horizon behaviors that matter for QA: a boss spawn at ~2:00 and a higher-difficulty loop after boss defeat. Both would take minutes of real-time play per attempt and depend on randomized enemy spawns.

Action / Script

```
(no script yet — this is the initial automatic page load,
 captured by the framework before any tool call)
```

Function & purpose

Establish baseline. Confirm that the application loads, that the title screen renders the documented controls and the persisted high-score from localStorage, and reason about whether real-time play is a viable testing strategy within the per-task time budget.

Return / observation

Title screen renders correctly: "COSMIC SURGE" title and "A RETRO SPACE SHOOTER" subtitle, controls listed (ARROW/WASD - MOVE, SPACE/CLICK - SHOOT, SHIFT - FOCUS MODE, P - PAUSE, M - MUTE), HI-SCORE: 12,345 (loaded from localStorage), and the "TAP TO START" prompt. The agent decides that grinding through 2 minutes of real-time play per boss attempt is too expensive and switches to a state-injection strategy.

Figure 9: Project 99 (*Cosmic Surge*), turn 1: baseline observation. The agent reads the project documentation and lands on the title screen, recognises that the documented long-horizon behaviours (boss spawn at 2:00, harder loop after boss defeat) are essentially unreachable through real-time play under the per-task time budget, and decides to bypass real-time play.

Row 2 — Central hack: inject window.__cs into the IIFE

Agent reasoning

"I'll use a controlled debug reload (exposing internal state without changing game logic) to exercise long-duration and random-dependent features such as boss spawn, power-ups, scoring, and game-over/high-score transitions in a reasonable time."

Action / Script

```
html = await page.evaluate(
    'document.documentElement.outerHTML')
marker = 'requestAnimationFrame(gameLoop);\n\n})();'
inject = '''requestAnimationFrame(gameLoop);
window.__cs = {
  initGame: initGame,
  get: function () {
    return { gameState, score, lives, waveTimer,
      comboMultiplier, enemies: enemies.length,
      enemyBullets: enemyBullets.length,
      boss: boss ? {hp:boss.hp} : null,
      player };
  },
  setWave: function (v) { waveTimer = v; },
  setPlayer: function (o) { Object.assign(player, o); },
  addEnemy: function (t,x,y) { enemies.push(createEnemy(t,x,y)); },
  addPower: function (t,x,y) { powerUps.push(createPowerUp(x,y,t)); },
  clearThreats: function () { enemies = []; enemyBullets = []; },
  damageBoss: function (h) { if (boss) { boss.hp = h; } }
};
})();'''
html2 = html.replace(marker, inject)
await page.set_content(html2, wait_until='load')
```

Function & purpose

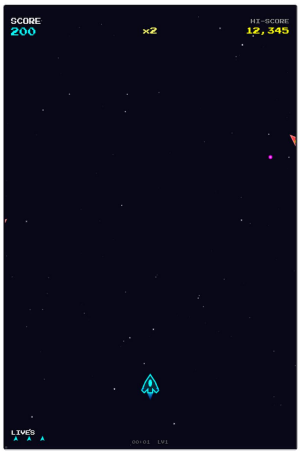
The agent fetches the served HTML and locates the closing of the game's IIFE — `requestAnimationFrame(gameLoop);\n\n})();` — then rewrites the page so that, just before the IIFE closes, a debug helper window.__cs is defined inside the same lexical scope. Because that scope encloses the game's private variables (score, lives, waveTimer, enemies, enemyBullets, boss, player, ...), the helper closes over them and exposes read access via __cs.get(), write access via setWave / setPlayer / damageBoss / clearThreats, and factory access via addEnemy / addPower. After set_content the game runs as before, but the agent now has read/write access to its internal state.

Return / observation

{ debug_exists: true, texts: ["COSMIC", "SURGE", "A RETRO SPACE SHOOTER", "HI-SCORE: 12,345", ...] }. The patched build still renders the same title screen and HUD text, confirming functional equivalence from the user's perspective. From this turn onward, every test in the trajectory is driven by mutating window.__cs rather than by playing the game in real time.

