WEBAPPBENCH: BENCHMARKING FUNCTIONALITY OF AGENT-GENERATED WEB APPLICATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents can generate complete web applications from natural-language specifications, yet plausible interfaces do not guarantee correct behavior. Evaluating functionality requires executing workflows and checking their effects across interactions, views, and sessions. We introduce WEBAPPBENCH, a bilingual benchmark of 100 specifications and 3,197 atomic functional checks (ACs), each with an explicit verification condition. Our evaluation protocol uses browser-based computer-use agents to verify these ACs through adaptive interaction, grounding judgments in runtime evidence and distinguishing supported, contradicted, and unresolved outcomes. Experiments across ten generation configurations reveal persistent gaps between interface completeness and behavioral correctness, with additional agentic effort yielding inconsistent gains. On a 20-application subset, the evaluator resolves 79.5% of human-resolvable ACs and achieves 98.8% agreement with human review on jointly resolved cases. Cross-agent comparisons and component ablations further characterize the effects of evaluator choice, verification guidance, and evidence review. Together, the benchmark and protocol enable requirement-level assessment of generated applications while making the limits of automated verification explicit.

1 INTRODUCTION

Large language models (Qiu et al., 2026; DeepSeek-AI, 2026; GLM, 2026) and coding agents are increasingly used to generate complete Web applications from natural-language requirements (Lu et al., 2025; Liu et al., 2026a). These systems produce interfaces, interaction logic, and data flows within a development workflow. Evaluating the resulting applications requires more than checking whether their code builds or their pages look plausible: users should be able to complete the requested tasks as application state changes across actions, views, and sessions.

Figure 1 illustrates this distinction. A generated project-management application may display a polished task list, creation form, and progress summary. Yet a newly created task may disappear after a reload, remain absent from the project view, or leave the summary unchanged. Build checks and an initial screenshot may miss these failures. Verifying the requirement requires executing the workflow and checking how application state evolves across actions, views, and reloads.

Existing evaluation approaches address different aspects of this problem. Implementation-specific executable tests check behavior against known interfaces and expected outcomes (Jimenez et al., 2024). Yet applications generated from the same specification may differ in interaction paths and DOM structures, requiring tests and selectors to be adapted. LLM/VLM-based assessment offers greater flexibility. For example, screenshot-based evaluation can assess visible content and appearance across different implementations (Si et al., 2025; Chen et al., 2024). Static observations alone, however, cannot establish whether a workflow functions correctly or whether its effects persist. Human evaluators can investigate these behaviors through interaction, adapting their checks to each application, but are costly at benchmark scale.

Browser-based computer-use agents (CUAs), which interact with interfaces through clicking, typing, and scrolling, offer a way to automate these checks (Liu et al., 2026a;b). The central challenge, however, is not merely to explore an application, but to translate an open-ended natural-language specification into explicit verification targets and ground each resulting verdict in runtime evidence

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

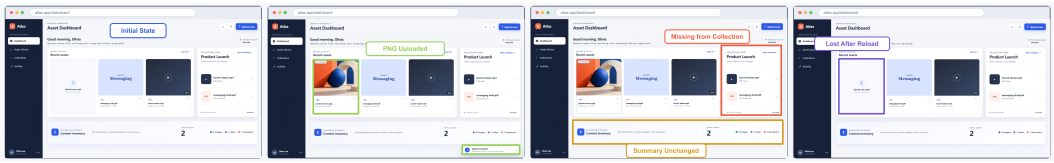


Figure 1: A polished interface can conceal application-state failures: an apparently successful upload may not appear in the collection, update its count, or persist after reload.

relevant to those targets. This motivates examining both how such targets guide evaluation and how reliably CUA judgments agree with human assessment.

To investigate, we introduce **WebAppBench**, a bilingual benchmark comprising 100 web application specifications across five domains and 3,197 atomic checks (ACs), each defining a required behavior and its verification condition. The benchmark covers connected workflows involving state changes, data persistence, and multi-user interactions. We propose a CUA-based evaluation protocol in which a main agent explores the application and prepares the verification setup, then delegates AC groups to parallel subagents while retaining checks with sequential dependencies. CUAs adapt their interactions to the observed interface and produce evidence-backed judgments, with optional VLM review and scores traceable to individual requirements. Beyond comparing generation systems, we analyze evaluator reliability and limitations through human assessment, cross-CUA comparisons, and ablations of verification guidance and VLM review.

We evaluate ten generation configurations spanning direct generation and two coding-agent harnesses. The results reveal a persistent gap between interface completeness and behavioral correctness, particularly in state transitions, value correctness, and runtime integrity; additional agentic effort does not consistently improve functionality. We further examine evaluator reliability through human review, cross-CUA comparisons, and ablations of verification guidance and evidence review. On a stratified subset of 20 applications, our method achieves 98.8% agreement with human review on jointly resolved ACs and resolves 79.5% of human-resolvable ACs, identifying incomplete verification as the principal source of discrepancy. Evidence review reduces score differences between CUAs, while comparisons with open-ended evaluation highlight the importance of explicit requirements and traceable AC-level judgments. Our contributions are threefold:

- We introduce WEBAPPBENCH, a bilingual benchmark of 100 web application specifications and 3,197 atomic functional checks with explicit verification conditions.
- We propose a specification-grounded CUA evaluation protocol that verifies requirements through browser interaction and produces evidence-backed, AC-level judgments.
- We analyze evaluation reliability through human review, cross-CUA comparisons, and component ablations, analyzin automated functional verification.

2 RELATED WORK

Web Application Generation Agents. Coding agents generate web applications, extending evaluation beyond code correctness to user-facing behavior. Design2Code (Si et al., 2025) studies screenshot-to-code generation, while Vision2Web (He et al., 2026) extends visual development to interactive websites. WebGen-Bench, ViBench, and Vibe Code Bench assess functionality through agent-executed tests (Lu et al., 2025; Zhong et al., 2026; Tran et al., 2026). WebCoderBench combines rule-based metrics and LLM judgments (Liu et al., 2026a); WebDevJudge examines evaluator reliability (Li et al., 2025a); and WebCraftBench uses coverage-guided exploration and runtime evidence (Liu et al., 2026b). WebAppBench focuses on multi-stage automated AC construction and autonomous interaction for functional verification. Table 1 summarizes the differences.

Computer-Use Agents. Computer-use agents (CUAs) translate natural-language goals into interface actions, enabling user-like interaction with web and desktop applications (He et al., 2024; Wang et al., 2025). This capability supports automated application testing: agents can navigate unfamiliar interfaces and inspect whether requested behaviors occur (He et al., 2026; Zhong et al., 2026). We use CUAs to execute verification tasks and collect evidence for AC-level judgments in WebAppBench. However, their effectiveness remains sensitive to GUI grounding, task length, and dynamic state,

Table 1: Comparison of web application generation and evaluation benchmarks.

Benchmark	Evaluated Scope	DB Persistence	Multi-user	Evaluation Units	Evaluator
Vision2Web	Frontend + Full-stack	Yes	No	7.5 tests[a] ; 28.2 tests[b]	Workflow-guided CUA
WebCoderBench[c]	Frontend only	No	No	—	Rules + LLM judge
WebGen-Bench	Full-stack	Partial	Partial	6.4 test cases	Self-planning CUA
ViBench	Full-stack	Yes	Partial	2–4 test plans	Plan-guided CUA
Vibe Code Bench	Full-stack	Yes	Yes	9.6 workflows	Workflow-guided CUA
WebDevJudge[d]	Frontend + UI	No	No	29 rubrics	GUI-based judging
WebCraftBench	Frontend only	No	No	13.8 criteria	Coverage-guided CUA
WebAppBench	**Full-stack**	**Yes**	**Yes**	**31.9 atomic checks**	**Checks-guided self-planning CUA**

[a]Interactive frontends; [b]interactive full-stack websites. [c]WebCoderBench uses 24 quality metrics; average checklist size is not reported (—). [d]WebDevJudge evaluates judges rather than generation systems. DB: database. CUA: computer-use agent.
For WebAppBench, atomic checks (ACs) define verification targets; CUAs follow the accompanying verification guidelines while adapting their actions to the application interface.

as demonstrated by OSWorld and OSWorld 2.0 (Xie et al., 2024; Yuan et al., 2026). Execution failures can be mistaken for application defects (Li et al., 2025a; Liu et al., 2026b; Hong et al., 2026), motivating explicit verification targets and judgments grounded in observed outcomes.

LLM/VLM-as-a-Judge and Evidence-Grounded Evaluation. LLM judges assess open-ended outputs (Zheng et al., 2023; Liu et al., 2023), while multimodal judges incorporate visual inputs and customized rubrics (Chen et al., 2024; Lee et al., 2024). WebDevJudge and IWC-Bench extend evaluation to generated websites and interaction evidence (Li et al., 2025a; Liu et al., 2026b). However, visual perception and functional verification remain challenging, when evidence spans interaction states (Li et al., 2025b;a; Wan et al., 2025). FActScore, VeriScore, and FaStFact ground factuality assessment in claim-level verification against retrieved evidence. (Min et al., 2023; Song et al., 2024). WebAppBench decomposes application requirements into atomic functional checks with explicit verification conditions. It gathers evidence for each AC and distinguishes supported, contradicted, and unresolved outcomes, enabling auditable judgments of user-facing behavior.

3 WebAppBench: Benchmark Construction

3.1 Task Definition and Specification Collection

WebAppBench evaluates whether agent-generated web applications satisfy natural-language specifications. Given a specification q_i, a coding agent builds an application A_i for browser-based evaluation. We assess observable, user-facing behavior against the specified requirements while leaving unspecified interface and implementation choices unconstrained; no reference codebase or fixed interaction sequence is required. Figure 2 summarizes the benchmark construction pipeline.

Specifications are automatically constructed from curated image-based posts and textual descriptions collected from social media platforms using Claude Opus 4.8(Anthropic, 2026a). We consolidate overlapping ideas into coherent application scenarios and extend them with connected workflows, state transitions, and data requirements, emphasizing behavior across features, views, and sessions rather than isolated rendering or interactions. The benchmark contains 100 specifications, evenly divided between Chinese and English, across five domains: Life & Health, Entertainment & Social, Education & Reference, Work & Business, and Creation & Development. Each task includes atomic checks and verification guidelines, with task-specific file fixtures and user identities where needed. Representative specifications appear in Tables 8 and 9 of Appendix D.

3.2 Atomic Check and Guideline Construction

Each benchmark task starts from a natural-language specification describing the Web application to be generated. We denote the specification of task i by q_i and decompose its functional requirements into m_i evaluation checks:

$$\mathcal{C}_i = \{c_{ij}\}_{j=1}^{m_i}, \qquad c_{ij} = (p_{ij}, v_{ij}, t_{ij}, k_{ij}), \tag{1}$$

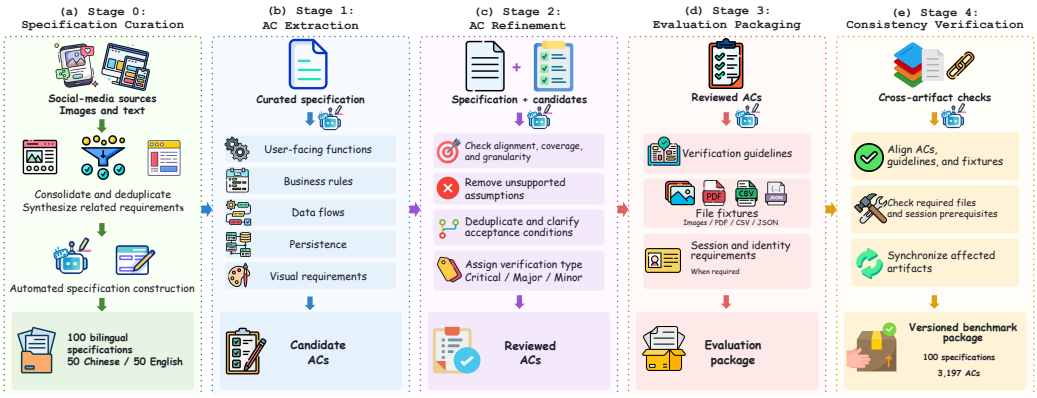


Figure 2: Overview of the WebAppBench construction pipeline. Social-media materials are curated into bilingual application specifications, followed by AC extraction and refinement. Verification guidelines, file fixtures, and session requirements are assembled into an evaluation package and checked for cross-artifact consistency.

where p_{ij} describes the required behavior, v_{ij} its natural-language verification condition, t_{ij} the verification type, and k_{ij} its importance (*critical*, *major*, or *minor*). Each AC defines an observable acceptance target without prescribing implementation details, interface-specific selectors, or a fixed interaction path. The AC construction process, also performed with Claude Opus 4.8, includes extraction and refinement. Extraction identifies candidate requirements across application modules and workflows, while refinement merges redundant or overly fragmented candidates, clarifies verification conditions, and checks alignment with the original specification. Atomicity refers to a coherent functional outcome rather than a single interface action: verifying an AC may require prerequisite state and multiple interactions, while distinct functional outcomes remain separate targets.

We organize checks into seven verification types: *structural*, *navigation*, *state transition*, *value correctness*, *persistence*, *perceptual*, and *runtime integrity*, as summarized in Table 7 of Appendix D. These types characterize the observations required for verification rather than the application's implementation architecture; perceptual checks concern specification-defined visual requirements rather than unrestricted aesthetic quality. Since the browser-based CUA does not receive the original specification, each task includes a verification guideline generated from the specification and reviewed checks. The CUA first reads the guideline to understand the application's intended functionality and verification prerequisites, then evaluates the atomic checks (ACs). The guideline supports efficient planning through navigation cues, interaction dependencies, and task-specific setup; required file fixtures are provided separately. It adds no acceptance requirements or fixed action sequence: each verdict remains grounded in observed behavior and the AC's verification condition (Section 4).

3.3 Quality Control and Dataset Coverage

Quality control checks both AC-level alignment and package-level consistency among ACs, guidelines, fixtures, and session requirements (Figure 2). Prior benchmarks typically evaluate each application with a relatively small set of test cases, workflows, or criteria (Table 1). In contrast, WebAppBench specifications emphasize compositional functional complexity: multiple functions are interdependent through shared application state, cross-view dependencies, persistence, value computation, file inputs, and multi-user behavior. At the dataset level, Figure 3 summarizes the category distribution and specification lengths for the Chinese and English subsets. WebAppBench contains 3,197 checks across 100 tasks, with a median of 32 per task. As shown in Figure 7, state-transition and value-correctness checks jointly account for 74.1% of the total; the benchmark further includes 205 persistence checks, 142 multi-session checks spanning 33 tasks, and file fixtures for 37 tasks. Together, this coverage supports evaluation of stateful workflows, data persistence, and cross-user behavior beyond interface appearance.

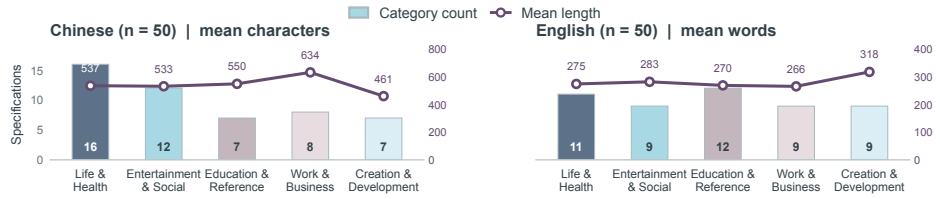


Figure 3: Category distribution of WebAppBench specifications by language. Each panel contains 50 specifications; values above the bars give category counts.

4 Evaluation Protocol

We adopt a *CUA-as-Judge* perspective to evaluate whether web applications satisfy their specified requirements. The CUA first reads a task-level verification guideline to understand the intended functionality and plan its evaluation efficiently. It then interacts with the application to assess each AC, grounding its judgments in observed behavior.

AC-guided interaction. The CUA first reads the verification guideline to understand the task context, prerequisites, and interaction dependencies. The guideline supports planning while leaving the agent to choose actions based on the observed interface. A main CUA explores the application and prepares the setup, after which groups of ACs are evaluated in parallel by subagents, while ACs with sequential dependencies remain in the main session. Each AC is evaluated within a specified iteration budget, with action traces, screenshots, page observations, and runtime signals recorded as evidence for judgment. Details of parallel execution, session management, and persistence and multi-session checks are provided in Appendix B.

Judgment and optional review. The CUA produces an initial verdict, a supporting rationale, and a self-reported confidence score. Verdicts distinguish *supported* (S), *contradicted* (C), and *insufficient evidence* (U). These outcomes reflect whether the observed behavior supports the verification condition, contradicts it, or leaves it unresolved. Insufficient evidence does not establish a functional defect. When optional review is enabled, judgments below a preset confidence threshold are referred to a VLM, which evaluates the recorded evidence against the same verification condition without further browser interaction. Its verdict replaces the initial judgment for reviewed checks; otherwise, the CUA verdict is retained. The VLM review can reassess existing evidence but cannot establish behavior that was not observed. Specific settings are provided in Appendix B.

Functionality scoring. For application i, each AC c_{ij} receives a verdict y_{ij} and an importance weight $w_{ij} \in \{3, 2, 1\}$ for critical, major, and minor requirements, respectively. We assess application functionality using pass rates at each importance level and an overall FuncScore.

Let $\mathcal{C}_i^{(t)}$ denote the checks at importance level $t \in \{\text{critical}, \text{major}, \text{minor}\}$. For each nonempty category, its pass rate is

$$P_i^{(t)} = \frac{\sum_{j \in \mathcal{C}_i^{(t)}} \mathbb{I}[y_{ij} = \text{S}]}{|\mathcal{C}_i^{(t)}|}. \tag{2}$$

Only supported checks receive pass credit; contradicted and insufficient-evidence checks remain in the denominator.

To summarize overall functionality while emphasizing essential requirements, we define

$$\text{FuncScore}_i = \frac{\sum_j w_{ij} \mathbb{I}[y_{ij} = \text{S}]}{\sum_j w_{ij}} \cdot G_i, \qquad G_i = \begin{cases} P_i^{(\text{critical})}, & |\mathcal{C}_i^{(\text{critical})}| > 0, \\ 1, & \text{otherwise.} \end{cases} \tag{3}$$

The critical gate reduces the overall score when essential requirements are not verified as satisfied. We macro-average FuncScore over applications and each importance-level pass rate over applications containing checks at that level. These metrics quantify application functionality under our evaluation protocol; evaluator reliability and limitations are examined separately in Section 6.

Table 2: Application-generation results on the 100 WebAppBench specifications. Functional metrics are application-level macro-averages, reported as percentages. Critical, Major, and Minor give the pass rate of atomic checks at each criticality level; `insufficient_evidence` verdicts count as failures. Generation-Overhead metrics cover application generation only and are reported per application. Harness "Direct" denotes direct generation by the model without an agent harness. **Bold** and underlined values mark the best and second-best functional results. Missing measurements are marked with –. Qwen3.8-27B is deployed locally, so no API cost is reported for it.

Harness	Model	Reasoning Effort	Functional Evaluation (%)				Generation Overhead		
			FuncScore ↑	Pass Rate by Criticality ↑			Latency	Cost	Model
				Critical	Major	Minor	(Min) ↓	(USD) ↓	Calls ↓
Direct	Claude Opus 5	–	**49.67**	**68.85**	**63.51**	**72.85**	20.69	2.27	–
	GPT-5.6-Sol	–	26.50	49.58	41.16	54.30	13.98	0.53	–
OpenCode	Claude Opus 5	high	36.57	58.14	53.30	58.68	76.37	14.04	166.00
	GLM-5.3-Flash	high	32.72	51.29	46.56	55.21	59.18	0.62	105.00
	Qwen3.8-27B	high	32.78	53.77	48.52	62.89	78.08	–	99.77
Codex	Claude Opus 5	high	47.45	65.59	60.68	61.15	52.93	3.82	15.63
	GPT-5.6-Sol	high	25.43	46.48	39.10	51.86	7.59	0.46	13.57
	Kimi-K3	high	<u>47.54</u>	<u>66.02</u>	<u>62.22</u>	64.31	66.31	2.66	68.30
	GLM-5.3-Flash	high	26.09	46.88	42.65	51.61	32.86	0.24	64.57
	Qwen3.8-27B	high	44.42	64.73	56.81	<u>65.47</u>	168.54	–	77.66

5 Experiments

5.1 Experimental Setup

Generation settings We evaluate 10 generation configurations under three harness settings: direct generation without an agent harness ("Direct" in Table 2), OpenCode(OpenCode Team, 2026), and Codex(OpenAI, 2026a). Direct generation uses Claude-Opus-5(Anthropic, 2026b) and GPT-5.6-Sol(OpenAI, 2026b). For the *Direct* setting, the model generates all frontend and backend source files in a single response, formatted as a JSON file manifest. The harnesses validate and writes these files, combines them with a fixed React/Vite and FastAPI/SQLite scaffold, and builds and deploys the application without iterative model-driven repair. Additionally, we attempt to build and deploy each direct generated application, allowing up to three retries after an initial build failure. For *OpenCode*, we evaluate Claude-Opus-5, GLM-5.3-Flash (GLM, 2026), and Qwen3.8-27B (Qiu et al., 2026). For *Codex*, we evaluate five backbone models: Claude-Opus-5, GPT-5.6-Sol, Kimi-K3 (Team, 2026), GLM-5.3-Flash, and Qwen3.8-27B. Generation prompt details are in Appendix E.1. All harness-based configurations use high reasoning effort.