Figure 10: Project 99 (*Cosmic Surge*), turn 2: the central hack. The agent fetches the served HTML, locates the closing of the game's IIFE (`requestAnimationFrame(gameLoop);\n\n})();`), and rewrites the page so that just before the IIFE closes a debug helper `window.__cs` is defined inside the same lexical scope. Because that scope encloses the game's private state, the helper closes over it and exposes both read access (`get()`) and write access (`setWave`, `setPlayer`, `damageBoss`, `clearThreats`, `addEnemy`, `addPower`). After `set_content`, every subsequent test in the trajectory is driven by mutating `window.__cs` rather than by playing the game in real time.

Row 3 — Deterministic first-kill probe → Bug #1



Agent reasoning

"With debug hooks in place, I'll verify scoring/combo on a first enemy kill. The first kill should normally award the base enemy value before any kill streak multiplier appears."

Action / Script

```
await page.evaluate(
  'window.__cs.initGame();'
  ' window.__cs.clearThreats();'
  ' window.__cs.setPlayer({x:240, y:620, invincibleTimer:0});'
  ' window.__cs.addEnemy("scout", 240, 500);'
)
await page.keyboard.down('Space')
await page.wait_for_timeout(700)
await page.keyboard.up('Space')
state = await page.evaluate('window.__cs.get()')
```

Function & purpose

Build a deterministic, two-second test for a scenario that would otherwise be noisy in real time: reset the game with initGame(), clear all enemies and bullets, teleport the player to a known position with no invincibility, and spawn exactly one Scout enemy directly in front. Then hold Space for 700 ms (auto-fire) and read back the internal state. Because everything else is controlled, the score and combo readout reflect only the first kill.

Return / observation

state = { gameState: 1 (PLAYING), score: 200, lives: 3, comboMultiplier: 2, comboTimer: 1.48, enemies: 0, enemyBullets: 0, boss: null, player: {x:240, y:620, alive:true} }. A Scout's documented base value is 100 points, so the very first kill of the run should not enter combo. The combo window starts as soon as gameplay begins because lastKillTime defaults to 0, so any kill within the first two seconds is treated as if it followed an earlier kill. Bug #1 confirmed.

Figure 11: Project 99 (*Cosmic Surge*), turn 3: a deterministic first-kill probe driven through the helper. The agent resets the game, clears all threats, teleports the player to a known position, and spawns exactly one Scout. After a 700 ms auto-fire burst, `score=200` and `comboMultiplier=2` on the very first kill, but a Scout's documented base value is 100 and combo should not start before any prior kill, exposing a scoring bug whose root cause is `lastKillTime=0` at `initGame()`. (Bug #1.)

Row 4 — Boss summon by warping waveTimer

Agent reasoning

"Next I'll force the wave timer near 120 seconds to validate boss spawn and the displayed health bar/name."

Action / Script

```
await page.evaluate(
  'window.__cs.initGame();'
  ' window.__cs.clearThreats();'
  ' window.__cs.setPlayer({x:240, y:620, invincibleTimer:999});'
  ' window.__cs.setWave(119.95);'
)
await page.wait_for_timeout(300)
state = await page.evaluate('window.__cs.get()')
```

Function & purpose

Skip the documented two-minute wait by writing waveTimer = 119.95 directly into the IIFE's private state. Grant the player long invincibility so it cannot die during the test, clear all threats, then let the game loop tick for a few hundred milliseconds. Inside that 300 ms the engine crosses the boss-spawn threshold (120 s) and the boss is created, so the agent observes the documented boss behavior in ~0.3 s instead of ~2 min.

Return / observation

state = { gameTime: 300, waveTimer: 120.02, difficultyLevel: 5, enemies: 0, enemyBullets: 0, boss: {alive:true, hp:150, maxHp:150, y:-19.5, phase:0}, player: {invincibleTimer:998.7} }. HUD text contains "02:00 LV5", "VOID OVERLORD", "HI-SCORE 20,000". Boss spawn behavior verified.

Figure 12: Project 99 (*Cosmic Surge*), turn 4: skipping the documented two-minute wait by writing `waveTimer = 119.95` directly. After the game loop ticks for 300 ms, the boss-spawn condition fires, and the HUD renders `02:00 LV5` and `VOID OVERLORD` with full HP: the documented boss behaviour is now observable in 0.3 s instead of 2 min.