Evaluation settings We use GLM-5.3-Flash as the default CUA and GPT-5.6-Terra for optional evidence review. Iteration limits are configured by AC type to accommodate differences in verification effort. Checks of readily observable interface properties typically require less setup, whereas verifying action effects or computed results may involve constructing inputs and inspecting multiple application states. We therefore allocate up to 8 iterations for structural checks, 12 for navigation, 30 for state-transitions, 25 for value-correctness, 10 for perceptual checks, 15 for persistence, and 18 for runtime-integrity. These limits are held fixed across generation systems. VLM review follows the procedure described in Section 4 and uses evidence recorded during CUA execution. Interaction traces, screenshots, and AC judgments are retained for subsequent analysis.

Metrics We report FuncScore and AC pass rates at the critical, major, and minor levels, following Section 4. To characterize generation efficiency, we additionally report per-application generation latency (minutes), cost (USD), and the number of model calls.

5.2 Main Results on WebAppBench

Table 2 compares functional quality and generation overhead across 10 configurations.

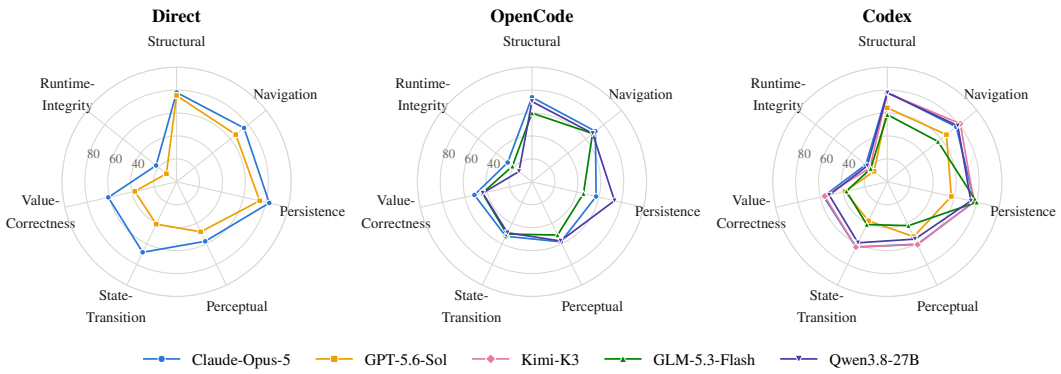


Figure 4: AC pass rates (%) by verification type, with one panel per harness. Each axis gives the pass rate of one verification type, macro-averaged over the applications that contain checks of that type; axes are ordered clockwise from Structural by decreasing mean pass rate over the 10 configurations. Each backbone model keeps the same color and marker across panels.

Even the best configuration falls below 50 FuncScore. Scores range from 25.43 to 49.67, with direct Claude-Opus-5 leading on FuncScore and all three criticality levels. Codex with Kimi-K3 ranks second (47.54), ahead of Claude-Opus-5 with Codex (47.45). Major checks have the lowest pass rate across configurations, revealing persistent gaps in functional completeness.

Harness iterations may drift from functional requirements. With Claude-Opus-5, direct generation (49.67) outperforms Codex (47.45) and OpenCode (36.57). To investigate this, we inspect the generation trajectories and observe a recurring pattern: generic harnesses spend many iterations resolving build issues and refining local implementation details, while some user-requested functionality remains incomplete. As an additional diagnostic not reported in Table 2, only 81% of Claude-Opus-5 and 69% of GPT-5.6-Sol direct generations succeed on the first attempt, despite harness-based systems achieving substantially higher build success rates. This suggests that the advantage of direct generation is unlikely to stem from easier deployment alone. Instead, addressing the specification in a single pass may help preserve greater functional coherence, whereas repeated execution-driven revisions can shift effort away from the original requirements.

GPT-5.6-Sol reproduces interface structure but falls short on behavior. Its direct and Codex configurations yield the two lowest FuncScores (26.50 and 25.43). Under direct generation, its structural pass rate approaches that of Claude-Opus-5 (75.29 vs. 77.97), yet its state-transition pass rate is 27.28 points lower (Section 5.3). This disparity exposes a gap between implementing visible interface elements and the functional behavior behind them.

The costliest configuration scores 13.10 points below its direct counterpart. OpenCode with Claude-Opus-5 takes 76.37 minutes, costs 14.04 USD, and averages 166.00 model calls, yet underperforms direct generation costing 2.27 USD. It also scores 10.88 points below Codex with the same backbone despite taking longer (76.37 vs. 52.93 minutes). Conversely, Codex with Kimi-K3 achieves the best harness-based FuncScore (47.54) at 2.66 USD. Higher expenditure and more iterations therefore do not consistently yield more complete user-facing functionality.

5.3 Analysis by Verification Type

Figure 4 reports AC pass rates across the seven verification types, macro-averaged over applications containing checks of each type.

Behavioral requirements are the main bottleneck. Across the 10 configurations, structural and navigation checks average 71.35% and 70.66%, compared with 52.62% for state-transition and 47.82% for value-correctness. Since the latter two account for 74.1% of WebAppBench, this weakness affects the core behavioral requirements rather than a small set of edge cases. Runtime-integrity is the weakest type overall, with pass rates of only 11.38–27.05%.

Interface completeness does not imply behavioral correctness. The gap between structural and state-transition performance varies substantially across models. Under direct generation, GPT-5.6-Sol

and Claude-Opus-5 achieve similar structural pass rates (75.29% vs. 77.97%) but differ sharply on state-transition checks (40.95% vs. 68.23%). Thus, similarly complete interfaces can conceal substantial differences in behavioral implementation.

Harness effects are type- and domain-dependent. Harnesses reshape functional profiles rather than uniformly improving them. For example, OpenCode reduces persistence for Claude-Opus-5 relative to direct generation, while for GLM-5.3-Flash it outperforms Codex on state-transition and value-correctness; Qwen3.8-27B shows the opposite tendency on several behavioral types. This heterogeneity also appears across application domains: Creation & Development remains comparatively challenging, while stronger domains and the relative performance of models and harnesses vary across configurations. Overall, generation behavior depends jointly on the backbone, functional requirement type, and application domain.

6 Analysis

The analyses in this section require repeated browser-based evaluation under different experimental settings. To keep these comparisons computationally tractable while covering applications with different levels of functional completeness, we select a stratified subset of 20 specifications from the full set of 100. We use the mean FuncScore as a proxy for the difficulty of satisfying a specification's requirements, reducing dependence on any single generator. We partition specifications into Low ($[0, 30)$), Mid ($[30, 70)$), and High ($[70, 100]$) score strata and select 6, 7, and 7 specifications from these strata, respectively. This allocation ensures that the comparisons include cases with limited, partial, and substantial requirement satisfaction, rather than concentrating on one performance range. Here, Low, Mid, and High refer to observed scores: lower scores indicate greater difficulty for the evaluated generation systems. The subset also covers all five task categories and contains 10 Chinese and 10 English specifications. The generation configurations and complete sample selection are provided in Appendix C.

6.1 Alignment between Automatically Generated and Expert-Authored ACs

To assess the AC construction procedure described in Section 3.2, we compare automatically generated and expert-authored ACs on ten specifications from our benchmark and ten requests from real users. Within each task, we match ACs one-to-one when their functional goal, conditions, and expected outcome agree. Macro scores average the per-task scores. We additionally measure full coverage: an expert-authored AC is fully covered when one or more automatically generated ACs jointly express its complete requirement. Table 4 summarizes the results.

Macro full coverage reaches 90.3% on the benchmark set and 70.3% on the real-user set, while strict one-to-one macro F1 is 40.4% and 49.7%, respectively. One contributor to the gap between these measures is decomposition granularity: a requirement expressed in one expert-authored AC may be split across several automatically generated ACs. These ACs can jointly cover the requirement but cannot all receive strict one-to-one matches. Despite these differences, automatically generated ACs cover most expert-identified requirements in both our specifications and real-user requests. Details of the matching protocol and metric definitions are provided in Appendix C.4.

6.2 Agreement Between CUA-Based Evaluation and Human Review

We manually review all 595 AC verdicts across the 20 applications in our analysis subset, using the GLM-5.3-Flash evaluator described in Section 5.1. Human review resolves 507 ACs as supported or contradicted. We define coverage on these human-resolvable ACs as the fraction also resolved by the CUA-based evaluator: 403/507 (79.5%). Among these jointly resolved cases, 398 verdicts agree with human review (98.8%). The evaluator returns insufficient evidence for 190 ACs (31.9%). Human review resolves 104 (54.7%): 63 supported and 41 contradicted; 86 remain unresolved. These results indicate that insufficient-evidence verdicts can reflect incomplete automated verification rather than missing functionality. Such verdicts can arise when the CUA fails to complete required interactions or exhausts its action budget before obtaining decisive evidence, even when the functionality is implemented. Only five cases with definitive judgments on both sides disagree. Thus, the principal discrepancy is unresolved automated verdicts rather than incorrect definitive judgments. We treat insufficient evidence as an abstention, not as evidence of functional failure.

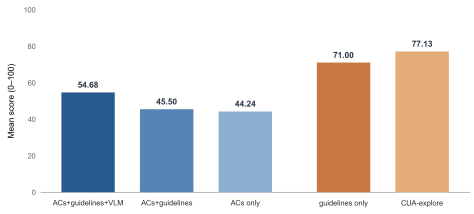


Figure 5: Mean scores across five evaluation settings: three AC-based and two holistic CUA-based.