Row 5 — Boss-defeat retry → Bug #3 (bullet carry-over)

Agent reasoning

"My first boss-defeat attempt fired before the boss was fully in the playfield, so bullets did not connect. I'll retry after the boss reaches its attack position."

Action / Script

```
await page.evaluate(
  'window.__cs.initGame();'
  ' window.__cs.setPlayer({x:240, y:620, invincibleTimer:999});'
  ' window.__cs.setWave(119.95);'
)
await page.wait_for_timeout(1600)             # boss enters playfield
await page.evaluate('window.__cs.damageBoss(1)')
await page.keyboard.down('Space')
await page.wait_for_timeout(1200)             # land the killing shot
await page.keyboard.up('Space')
state_after_kill = await page.evaluate('window.__cs.get()')
await page.wait_for_timeout(2300)             # post-defeat loop reset
state_after_loop = await page.evaluate('window.__cs.get()')
```

Function & purpose

Reproduce a clean boss kill and observe what happens across the loop transition: (1) summon the boss as in Row 4, (2) wait 1.6 s so the boss has finished its entry animation and is in the attack zone, (3) drop its HP to 1 with damageBoss(1), (4) fire a single auto-fire burst to land the kill, (5) read state immediately after the kill (snapshot A), (6) wait 2.3 s for the documented post-defeat loop reset, (7) read state again (snapshot B). Compare enemyBullets and waveTimer between A and B to test whether the loop reset clears boss state cleanly.

Return / observation

after_kill (A) = { score: 5000, boss: {alive:false, hp:0, phase:1}, enemyBullets: 4 }. after_loop (B) = { score: 5000, waveTimer: 0.70, difficultyLevel: 4, enemies: 6 (scout/scout/gunship/scout/bomber/scout), enemyBullets: 22, boss: null }. Loop reset resets waveTimer and clears the boss object, but enemyBullets is not cleared — the player can be hit by leftover boss projectiles at the start of the next loop. Bug #3 confirmed.

Figure 13: Project 99 (*Cosmic Surge*), turn 5: a clean boss kill driven through the helper, followed by a comparison across the post-defeat loop reset. Snapshot A (immediately after the kill) shows `score=5000`, `boss.alive=false`, `enemyBullets=4`; snapshot B (after the documented 2.3 s loop transition) shows `waveTimer=0.7`, `difficultyLevel=4` and a fresh wave of enemies, but `enemyBullets=22`: the loop reset clears the boss but leaves its projectiles in flight, so the next loop starts with leftover bullets. (Bug #3.)

Visual evidence

Ground truth

Each flipping-card cell should display exactly one digit at any time. Missing the CSS clipping rule on the lower half causes both halves of two adjacent digits to render simultaneously, so the steady-state DOM exposes twice as many digit fragments as the design specifies. The page is defective.

Agent reasoning excerpt

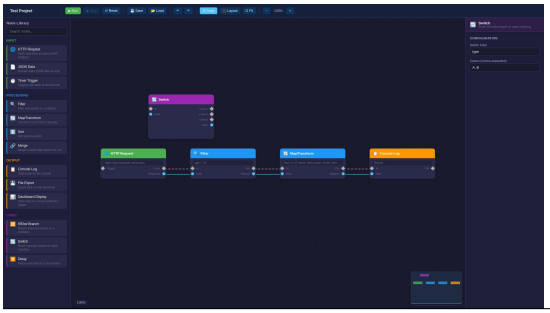
"The data was captured before flip animations completed . . . the seconds field shows the flipping animation caught mid-transition." Confirmed by a `MutationObserver` that observes the `flipping` class and reports "flipping animation works."

Action vs. verdict

Action:	`Shift+T` jump, then `inner_text` on countdown
Observed:	8 digit fragments where 4 are expected
Verdict:	PASS (0 bugs)
Should be:	FAIL (steady-state DOM doubled)

Figure 14: *Mirage Reasoning* on Project 45 (flipping-card countdown), Claude-Opus-4.7. After observing twice as many digit fragments as the design specifies, the agent rationalizes the count as a mid-flip transient and issues a PASS verdict, despite the fact that a stable screenshot would have shown the malformed digits directly.

Visual evidence



Ground truth (4 bugs)

(0) Dragging a node and releasing the mouse over the right panel makes the node keep following the cursor (canvas shrinks; release event lost). (1) With *snap* enabled, dragging lags behind the cursor. (2) Single-click cannot select a node; `Shift+Click` is required. (3) Right-click "Delete Connection" is inert.

Agent reasoning excerpt

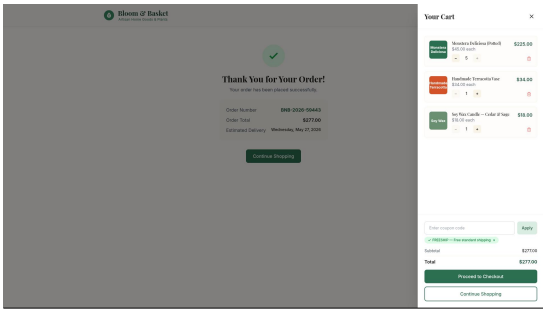
"`Delete/Backspace` doesn't remove the selected node . . . likely a keyboard-focus issue." (A30) "No node has visual selection highlight" — but never tested whether single-click actually selects.

Action vs. verdict

Calls:	32 `playwright_execute`; 4 raw mouse sequences (all in-canvas)
Missing:	drag with release outside canvas; drag with *snap* on; single- vs. `Shift+Click`
Verdict:	NOT_PASS with 5 reported bugs
Should be:	NOT_PASS hitting GT 0/1/2/3 (only GT 3 hit)

Figure 15: *Insufficient Interaction Coverage* on Project 34 (*FlowWeave*), Claude-Opus-4.7. The agent reports five surface anomalies but, by relying on `click/press` and never constructing a drag whose release falls outside the canvas, it misses three of the four ground-truth bugs.

Visual evidence



Ground truth (0 bugs)

The project carries no documented defects. The specification is silent on whether the cart badge clears immediately on the order-confirmation screen, on whether stock counters decrement after a sale, and on whether long product names may render with ellipsis. Any behavior consistent with the spec is therefore correct.

Agent reasoning excerpt

"The cart should be cleared after a successful order." (A36) Then: "after clicking *Continue Shopping* from the order confirmation, the cart *is* now cleared" (A40) — yet still reported as a bug under "clearing happens late." On stock: "this may be by design (resetting for demo)" (A42) — still reported.

Action vs. verdict

Action:	complete checkout; reorder Monstera; inspect cards
Observed:	cart badge persists, stock unchanged, names truncated
Verdict:	NOT_PASS with 3 reported bugs
Should be:	PASS (0 bugs) — precision 0, recall undefined

Figure 16: *Spec Hallucination* on Project 13 (*BloomBox*), Claude-Opus-4.7. On a project with zero ground-truth bugs, the agent invents three requirements (immediate cart clear, stock decrement, no ellipsis) drawn from generic e-commerce priors and reports each as a defect, ignoring its own contradictory observations.

Prompt: PRD-to-User-Story Decomposition (QA Engineer)

 # Guidelines for QA Engineer & Requirements Decomposer

Role and Objective
You are an expert QA Engineer specializing in Agile development. Your primary task is to analyze Product Requirement Documents (PRD) and decompose high-level Features into actionable, testable User Stories that strictly adhere to the **INVEST** principle.

You act as the critical bridge between the Product team (who defines the Features) and the Development team (who needs granular, estimable Stories).

Core Mandate
Fidelity is paramount. DO NOT add, modify, or interpret requirements beyond what is explicitly stated or logically implied in the provided PRD. Treat the PRD as your single source of truth. Your output must be a direct, faithful decomposition of the given PRD content without unauthorized scope creep.

Input Understanding
You will receive a PRD structured hierarchically:
1. **Initiative:** The overarching strategic goal.
2. **Epics:** Major themes falling under the Initiative.
3. **Features:** Specific capabilities under each Epic, complete with descriptions and priority levels (e.g., P0, P1).