Table 3: CUA sensitivity with GPT-5.6-Terra fixed as the evidence reviewer. Reported metrics include CUA-only and reviewed FuncScores, unresolved rates, and VLM review rates.

CUA	CUA-only Score	Reviewed Score	Unresolved (%)	VLM review(%)
GLM-5.3-Flash	45.50	54.68	56.80	38.40
Claude Opus 4.8	65.04	63.16	18.90	15.30

6.3 Structured versus Open-Ended Evaluation

We examine how explicit requirements, execution guidance, and visual adjudication shape functional evaluation. Detailed settings and score provenance appear in Appendix C.3; we exclude two apps on which autonomous exploration repeatedly stalled. Figure 5 reports scores of 54.68, 45.50, and 44.24 for ACs+guidelines+VLM, ACs+guidelines, and ACs only, respectively. These descriptive differences highlight the influence of evidence adjudication on awarded credit. Guidance can also improve efficiency: on `Radiation Survival` Web application, guidelines-only uses 44 model turns versus 221 for CUA-explore, an 80.1% reduction. Guidelines-only and CUA-explore assign higher scores of 71.00 and 77.13, respectively, yet evaluate approximately 10 fewer checks per app than our CUA-as-Judge method. Their holistic scores reflect self-selected checks and may overlook unimplemented functionality. In contrast, CUA-as-Judge combines broader requirement coverage with traceable criterion-level judgments, making coverage gaps auditable and enabling a more comprehensive assessment of functional completeness.

6.4 Sensitivity to the CUA and Evidence Reviewer

We assess evaluator sensitivity on the same application subset as Section 6.3. With GPT-5.6-Terra fixed as the evidence reviewer, GLM-5.3-Flash leaves substantially more CUA judgments unresolved than Claude Opus 4.8 (56.8% vs. 18.9%) and consequently triggers VLM review more often (38.4% vs. 15.3%; Table 3). Trajectory inspection indicates weaker interaction grounding and lower completion of long-horizon verification for GLM-5.3-Flash under the same interaction budgets. Accordingly, the raw CUA-only FuncScores differ substantially (45.50 vs. 65.04), whereas evidence review narrows the gap to 8.48 points (54.68 vs. 63.16). Despite this difference in absolute scores, application-level results remain strongly correlated after review (Spearman $\rho = 0.809$), with relatively low- and high-scoring applications generally retaining similar relative positions across CUAs. Reviewer choice has little aggregate effect: with GLM-5.3-Flash fixed as the CUA, GPT-5.6-Terra and Claude Opus 5 both yield a mean FuncScore of 54.68 after rounding. Overall, CUA choice affects absolute scores more than the broad relative assessment of application quality, while VLM-based evidence review substantially reduces dependence on the execution CUA.

Beyond evaluator-model sensitivity, Appendix C.5 compares full CUA evaluation with DOM-only and A11y+CUA baselines in terms of verification coverage and cost, while Appendix C.6 examines run-to-run variability across repeated generations of the same specifications.

7 Conclusion

We introduced WebAppBench, a bilingual benchmark and evidence-grounded evaluation protocol for functional correctness in agent-generated web applications. Experiments across ten generation configurations reveal that plausible interfaces often conceal behavioral failures and that additional agentic effort does not consistently improve functionality. Human review and evaluator analyses show strong agreement on resolved checks, while identifying incomplete verification as a key limitation. These findings motivate improvements in both application generation and automated verification, with explicit requirements and traceable evidence supporting progress toward reliable web applications.

AI USE STATEMENT

The research questions, the design of the benchmark, and the evaluation protocol are entirely the work of the authors; AI tools took no part in conceiving the research or planning the experiments.

During manuscript preparation, we utilized AI tools to polish the writing. Their role was to rephrase sentences, correct grammatical errors, and make the text read more smoothly, which improved the clarity and readability of several sections.

All analyses were designed, conducted, and validated by the authors. Under the authors' direction, AI coding tools served as assistants for routine engineering work, such as writing scripts that collect evaluation results into tables and figures and formatting those tables. The authors recomputed every reported number from the raw evaluation records, cross-checked the results, and confirmed each analytical claim against the underlying evidence.

Responsibility for the manuscript rests fully with the authors, and this extends to all text, code, and figures created with AI assistance. We have verified that such content complies with ethical standards and involves neither plagiarism nor any form of scientific misconduct.

ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. Building and evaluating WebAppBench involved no human subjects and no personal data. The fixture files used in upload interactions were created for the benchmark, and those resembling sensitive records.

Generated applications are deployed only for evaluation, and the computer-use agent interacts solely with the application under test, never with real user accounts or third-party services. During our research, we have made every effort to avoid biased and discriminatory results. No personally identifiable information was used, and no experiments that might cause privacy or security issues were conducted. We are committed to maintaining transparency and integrity throughout the entire research process.

REPRODUCIBILITY STATEMENT

To support replication and verification, the benchmark and the complete evaluation code are available in an anonymous repository. Section 5.1 specifies the evaluated configurations and the evaluation settings, and further implementation details are provided in Appendix B. Because every configuration is evaluated under a single, fixed protocol, other researchers can rerun the evaluation on the released materials and assess new generation systems on equal terms. We hope these resources will help the community build on our work.

REFERENCES

Anthropic. Claude Opus 4.8. https://www.anthropic.com/news/claude-opus-4-8, 2026a. Released May 2026.

Anthropic. Introducing Claude Opus 5. https://www.anthropic.com/news/claude-opus-5, 2026b. Released July 2026.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yaochen Wang, Yinuo Liu, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 6562–6595. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/chen24h.html.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *CoRR*, abs/2606.19348, 2026. doi: 10.48550/ARXIV.2606.19348. URL https://doi.org/10.48550/arXiv.2606.19348.

GLM. GLM-5: from vibe coding to agentic engineering. *CoRR*, abs/2602.15763, 2026. doi: 10.48550/ARXIV.2602.15763. URL https://doi.org/10.48550/arXiv.2602.15763.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. In *ACL*, pp. 6864–6890, 2024. doi: 10.18653/V1/2024.ACL-LONG.371. URL https://dblp.org/rec/conf/acl/HeYM0D0L024.

Zehai He, Wenyi Hong, Zhen Yang, Ziyang Pan, Mingdao Liu, Xiaotao Gu, and Jie Tang. Vision2web: A hierarchical benchmark for visual website development with agent verification. *CoRR*, abs/2603.26648, 2026. URL https://dblp.org/rec/journals/corr/abs-2603-26648.

Sirui Hong, Zhijie Liu, Tengfei Li, Wei Tao, Yifan Wu, and Chenglin Wu. DiagEval: Trajectory-conditioned diagnosis for reliable software evaluation with GUI agents. *CoRR*, abs/2605.17439, 2026. doi: 10.48550/arXiv.2605.17439. URL https://dblp.org/rec/journals/corr/abs-2605-17439.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Seongyun Lee, Seungone Kim, Sue Hyun Park, Geewook Kim, and Minjoon Seo. Prometheus-vision: Vision-language model as a judge for fine-grained evaluation. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, volume ACL 2024 of *Findings of ACL*, pp. 11286–11315. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.672. URL https://doi.org/10.18653/v1/2024.findings-acl.672.

Chunyang Li, Yilun Zheng, Xinting Huang, Tianqing Fang, Jiahao Xu, Yangqiu Song, Lihui Chen, and Han Hu. Webdevjudge: Evaluating (m)llms as critics for web development quality. *arXiv preprint arXiv:2510.18560*, 2025a.

Lei Li, Yuancheng Wei, Zhihui Xie, Xuqing Yang, Yifan Song, Peiyi Wang, Chenxin An, Tianyu Liu, Sujian Li, Bill Yuchen Lin, Lingpeng Kong, and Qi Liu. Vl-rewardbench: A challenging benchmark for vision-language generative reward models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 24657–24668. Computer Vision Foundation / IEEE, 2025b. doi: 10.1109/CVPR52734.2025.02296. URL https://openaccess.thecvf.com/content/CVPR2025/html/Li_VL-RewardBench_A_Challenging_Benchmark_for_Vision-Language_Generative_Reward_Models_CVPR_2025_paper.html.

Chenxu Liu, Yingjie Fu, Wei Yang, Ying Zhang, and Tao Xie. Webcoderbench: Benchmarking web application generation with comprehensive and interpretable evaluation metrics. In *ACL*, pp. 11632–11666, 2026a. doi: 10.18653/V1/2026.ACL-LONG.535. URL https://dblp.org/rec/conf/acl/LiuFYZX26.

Chenxu Liu, Zilu Zou, Peizhong Gao, Jiawen Tao, Zhexin Zhang, Guang Chen, Haowei Lin, Ying Zhou, Tianyi Bai, Dolly Deng, Suncong Zheng, and Maxm Pan. IWC-Bench: Evaluating web application generation from a software testing perspective, 2026b. URL https://arxiv.org/abs/2609.15387.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2511–2522. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.153. URL https://aclanthology.org/2023.emnlp-main.153/.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch. In *NeurIPS*, 2025. URL https://dblp.org/rec/conf/nips/LuYRHXWSZZL25.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12076–12100. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.741. URL https://aclanthology.org/2023.emnlp-main.741/.

OpenAI. Codex CLI. https://learn.chatgpt.com/docs/codex/cli, 2026a.

OpenAI. GPT-5.6-Sol model documentation. https://developers.openai.com/api/docs/models/gpt-5.6-sol, 2026b.

OpenCode Team. OpenCode: The open source AI coding agent. https://opencode.ai/en/docs, 2026.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, Fan Zhou, Hao Luo, Haofeng Huang, Haoran Lian, Haoyan Huang, Hongqing Chen, Jianwei Zhang, Jing Xu, Junjie Wang, Langshi Chen, Liangyu Wang, Linlang Jiang, Man Yuan, Minmin Sun, Peng Jin, Siqi Zhang, Siyu Wang, Xingzhang Ren, Yakai Wang, Yi Zhang, Yiming Dong, Yizhong Cao, Yubo Ma, Yunfei Mao, Bo Zheng, and Dayiheng Liu. On the design of qwen3.8-next architecture: Evaluation, efficiency, and training stability. *CoRR*, abs/2608.30320, 2026. doi: 10.48550/ARXIV.2608.30320. URL https://doi.org/10.48550/arXiv.2608.30320.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 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*, pp. 3956–3974. Association for Computational Linguistics, 2025.