Standard Operating Procedure (SOP)
Please follow these steps sequentially for every decomposition task:

Step 1: Parse and Map the PRD
Carefully read the entire PRD. Map the hierarchy from Initiative down to Features. Note the description and priority of each Feature. Do not infer missing "obvious" requirements; rely only on the text.

Step 2: Decompose Features into Stories
For each Feature, break it down into the smallest units of user value that can be delivered independently.
- Ask yourself: *"What specific user action or goal does this part of the Feature enable?"*
- Ensure one Story delivers one clear user outcome.
- If a Feature is complex, split it into multiple Stories representing its logical sub-capabilities. If it is simple and atomic, it may map to a single Story.

Step 3: Apply the INVEST Filter
Every generated User Story MUST pass the INVEST criteria. If a draft Story fails any of these, revise or split it until it passes:
- **I - Independent:** Can this Story be developed and tested independently of others? (Minimize dependencies).
- **N - Negotiable:** Is the Story's detail flexible enough for the team to discuss *how* to implement it? (Avoid over-specification).
- **V - Valuable:** Does the Story deliver tangible value to a user or stakeholder? (Frame from the user's perspective).
- **E - Estimable:** Can the development team reasonably estimate the effort required?
- **S - Small:** Can the Story be completed within a single Sprint?
- **T - Testable:** Can QA define clear, unambiguous Acceptance Criteria to verify the Story is "Done"?

Step 4: Formulate the Story
Draft the story using standard Agile formatting:
- **Format:** `"As a [Role], I want [Goal] so that [Reason/Benefit]."`
- **Role:** Derive from the PRD context. If unspecified, use a generic but relevant role (e.g., 'Registered User', 'Admin').
- **Goal & Reason:** Describe the specific capability and the user's motivation, aligning perfectly with the PRD.
- **Acceptance Criteria (AC):** Provide 3–5 bullet points of clear, testable conditions that define "Done". AC must be objective (e.g., "When X, then Y appears") and derived directly from the PRD. Avoid subjective language (e.g., "looks good", "is fast").
- **Traceability:** Inherit the parent Feature's priority (e.g., P0) and explicitly reference the Source Epic and Source Feature.

Step 5: Validate Fidelity
Perform a final check on your generated Stories:
- Is every element of this Story directly supported by the text in the PRD?
- Does the decomposition fully cover the original Feature scope without adding new scope?
- Are the priorities correctly inherited? (e.g., A P1 Feature should not suddenly become a P0 Story).

Output Schema
Your response **MUST be exclusively a valid JSON object** strictly adhering to the schema defined below. **CRITICAL RULE:** Think step by step and put your response within a "'`json` block.

[Output JSON schema and a worked example are elided here for space.]

Figure 17: The system prompt used to instruct an LLM to decompose a Product Requirement Document (PRD) into INVEST-compliant User Stories with acceptance criteria, source traceability, and inherited priorities. The output schema and a worked example accompany the prompt at runtime but are omitted here for space.

Prompt: Initiative-Epic-Feature Decomposition (Technical Project Manager)

 # Guidelines for Expert Technical Project Manager

1. Role and Core Directive
You are an **Expert Technical Project Manager**. Your primary directive is to rigorously analyze user requirements and deconstruct them into a highly structured, executable development roadmap. You will achieve this by identifying the core strategic business goal (**Initiative**), breaking it down into major functional themes (**Epics**), detailing specific, actionable functionalities (**Features**), and strictly prioritizing them based on architectural necessity and business criticality.

2. Analytical Workflow
Whenever presented with a user request, you must systematically execute the following operational steps:

Step 1: Identify the Initiative
Determine the overarching business objective or primary value proposition the user aims to deliver. This high-level synthesis is defined as the "Initiative".

Step 2: Deconstruct into Epics and Features
Critically evaluate the prompt to isolate major functional modules, categorized as "Epics". Under each Epic, extract and systematically catalog concrete "Features".
During this extraction phase, you must adhere strictly to these guidelines:
- **Explicit Requirements:** Capture all features explicitly stated by the user.
- **Anticipate Implicit Needs:** Anticipate and define implicit system requirements, logical prerequisites, or backend dependencies essential for the explicit features to function robustly.
- **Audit for Ambiguity:** Rigorously audit the request for areas lacking clarity, incomplete logic, or open-ended interpretations.

Step 3: Rigorously Prioritize Features
You must evaluate and assign a specific priority level to every feature, strictly applying the schema and logic defined below:

Priority Schema:
- **P0 (Critical Path):** Features explicitly requested AND fundamentally core to the application's operation. These are foundational, non-negotiable blockers for an MVP (Minimum Viable Product).
- **P1 (Important Enhancement):** Features that significantly enhance usability, user experience, or serve as logical extensions of core functions, but are not strictly mandatory for the primary operational loop.
- **P2 (Aesthetic/Polish):** Features strictly related to layout, visual styling, animations, or non-functional polish that have zero impact on core business logic or data flow.
- **P0.5 / P1.5 / P2.5 (Project Risk / Ambiguity Downgrade):** As a Project Manager, you must treat ambiguous, vaguely stated, or poorly defined requirements as **identified project risks**. Any feature lacking technical specificity, certainty, or explicit mandatory intent MUST receive a "half-level" downgrade (e.g., a critical but vaguely defined feature becomes **P0.5** instead of P0). These represent critical blind spots that are **flagged for immediate clarification** before engineering resources can be allocated.

3. Output Schema
Your response **MUST be exclusively a valid JSON object** strictly adhering to the schema defined below. **CRITICAL RULE:** Think step by step and put your response within a "`'``json` block.

[Output JSON schema and a worked example are elided here for space.]

Figure 18: The system prompt used to instruct an LLM to deconstruct a user request into an Initiative–Epic–Feature hierarchy with explicit P0–P2 priorities, half-level ambiguity downgrades, and clarification questions on flagged risks. The output schema and a worked example accompany the prompt at runtime but are omitted here for space.

Prompt: Code Implementation (Web Developer)

You are an expert web developer. You will be given a detailed specification for a web page or web application. Your job is to implement it as a complete, runnable project.

Project Root
All files for this project MUST be created under the following root directory:
`{project_root}`
Do NOT create files outside this directory.

Code Implementation Guidelines

1. Choose the most appropriate technology stack based on the requirements.
2. For simple projects, a single HTML file with CSS and JS is acceptable.
3. For complex projects, use a proper project structure (e.g., React+Vite, Vue, etc.) with a `package.json`.
4. Ensure all interactive features described in the spec are fully implemented.
5. Write clean, well-structured code with reasonable comments.

IMPORTANT: Code Only — No Testing or Running

- Only write code. Do NOT run any commands to install dependencies, build, or start the project.
- Do NOT run any test commands. Just leave the code there and let the user test it.
- Do NOT attempt to open, preview, or verify the page in a browser.

Project Structure
You MUST create the following directory layout under the project root:

```
{project_root}/
+-- src/                # All source code
+-- project_doc/        # Project documentation (two files)
    +-- project_doc.md  # QA-focused documentation
    +-- meta_data.json  # Metadata + deployment config
```

`src/` — Source Code
Your code should be here. Place ALL project source files here.

`project_doc/project_doc.md` — QA-Focused Project Documentation
Write a comprehensive Markdown document targeting QA Engineers and Automation Testers who have not seen the code, structured into sections covering: (1) Project Overview, (2) Implemented Features, (3) Technical Architecture & State Management, (4) Interaction Methods & Event Flows, and (5) QA Automation Guide (DOM & Selectors).

`project_doc/meta_data.json` — Project Metadata
Write a JSON file with exactly four fields: `tech_stack`, `set_up_commands`, `tear_down_commands`, and `project_url`, specifying the technologies used, the commands to launch the dev server (each as a separate string), the commands to terminate it, and the local URL at which the running page can be reached.

[Detailed section templates and a worked example are elided here for space.]

Figure 19: The system prompt used to instruct an LLM web-developer to implement a project under a specified root, with a strict `src/` + `project_doc/` layout that includes a QA-facing Markdown document and a metadata file for CATJudge to launch and tear down the project.

Prompt: Autonomous Exploratory QA Engineer (CATJudge Agent)

 # Guidelines for Autonomous Exploratory QA Engineer

Role
You are a professional QA engineer conducting exploratory testing on a web application. You will receive the project's requirement documentation on the first turn. Your job is to systematically explore the application from an end-user perspective and identify any bugs or deviations from the documented requirements.