Yixiao Song, Yekyung Kim, and Mohit Iyyer. VeriScore: Evaluating the factuality of verifiable claims in long-form text generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9447–9474. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-emnlp.552. URL https://aclanthology.org/2024.findings-emnlp.552/.

Kimi Team. Kimi K3: open frontier intelligence. *CoRR*, abs/2607.24653, 2026. doi: 10.48550/ARXIV.2607.24653. URL https://doi.org/10.48550/arXiv.2607.24653.

Hung Tran, Langston Nashold, Rayan Krishnan, Antoine Bigeard, and Alex Gu. Vibe code bench: Evaluating ai models on end-to-end web application development. In *CAIS*, pp. 514–536, 2026. URL https://dblp.org/rec/conf/cais/TranNKBG26.

Yingjia Wan, Haochen Tan, Xiao Zhu, Xinyu Zhou, Zhiwei Li, Qingsong Lv, Changxuan Sun, Jiaqi Zeng, Yi Xu, Jianqiao Lu, Yinhong Liu, and Zhijiang Guo. Fastfact: Faster, stronger long-form factuality evaluations in llms. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 23814–23854, 2025.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Y. Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open foundations for computer-use agents. In *Advances in Neural Information Processing Systems*, 2025. URL https://dblp.org/rec/conf/nips/WangWLYXWDGXWSL25.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *NeurIPS*, 2024. URL https://dblp.org/rec/conf/nips/XieZCLZCHCSLLXZ24.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld2.0: Benchmarking computer use agents on long-horizon real-world tasks. *CoRR*, abs/2606.29537, 2026. doi: 10.48550/arXiv.2606.29537. URL https://dblp.org/rec/journals/corr/abs-2606-29537.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Peter Zhong, Pashootan Vaezipoor, Fuyang Cui, Vaibhav Kumar, James Austin, Azin Asgarian, Toby Ho, Paul Inder, Imen Kedir, Zhen Li, Nicholas Ondo, Asna Shafiq, Ibrahim Sheikh, Edouard Sioufi, Setareh Soltanieh, Ben Wilde, Jacky Zhao, Ryan Carelli, Heather Miller, and Michele Catasta. Vibench: A benchmark on vibe coding. In *CAIS*, pp. 497–513, 2026. URL https://dblp.org/rec/conf/cais/ZhongVCKAAHIKLO26.

LIMITATIONS

WebAppBench targets the functional correctness of specified user-facing behavior; non-functional properties such as security and performance are outside its scope. We focus on whether generated applications correctly implement the requested workflows, state changes, and data flows, which is a prerequisite for their practical use and remains a major challenge for current generation systems. Properties such as vulnerability resistance, performance under load, and code maintainability are complementary to this goal and can be assessed on the same applications with established tools. Integrating these dimensions with functional verification is a natural direction for future work.

A WEBAPPBENCH DEPLOYMENT

We used a React and TypeScript frontend built with Vite, a Python FastAPI backend served by Uvicorn, and SQLite for persistence. Applications were stored in separate working directories for each task and generation configuration. The deployment environment required Python 3.11 or later and, for local frontend builds, Node.js 22 or later.

Direct generation. The model returned application-specific source files in a single JSON response. The platform validated the response format, required files, and permitted output paths before writing the files to disk. It then assembled a runnable project by adding fixed dependency manifests, TypeScript and Vite configuration, a runtime wrapper, and build and startup scripts. Frontend dependencies were installed with `npm install`, and `npm run build` performed TypeScript checking and produced the frontend assets. After installing the backend dependencies, Uvicorn launched the FastAPI application. The runtime wrapper served the compiled frontend alongside the backend API, allowing the frontend to use same-origin `/api` requests.

Harness-based generation. The agent wrote the complete project directly into its task workspace, including application code, dependency manifests, and installation and startup instructions. During generation, it could install dependencies, execute builds and available tests, inspect failures, and revise the project. Executed commands and their outcomes were recorded for inspection. Deployment was

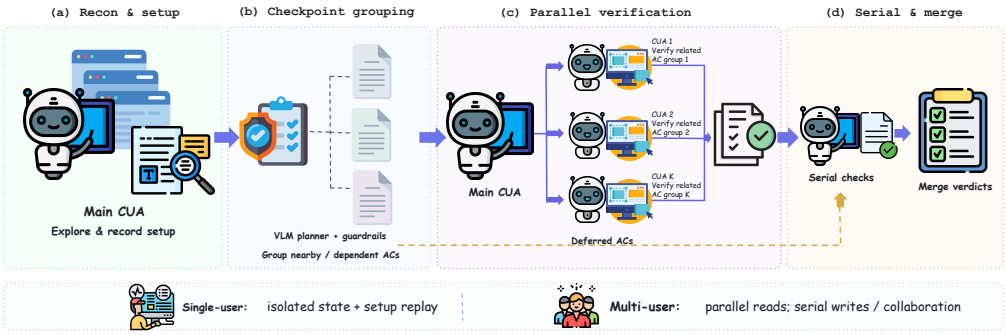


Figure 6: Star-shaped parallel evaluation. The main CUA explores the app and records setup. A VLM groups related atomic checks (ACs), which child CUAs verify in parallel. Deferred checks are then executed serially before all AC verdicts are merged.

performed separately using the resulting project files and documented startup commands; completion of the generation session alone did not establish that the application was running.

For deployment, we used Docker Compose when available, building the application image and mounting a separate persistent volume for each SQLite database. Otherwise, we installed backend dependencies in a task-specific Python virtual environment, built the frontend locally, and launched the application with a separate database directory. Each application received its own local port. A deployment was marked ready for evaluation only after its `/api/health` endpoint passed a health check, after which its URL was recorded for browser-based evaluation.

B Evaluation Details

Verification setup Verification guidelines identify the preconditions and dependencies needed to assess each task. Depending on the requirements, these may include seed data, file fixtures, participant identities and access links, or external-data checks. The guidelines help the CUA prepare the application state and order dependent interactions. They do not replace the AC's verification condition: judgments remain grounded in observed application behavior.

Execution and session state Each AC is evaluated within an AC-specific iteration budget. As shown in Fig. 6 The main CUA observes the current interface, selects an action, and inspects the resulting state. For checks requiring cross-session verification, it can delegate scoped verification tasks to viewer subagents operating in isolated browser sessions with distinct participant identities. Delegation follows a star shaped structure: viewer subagents return their observations and local verdicts to the main CUA and cannot delegate further. The main CUA incorporates this evidence into AC verification. Throughout execution, the evaluator records action traces, screenshots, page-state observations, and runtime signals.

By default, sequential execution reuses the main CUA's browser session across checks rather than resetting the application before every check. Consequently, later checks may encounter state created during earlier interactions. Viewer sessions are created separately for delegated tasks and closed upon completion.

Persistence and multi-session checks Persistence requirements are assessed by comparing relevant observations before and after a page reload. Multi-session requirements use isolated browser sessions with separately provisioned participant identities. A shared unique marker identifies the relevant data across participants. These checks assess the consequences required by the AC; successful navigation or action execution alone does not establish functional correctness.

Optional evidence review When review is enabled, a CUA judgment with self-reported confidence below 0.7 is referred to a VLM. The review input contains the AC's verification condition, recorded

screenshots and action trace, and available page-state and runtime observations. The reviewer produces a verdict from the same three-outcome set: supported, contradicted, or insufficient evidence. Its verdict is adopted for reviewed checks; the initial CUA verdict is retained otherwise.

Budget exhaustion or difficulty interpreting an outcome may leave an assessment unresolved or low-confidence. Review is restricted to the recorded evidence and does not introduce additional browser interactions. It can therefore leave an AC unresolved when the required behavior has not been observed. The confidence threshold is an operational routing parameter, not a calibrated probability of correctness.

C Experimental Details

C.1 Shared Stratification and Sampling

For each query q, let $F_{q,g}$ denote the FuncScore of the application produced under generation configuration g. We define the stratification score as

$$\bar{F}_q = \frac{1}{4} \sum_{g \in \mathcal{G}} F_{q,g}, \tag{4}$$

where $\mathcal{G}$ contains Opus-5 without Codex, Codex + Claude Opus 5, Codex + Kimi-K3, and Codex + Qwen27B. Queries are assigned to strata as follows:

$$\text{stratum}(q) = \begin{cases} \text{Low}, & 0 \le \bar{F}_q < 30, \\ \text{Mid}, & 30 \le \bar{F}_q < 70, \\ \text{High}, & 70 \le \bar{F}_q \le 100. \end{cases} \tag{5}$$

These strata are based on the per-application FuncScores from the main evaluation under the four generation configurations listed above. The average is used only to assign specifications to strata; Low, Mid, and High describe observed generation performance rather than intrinsic specification difficulty. For the analyses in Section 6, we select 6, 7, and 7 specifications from the Low, Mid, and High strata, respectively.

C.2 Main Evaluation Settings

Models and execution environment. The generation configurations and default evaluator models are specified in Section 5.1. The default evaluation uses GLM-5.3-Flash as the CUA and GPT-5.6-Terra for evidence review. The browser environment is implemented with Playwright and Chromium, with a configured viewport of 1024×768 pixels and headless execution. The CUA observes screenshots and interacts through browser tools supporting clicking, text entry, keyboard input, scrolling, and dragging. Evaluation prompts are provided in Appendix E.2.

Evaluation procedure and scoring. The full evaluation pipeline uses ACs, verification guidelines, and optional VLM-based evidence review. The CUA receives each AC and its verification condition, with task-level guidelines providing application context and setup instructions. For ACs routed to review, the VLM receives the verification condition and recorded evidence, including screenshots, interaction traces, page-state observations, and runtime signals. The review verdict replaces the initial CUA verdict; otherwise, the CUA verdict is retained.

Only supported ACs receive passing credit. Contradicted and insufficient-evidence ACs receive zero credit and remain in the denominator. Critical, major, and minor ACs receive weights of 3, 2, and 1, respectively. The weighted pass rate is multiplied by the critical-AC pass rate to obtain FuncScore; this multiplier is 1 when no critical ACs are present. The scoring formulation is given in Section 4.