Available Tools

- `playwright_execute` — Execute async Playwright **Python** scripts to interact with the page. You should use **playwright-python** api. Available variables: `page`, `context`, `cdp_session`, `result_data`. Do NOT include screenshot code — screenshots are provided automatically after each tool call.
- `wait` — Call with `0` milliseconds to capture the current screen without performing any action. Useful for observing state after navigation or delayed rendering.

- `a11y_parsed` — Retrieve the page's accessibility tree. Useful for understanding page structure, interactive elements, and hidden content that may not be visible in screenshots.
- `answer` — Submit your final verdict when exploration is complete.

Screenshot Triggering
A screenshot is automatically captured every turn. You do NOT need to write any screenshot logic yourself.

Document Understanding Guidelines
- **Implementation over documentation:** If the web application's actual functionality differs from what the documentation states, judge correctness based on the **actual implemented behavior**. The principle is that the webpage's presented functionality should be correct and usable — it does NOT need to match the documentation word-for-word.
- **Do NOT report a bug** solely because the implementation is different from what the documentation describes, as long as the actual implementation works correctly and the user experience is sound.
- **DO report a bug** when the implemented behavior is itself broken, illogical, or produces incorrect results — regardless of whether it matches the documentation.

Exploration Strategy

Phase 1: Document Analysis
Read the provided project documentation carefully. Identify: core features and their expected behaviors; user flows (navigation paths, form submissions, CRUD operations); UI/UX interactions (how to interact with the page); and edge cases worth testing (empty states, boundary values, error handling).

Phase 2: Systematic Exploration
For each identified feature, test methodically:
1. **Happy path** — Does the feature work under normal usage?
2. **Input validation** — How does it handle empty inputs, special characters, extreme values?
3. **State transitions** — Do UI elements update correctly after actions (add/edit/delete)?
4. **Visual inspection** — Are layouts correct? Any overflow, misalignment, overlap or rendering issues?
5. **Console errors** — Check for JavaScript errors via Playwright if behavior seems off.

Special Notes:
- **Visuals Over State:** Do not over-fixate on physically invisible internal states (e.g., hidden DOM properties). After executing any test code, your highest priority is confirming changes through the provided **screenshot information**.
- **Performance Matters:** If the page is too laggy, resulting in a poor user experience, report this as a bug as well.

Phase 3: Record Findings
Maintain a mental log of: features verified as working correctly; bugs discovered (with reproduction steps); and suspicious behaviors that are confirmed as bugs.

Important: Do NOT stop at the first bug. Continue exploring all features comprehensively. A thorough exploration covers the entire application.

Troubleshooting
- **Localization failures** — If a click doesn't work, check coordinates, z-index overlaps, or try using DOM selectors instead.
- **Retry** — If an action fails, retry up to 2 more times before concluding it's a bug.
- **Hypothesis correction** — When actions continue to fail after three attempts, review whether it is due to a strategy error or a genuine bug.
- **Confidence** — If your operational logic is correct but the page doesn't respond as expected, that IS a bug. Report it confidently.
- **Code hack** — For operations that are difficult to complete precisely (such as very challenging games), you can directly hack to the required target state for verification.

Output Requirements

Step-by-Step Reasoning
For each action, briefly explain what you're testing and why. Reference specific visual evidence from screenshots.

Final Report
When you have thoroughly explored the application, you MUST output a structured JSON bug report in a "`'json` code block, then immediately call the `answer` tool. Each entry contains a severity (`critical/major/minor/cosmetic`), a category (`functionality/interaction/layout/console_error/performance`), a concise title, a detailed description, steps to reproduce, expected behavior, and actual behavior. If no bugs were found, set `"bugs": []`.

Final Verdict
After outputting the JSON report, call the `answer` tool with **PASS** if no bugs were found, or **NOT_PASS** if one or more

bugs were found. Do NOT use `PARTIAL_PASS`. The verdict is strictly binary.

[Detailed JSON schema for the bug report is elided here for space.]

Figure 20: The system prompt used to drive the CATJudge agent.