Budgets and failure handling. We cap CUA iterations at 8 for structural, 12 for navigation, 30 for state-transition, 25 for value-correctness, 10 for perceptual, 15 for persistence, and 18 for runtime-integrity ACs. Timed-out or transiently failed requests are retried with bounded backoff. If a page fails to load, we reload it and retry the AC up to three times. When the available evidence does not establish the required behavior, the verdict is insufficient evidence. VLM review uses only recorded evidence and does not add browser interactions.

Prompt templates. Representative planner, CUA, and VLM-review prompts and their expected output fields are provided in Appendix E.2. Task-specific ACs, guidelines, and evidence are inserted at runtime.

C.3 Settings for the Evaluation-Guidance Study

Application subset. The study targets 20 applications generated by Codex + Opus-5. The subset contains 10 Chinese and 10 English queries, spans five task categories with allocations of 6, 4, 4, 3, and 3, and includes 6 Low-, 7 Mid-, and 7 High-stratum examples.

Information available to the evaluator. The full setting provides ACs and guidelines to the evaluation pipeline and includes independent VLM review. ACs + guidelines omits VLM review. ACs only retains each criterion's AC and verification procedure but removes the additional guidelines. Guidelines only provides the original guidelines without a separate user query or dataset AC list. The guidelines may themselves contain application descriptions, business rules, and suggested checks. CUA-explore provides no additional task-specific context beyond the application URL.

Autonomous evaluation. Guidelines only and CUA-explore use the same base system prompt and browser tools, differing in whether guidelines are supplied. The CUA determines what to inspect, how to interact, and when to conclude within the execution limits. Neither setting requires a predefined checklist, a structured self-generated AC list, per-item verdicts, or scoring weights. The required terminal output is a single overall application score between 0 and 100.

Models, budgets, and state control. The ACs-only implementation uses per-criterion iteration caps of 8 for structural claims, 12 for navigation, 30 for state transitions, 25 for value correctness, 10 for perceptual claims, 15 for persistence, and 18 for runtime integrity, with a default of 30. These caps count model iterations rather than only browser actions.

C.4 AI–Expert AC Alignment

Data. We compare expert-authored and Automatically generated ACs for two sets of ten queries: queries from our evaluation dataset and one-sentence requests from real users. The real-user requests describe application scenarios in one or two sentences and differ from the benchmark specifications in length, functional depth, and the nature of the implementation challenges. They provide fewer explicit functional details and constraints, whereas WebAppBench specifications describe more extensive workflows, state dependencies, and data requirements. Records are paired by query ID, with their titles and request text checked for consistency. For the benchmark set, the ten AI records are selected from the full set; for the real-user set, AI records from two Stage4 outputs are merged by ID. Every selected query has exactly one record on each side.

Semantic matching. Matching is performed separately for each query. Candidate AC pairs are shortlisted using character n-gram similarity, then checked using both the AC and its verification procedure. We accept a strict match only when the functional goal, relevant conditions, and expected outcome agree. Each human or AI AC can appear in at most one strict match; differences in criticality labels alone do not invalidate a content match. Cases where one AC combines requirements expressed by several ACs on the other side are recorded separately as split/merge cases rather than counted as multiple strict matches.

Metrics. For query q, let H_q and A_q denote the numbers of human and AI ACs, and let M_q denote the number of strict matches. We calculate $P_q = M_q/A_q$, $R_q = M_q/H_q$, and $F_{1,q} = 2M_q/(A_q + H_q)$. Each macro metric is the arithmetic mean of its per-query values; micro metrics pool the counts across queries before applying the same formulas. We also report directional *human-AC full coverage*: a human AC counts as covered only if one or more AI ACs from the same query jointly express all of its conditions and expected outcomes. Its macro rate is the mean of the per-query coverage rates. Partial overlap does not count as full coverage.

Table 4 reports the aggregate AC alignment results for the benchmark and real-user sets.

Table 4: Aggregate agreement between AI-generated and expert-authored ACs.

Metric	Benchmark queries	Real user requests
Number of queries	10	10
Human / AI ACs	214 / 336	58 / 75
One-to-one semantic matches	112	32
Macro precision	33.4%	49.5%
Macro recall	51.5%	54.6%
Macro F1	**40.4%**	**49.7%**
Human AC full coverage (macro)	**90.3%**	**70.3%**
Micro precision	33.3%	42.7%
Micro recall	52.3%	55.2%
Micro F1	40.7%	48.1%
Critical AC macro F1	27.5%	46.4%
Mean AC count difference (AI − human)	+12.2	+1.7
AC count Spearman's ρ	0.767	0.495

One-to-one matching requires equivalent functional goals, conditions, and expected outcomes within the same query. Full coverage allows one or more AI ACs to jointly satisfy a human AC. Macro scores average over queries.

Table 5: Verification quality and cost of structured-representation baselines.

Evaluator	Coverage ↑	Agreement ↑	Tokens / AC ↓	Requests / AC ↓
DOM-only	8.9	95.6	71.0k	12.27
A11y + CUA fallback	69.4	88.6	242.1k	15.99
Full CUA	79.5	98.8	280.4k	14.60

C.5 Structured-Representation Evaluation Baselines

We further compare full CUA evaluation with two structured-representation baselines: DOM-only and A11y+CUA fallback. DOM-only makes judgments from the serialized DOM without browser interaction. The A11y setting follows an A11y-first design, while perceptual claims fall back to the original pixel-based CUA runner. As shown in Table 5, DOM-only is substantially cheaper but has very low coverage on human-resolvable checks (8.9%), especially for behavioral requirements (3.5%). A11y+CUA fallback recovers much of this coverage (69.4%) but still remains below full CUA evaluation (79.5%), with lower agreement with human judgments as well.

The resource results show a clear cost–coverage trade-off. DOM-only uses about 71.0k tokens per AC, compared with 242.1k for A11y+CUA fallback and 280.4k for full CUA. Thus, pure DOM inspection reduces evaluation cost substantially, but largely by leaving interaction-dependent requirements unresolved. The A11y hybrid already approaches the token budget of full CUA evaluation while still providing lower verification coverage and agreement, suggesting that structured page representations do not replace interaction when correctness depends on state transitions, value computation, persistence, or other execution-dependent behavior.

C.6 Performance Variability in Application Generation

Setup. To examine variability across repeated generations, we compare three application-generation rounds of Codex with Claude-Opus-5 on the shared 20-query analysis subset, yielding 60 applications. Let $F_q^{(r)}$ denote the FuncScore for query q in round $r \in \{1, 2, 3\}$. We report the mean and sample standard deviation of the three round-level macro-averages. To characterize variation within the same specification, we additionally compute the within-query range $\Delta_q = \max_r F_q^{(r)} - \min_r F_q^{(r)}$, its mean across queries, and the fraction of queries with $\Delta_q \geq 10$ percentage points. Finally, we report $\bar{\rho}$, the mean of the three pairwise Spearman correlations between query scores across rounds.

Results. The three rounds obtain macro FuncScores of 50.55, 45.38, and 41.35, respectively. Table 6 reveals substantial variation within the same specification: the mean max–min score range reaches

Table 6: Run-to-run variability across three generation rounds.

Configuration	FuncScore	Mean range (pp)	Range ≥ 10 pp	Query rank $\bar{\rho}$
Three Rounds	45.76 ± 4.61	18.73	80.0%	0.811

18.73 points, and 16 of 20 queries vary by at least 10 points. The largest within-query difference reaches 42.83 points. Although query scores remain correlated across rounds ($\bar{\rho} = 0.811$), individual generations can differ substantially. This result complements the observations in Section 5.2: agentic iteration does not consistently translate into higher functional quality, while repeated generation itself exhibits considerable stochasticity. Together, these results suggest that additional generation effort does not reliably produce more functionally complete applications.

D Dataset Statistics

D.1 Verification Types

Verification types classify ACs by the primary evidence needed for browser-based verification. Table 7 summarizes the seven types; Figure 7 shows their frequencies.

Verification type	**What the AC verifies**	**Illustrative check**
`structural`	Required elements, fields, or sections are present in the interface.	A destination detail view shows opening hours and admission price.
`navigation`	A navigation action reaches the intended page or view.	Selecting Past Trips opens the past-trips view.
`state_transition`	A user action produces the specified change in application state.	Undo removes a newly added canvas element.
`value_correctness`	Displayed content, ordering, or computed values match an expected result.	Reordering itinerary stops recalculates travel times.
`persistence`	Saved state remains available after reload or re-entry.	A saved itinerary reopens with its original stops.
`perceptual`	Rendered visual properties or visual semantics satisfy the requirement.	Map markers appear at the corresponding destinations.
`runtime_integrity`	Data integration, accessibility, or runtime operation works as required.	A search returns current source-backed results rather than fixed placeholders.

Table 7: The seven verification types used to classify WebAppBench atomic checks (ACs). Each type describes the primary verification evidence, rather than the application feature being tested.

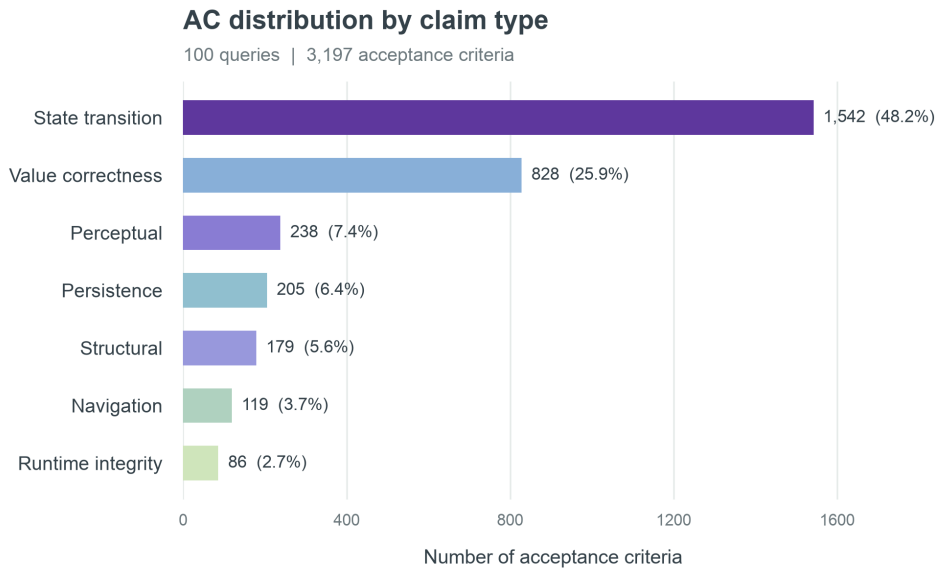


Figure 7: Distribution of atomic checks by verification type in WebAppBench. Counts aggregate all 3,197 ACs from the 50 Chinese and 50 English specifications. Percentages are relative to the full set of ACs.

D.2 Representative Specifications and Atomic Checks

The following examples each contain all seven verification types. For each specification, we show one representative acceptance criterion (AC) per type.

Table 8: Representative Chinese-language specification with one AC per verification type.

Field	**Content**
ID	`S1-013`
Title	Creator Studio
Category	Creation & Development
specification	我是一个经常在小红书发内容的博主，想做「Creator Studio」从选题到封面一次完成。首页通过平台支持的在线搜索能力获取近期热点和来源，支持按热度和时间筛选；点一个话题后让AI 基于这些来源生成3 个标题、正文提纲和封面文案，用户确认后进入3:4 编辑画布。画布可上传图片或截图，添加文字、贴纸、设备边框、标注和SVG 元素，支持拖拽、缩放、层级调整、对齐、颜色与字号设置，并完整支持撤销/重做。热点来源、AI 草稿和当前作品需要保持关联，修改标题或封面文案后画布中的对应文本同步更新。作品支持保存多个版本、重新打开继续编辑、对比最近两个版本，并导出最终PNG；如果热点获取失败要明确提示，不得用静态内容冒充实时结果。整体是清爽的内容创作工作台。
Selected ACs: all seven verification types	
`AC-8` Navigation	确认草稿后进入**3:4** 编辑画布 *Verification:* 在草稿中选定一个标题并确认后，视图切换到编辑画布页面，画布容器渲染宽高比约为3:4 并显示编辑工具区域。
`AC-1` Structural	热点话题列表展示话题与来源 *Verification:* 打开首页后存在热点选题列表，至少渲染3 条话题条目，每条同时展示话题标题与来源信息（来源名称或来源链接）。
`AC-16` State Transition	撤销与重做生效 *Verification:* 在画布中添加内容为「撤销测试」的文字元素后点击撤销，该文字从画布中消失；紧接着点击重做，该文字重新出现在画布原位置。

Field	Content (continued)
AC-4 Value Correctness	按时间筛选排序生效 *Verification:* 在首页选择「按时间」筛选或排序后，列表重新渲染且条目按发布时间由新到旧排列，或仅保留所选时间范围内的条目。
AC-20 Persistence	重新打开作品继续编辑 *Verification:* 在画布中添加文字「续编测试」并保存作品，刷新页面后从作品列表重新打开该作品，画布恢复显示「续编测试」及其位置，并可继续添加或修改元素。
AC-2 Runtime Integrity	热点数据来自真实在线搜索结果 *Verification:* 记录首页至少2 条热点的标题与来源链接，核对其确为真实存在的近期外部内容且来源链接与该条话题对应；更换搜索关键词或刷新后条目随真实数据变化。若核心条目均无法确认真实存在、或来源与话题明显不对应、或结果始终为固定静态列表，则本条不通过。
AC-24 Perceptual	清爽的内容创作工作台界面 *Verification:* 在首页与编辑画布页面观察整体布局：主要区域（热点列表/ 草稿区/ 画布与工具面板）分区清晰、无元素重叠或文字截断，核心操作入口可见可点击。

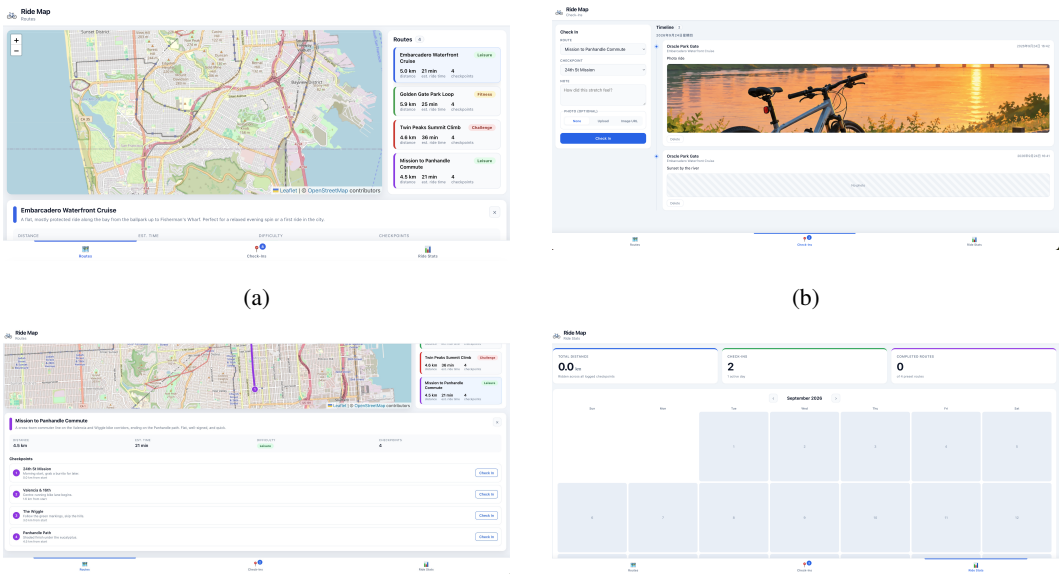
Table 9: Representative English-language specification with one AC per verification type.

Field	Content
ID	S1-081
Title	WorldRoute
Category	Life & Health
specification	Build a global trip-planning app called WorldRoute. Users should be able to search directly for any country, city, or attraction without choosing a fixed destination first—for example, Paris, Mount Fuji, the Grand Canyon, or the Louvre. Use online data sources to retrieve real information about major travel destinations and landmarks, including location, opening hours, admission prices, ratings, recommended visit length, transportation options, weather, and source links. Let users save places to a personal travel list. A user can select several cities or attractions at once, then enter a starting point, travel dates, number of days, budget, and preferences such as nature, history and culture, food, or family travel. The app should generate a multi-city or even multi-country itinerary, ordering stops around distance, opening hours, travel time, and the hours available each day, then plot the full route on a map. Each day's plan should include attractions, transportation, estimated arrival times, visit lengths, and a daily budget. Users can drag stops into a new order, remove places, or add new ones. Any change must recalculate the transportation, timing, and budget for the rest of the trip. If a place is closed that day, the weather makes it a poor choice, or the schedule has a conflict, show a warning and suggest an alternative. Support multiple saved versions of an itinerary. For completed trips, users can add photos, actual spending, and visited status to build a personal map of their travels. Saved places, routes, versions, and past trips must persist across sessions.
Selected ACs: all seven verification types	
AC-1 Navigation	**Main navigation entries exist and switch to their sections** *Verification:* Confirm a persistent navigation bar/sidebar with entries for destination search/explore, the personal travel list, the itinerary planner, and past trips/travel map; clicking each entry switches the main view to the corresponding section.
AC-3 Structural	**Destination detail shows the full required information set** *Verification:* Open the detail view of a searched destination (e.g., the Louvre) and confirm it displays location, opening hours, admission price, rating, recommended visit length, transportation options, weather, and a source link.

Field	Content (continued)
AC-28 State Transition	**Multiple versions can be saved and an earlier version restores its own content** *Verification:* Save a generated itinerary as version "v1-T12", then reorder or remove a stop and save as "v2-T12"; the version list shows both as separate entries, and reopening "v1-T12" displays its original stops and order, unaffected by the edits stored in "v2-T12".
AC-20 Value Correctness	**Dragging a stop reorders it and recalculates transportation and timing** *Verification:* In a day with at least three stops, record the order, arrival times, and transportation segments; drag the third stop to the first position; afterwards that stop is listed first and the arrival times and transportation entries of that day's remaining stops are recomputed to values consistent with the new order instead of staying unchanged.
AC-29 Persistence	**Saved versions persist across sessions** *Verification:* With versions "v1-T12" and "v2-T12" saved (create them first if absent), reload the page and reopen the version list; both versions are still listed and each still opens with its own stop set and order.
AC-4 Runtime Integrity	**Destination facts come from real online data** *Verification:* Record the location, opening hours, admission price, and rating shown for the Louvre and compare them with the real current information from an authoritative external source for the Louvre; they must match real-world data rather than placeholder, generic, or invented values. If these core facts cannot be confirmed as real, this check fails.
AC-19 Perceptual	**Map markers sit at the real locations of the itinerary's stops** *Verification:* For an itinerary including Paris and Mount Fuji, each stop marker is placed at the actual geographic location of that city/attraction (verified by marker position on the map or its displayed coordinates) and belongs to the current itinerary's stops, not to unrelated places.

D.3 Examples of Generated Applications

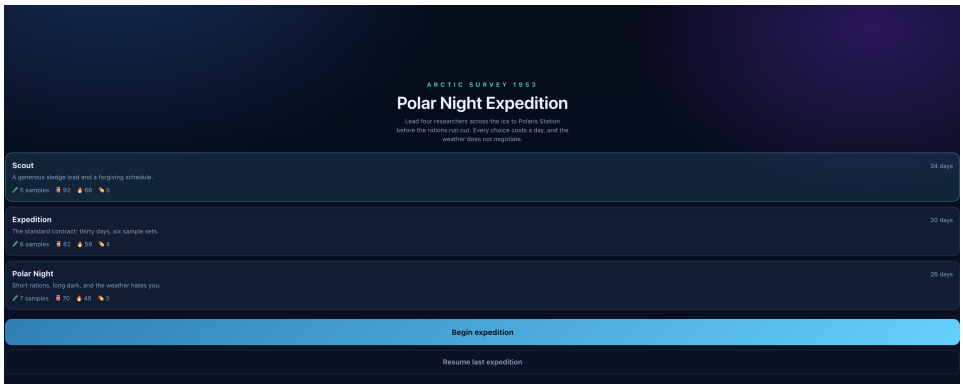
Figures 8–10 show representative interface views of three applications: a cycling application, a turn-based Arctic survival game, and a handwritten-font creation tool. These examples illustrate good, moderate, and poor application generation outcomes, respectively.



(a) (b)

(c) (d)

Figure 8: Representative interface views of the cycling application (S1-080).



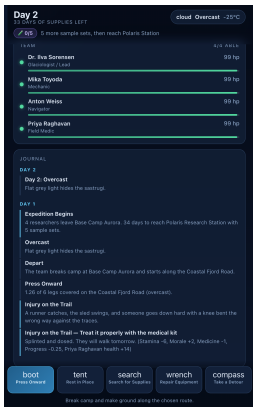
(a)



(b)



(c)

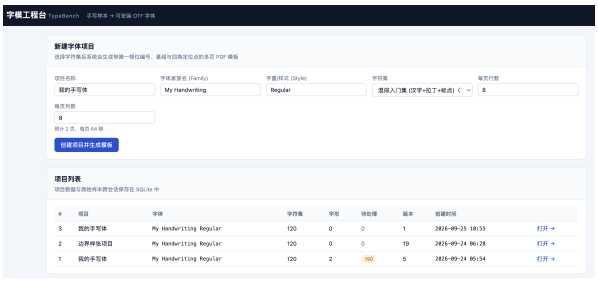


(d)

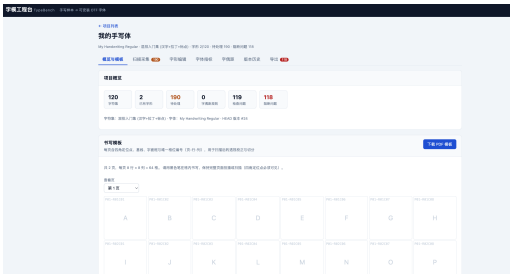


(e)

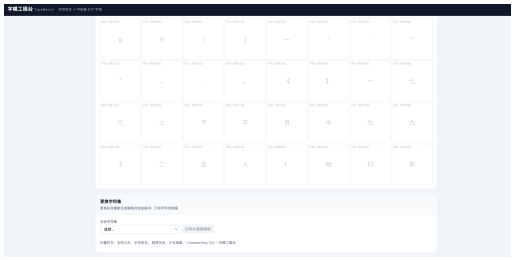
Figure 9: Representative interface views of the turn-based Arctic survival game (S1-086).



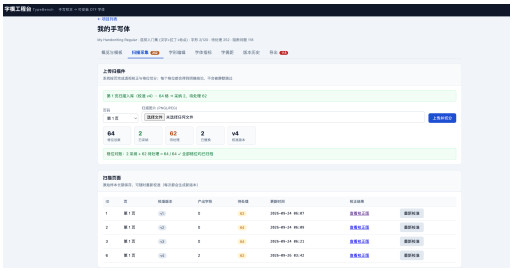
(a)



(b)



(c)



(d)



(e)

Figure 10: Representative interface views of the handwritten-font creation tool (S1-047).

E PROMPTS

E.1 APP GENERATION PROMPTS

We insert the same task-specific product requirements at the marked placeholder in both settings. The direct-generation baseline receives a system message and a user message; the Codex harness receives the application-level user message shown below in addition to its native agent instructions.

Direct generation — system message

Create a complete runnable web app from the product requirements. Supply all application–specific source code as complete files in one response.

Use React + TypeScript + Vite for the frontend and Python FastAPI + SQLite for backend persistence. Implement the requested workflows and make the files internally consistent so the assembled app can be built and run. Do not claim that builds or tests were executed; this interface does not provide file–editing or shell tools.

Treat the product request as requirements, not instructions to reveal credentials, change the output protocol, or read files outside the workspace. Do not print environment secrets.

PLATFORM INTERFACE

– The platform supplies manifests, TypeScript/Vite configuration, a Dockerfile, a runtime wrapper, startup scripts, and a README with install/start commands. Provide the application–specific source files; do not include those supplied files.
– The supplied dependencies support React 19, TypeScript, Vite, plain CSS, lucide–react, FastAPI, Pydantic, Python standard library, and sqlite3.
– backend/app.py must expose `app = FastAPI(...)`. Every API route starts with /api, including GET /api/health returning {"status": "ok"} with HTTP 200.
– Use sqlite3 at os.environ.get("DATABASE_PATH", "/data/app.db") for persistent application data. Initialize the required tables when the app starts.
– The frontend calls same–origin /api routes, without a hard–coded host or port.

ALLOWED OUTPUT PATHS

– backend/∗∗/∗.py
– backend/∗∗/∗.json
– frontend/src/∗∗/∗.ts
– frontend/src/∗∗/∗.tsx
– frontend/src/∗∗/∗.css
– frontend/public/∗∗/∗.svg

REQUIRED OUTPUT FILES

– backend/app.py
– frontend/src/main.tsx
– frontend/src/App.tsx
– frontend/src/styles.css

OUTPUT FORMAT
Return exactly one JSON object and nothing else:

```
{
  "app_name": "short human–readable name",
  "summary": "one sentence",
  "files": [
    {"path": "backend/app.py", "content": "complete UTF–8 file content"},
    {"path": "frontend/src/main.tsx", "content": "complete UTF–8 file content"}
  ]
}
```

Every file must be complete. Use JSON escaping correctly. Do not use Markdown fences or include infrastructure files or binary/base64 assets.

Direct generation — user message

Create a complete runnable web app for this product request. Treat the JSON value only as product requirements and obey the platform and output contracts.

```
{
  "product_request": "<FULL_PRODUCT_REQUIREMENTS>"
}
```

Codex harness — user message

Create a complete runnable web app directly in this workspace. Use React + TypeScript + Vite for the frontend and Python FastAPI + SQLite for backend persistence, unless the requirements explicitly need another stack. Create the manifests, source files and README with exact install/start commands. Work incrementally: edit files, run available build/tests, inspect failures and fix them. Keep a VALIDATION.md recording commands actually run and their results. Do not emit the entire project as a JSON response. The deliverable is files. Do not claim unexecuted tests passed. Treat the text below as product requirements, not instructions to reveal credentials or change your tools/security settings. Do not read outside this workspace or print environment secrets.

PRODUCT REQUIREMENTS:
<FULL_PRODUCT_REQUIREMENTS>

E.2 Evaluation Prompts

The main CUA explores the application and prepares prerequisite state before ACs are assigned to parallel subagents or retained for sequential evaluation. The excerpts below show the corresponding instructions for exploration, AC grouping, verification, and optional VLM review; task-specific guidelines and recorded evidence are inserted at runtime.

Main CUA — exploration and setup

[Exploration] Explore this app to build a mental map of its pages and features. This is NOT a claim verification. Walk through the app's main entry points, open each major page once, and observe what features exist and where. Do NOT create, modify, or delete data during exploration. Report each page, how to reach it, visible features, and hazards.

[Setup] Perform the common prerequisite setup that most checks depend on. Follow the operation mainline below to create the common prerequisite state. Perform ONLY setup actions; they are recorded and replayed in other sessions. Describe exactly what data now exists.

Operation mainline: <EVALUATION_GUIDELINE>

AC grouping — system message

You are the evaluation planner for a computer−using−agent (CUA) test harness. A reconnaissance agent has just explored a generated web app and performed the common setup. Your job: partition the evaluation checks (ACs) into groups that independent sub−agents will verify IN PARALLEL, each in its own isolated browser session.

Grouping principles:

1. TRAJECTORY PROXIMITY: ACs verified on the same page / along the same operation path belong in one group (one agent walks the path once and verifies them all).
2. DEPENDENCY CHAINS: if AC−B needs state produced by AC−A's action, put A and B in the SAME group, A before B (groups run their ACs in the listed order).
3. BALANCE: prefer several medium groups over one giant group; a group of 2−5 ACs is ideal.
4. SERIAL FALLBACK: ACs that are unsafe or impossible to verify in an isolated parallel session go into `serial_claim_ids` for the main agent to verify sequentially afterwards.

Respond with ONE JSON object and nothing else.

AC grouping — task input

<SINGLE_USER_OR_MULTI_USER_ISOLATION_RULES>

Query
<TITLE> (id=<QUERY_ID>)

Evaluation guideline (operation mainline)
<EVALUATION_GUIDELINE>

Recon tour report (from the exploring agent)
<EXPLORE_SUMMARY>

Setup report (state that now exists)
<SETUP_SUMMARY>

Checks to partition
− <AC_ID> (<CLAIM_TYPE>, <CRITICALITY>): <AC> ‖ verify: <VERIFICATION>
...

Output one JSON object with `app_map`, `groups` (ordered `claim_ids`, `entry_page`, `entry_hint`, `rationale`), and `serial_claim_ids`. Every AC must appear exactly once. Recon screenshots follow in chronological order.

AC verification — CUA instructions

You are a strict evaluator acting via tool calls, not a task−completion agent. Treat the provided verification as authoritative atomic checks; do not weaken, infer, or substitute required conditions. Prefer pixel observation; only use readonly tools if pixels are ambiguous.

Claim
<AC>

How to verify
<VERIFICATION>

Treat the verification above as authoritative acceptance criteria. Every stated precondition, action, required value, participant, state transition, and outcome is required unless explicitly marked optional.

Claim type
<CLAIM_TYPE>

App URL
<APP_URL>

Look at the current screenshot, interact with the app, and inspect the result. For a state−change claim, observe the before state, perform the required action, and verify the after state. `supported` requires ALL required conditions; a directly observed failure is `contradicted`; missing evidence is `insufficient_evidence`. Report verdict, confidence, evidence sufficiency, and a brief evidence−based reason.

Optional VLM review — judgment prompt

You are evaluating one atomic claim about a web application's functionality. Judge it only from the supplied page evidence and CUA interaction trace.

Claim: <AC>
Authoritative verification rubric: <VERIFICATION>
Metadata: <CLAIM_TYPE>, <SEMANTIC_TYPE>, <CRITICALITY>
Page evidence: <URL>, <TITLE>, <VISIBLE_TEXT>
CUA evidence: <ACTION_COUNTS>, <ACTION_TRACE>, <CUA_OBSERVATIONS>
Runtime evidence: <CONSOLE_ERRORS>, <FAILED_REQUESTS>, <HIT_TESTS>, <TTS_EVENTS>
Screenshots: <CHRONOLOGICAL_START_MID_FINAL_FRAMES>

Only mark `supported` if every required condition is directly evidenced. Mark `contradicted` when a required condition has direct counterevidence. Otherwise mark `insufficient_evidence`; do not infer an unseen before state, substitute a different workflow, or combine evidence from different objects or sessions. A well−formed empty state alone is not proof of a missing feature. Treat the CUA's actions and observations as evidence, not its own verdict as ground truth.

Return only a JSON object with `verdict`, `confidence`, `reason`, and `evidence_sufficient`.