WEBPROGRAMBENCH: CAN CODING AGENTS RECONSTRUCT LIVE APPLICATIONS FROM SCRATCH VIA BLACK-BOX INTERACTION?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents have recently achieved substantial progress in fixing bugs and implementing features within existing repositories, yet building complete real-world applications from scratch remains an open challenge. Existing software-development benchmarks assess only partial aspects of this capability through requirement-driven generation and screenshot-based interface cloning. Their evaluations emphasize visual similarity, selected functional checks, or model-based judgments, leaving room for shortcut solutions that pass evaluation. To address these limitations, we introduce WebProgramBench, the first execution-based benchmark for evaluating whether LLMs can reconstruct real-world live applications from scratch through black-box interaction. WebProgramBench requires agents to actively explore a live reference application and reconstruct its behavior in a deployable implementation, without access to its source code. To evaluate task performance, we introduce a novel cross-layer protocol that jointly verifies interface, service, and persistence behavior, preventing frontend-only mocks from passing. The benchmark comprises 20 real-world applications spanning four domains, with reference implementations totaling nearly one million lines of code. To scale this evaluation, our fully automated pipeline generates 1,254 verified executable tests with 20,851 assertions. Our experiments across five leading LLMs show that frontier LLMs remain far from complete real-world application reconstruction. Even the best-performing model achieves near-complete reconstruction on only three applications, only one of which is fully reconstructed. Together, these results establish WebProgramBench as a rigorous benchmark for measuring LLM capabilities in autonomous, closed-loop software engineering.

1 INTRODUCTION

As large language models become more capable, coding agents are increasingly expected to take end-to-end responsibility for delivering complete software systems from high-level intent rather than assisting with isolated implementation steps (Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025). A critical frontier is whether they can reliably engineer production-scale software systems from scratch over long horizons. This requires sustained, closed-loop engineering across system design, environment configuration, implementation, and verification through execution feedback.

Earlier coding-agent benchmarks primarily framed software engineering as bounded repair or feature-development tasks within established repositories (Jimenez et al., 2024; Miserendino et al., 2025). Recent benchmarks have begun to move beyond localized modification by asking agents to construct entire libraries or repositories, starting from scaffolded or empty workspaces (Zhao et al., 2025; Ding et al., 2026). However, these benchmarks often provide detailed repository layouts, function and class signatures, and preconfigured runtime environments. Such scaffolding predetermines foundational engineering choices, thereby limiting the assessment of agents' ability to design architectures, decompose systems into coherent modules, and configure and manage runtime environments. A parallel line of work evaluates website generation from natural-language requirements and screenshots (Si et al., 2025; Chen et al., 2026). However, this line of work remains limited in both task definition and evaluation. At the task level, static screenshots convey visual appearance but cannot specify state-dependent behavior, and a small set of screenshots captures only a narrow slice

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.

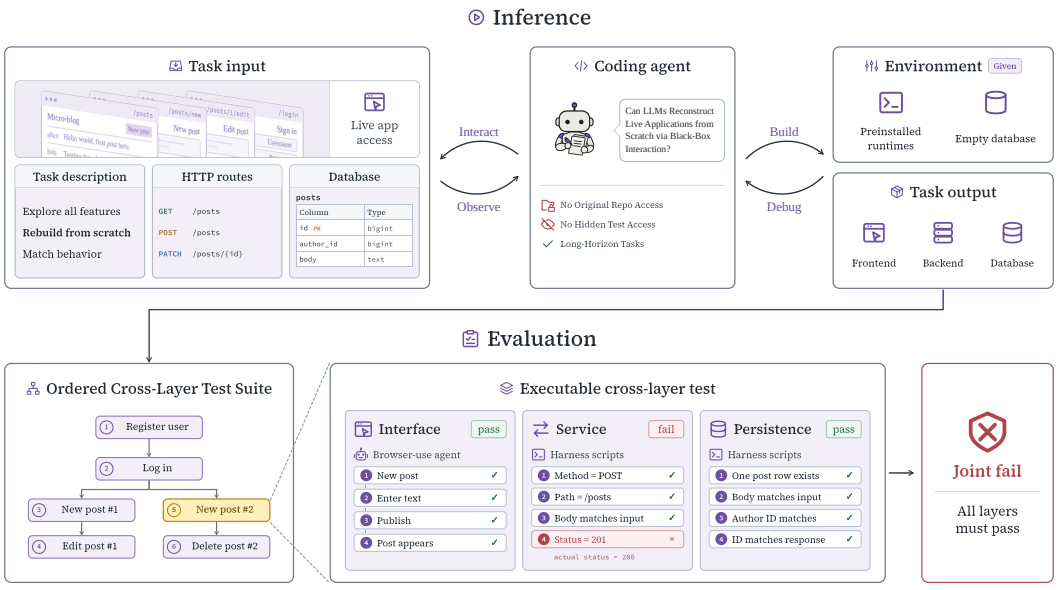

Figure 1: Overview of WebProgramBench. Given access to a live reference application and endpoint and database contracts, a coding agent explores the reference behavior and reconstructs a deployable full-stack application from scratch within an isolated implementation workspace. Hidden ordered tests evaluate behavioral equivalence across the interface, service, and persistence layers.

of the reference application's behavior. These benchmarks also typically prescribe the technology stack and runtime environment, leaving agents' ability to select technology stacks and configure and operate runtime environments outside evaluation (Lu et al., 2026).

At the evaluation level, these benchmarks often rely on human-authored functional tests executed by GUI agents and scored by model judges, making this human-in-the-loop process difficult to scale (Desai et al., 2026; He et al., 2026). More fundamentally, these evaluations report functional success on predefined tests but provide no behavioral coverage metric for assessing test-suite adequacy (Jain et al., 2025). Consequently, a generated website may satisfy the benchmark while remaining a partial demonstration rather than a production-ready system (Bian et al., 2025).

To address these limitations, we introduce WebProgramBench, the first benchmark to require coding agents to reconstruct production-scale applications from scratch through black-box interaction with live references. Each task exposes a live reference application and its endpoint and database contracts while withholding the source code. Beyond these external contracts, the agent is required to independently make every engineering decision and handle every implementation detail. WebProgramBench asks whether coding agents can infer the behavior of a production-scale application through black-box interaction and independently engineer a behaviorally equivalent system.

To answer this question, WebProgramBench evaluates each reconstructed application using an ordered suite of executable cross-layer tests. Each test targets one endpoint behavior, triggers it through browser interaction, and jointly verifies its effects across the interface, service, and persistence layers. This strict cross-layer verification prevents shortcut solutions that reproduce only a superficial subset of the required behavior from passing while leaving the concrete implementation unconstrained. Moreover, our executable cross-layer tests provide fine-grained evidence of behavioral equivalence rather than only a terminal result. Unlike prior benchmarks that rely on expert annotation, WebProgramBench automatically constructs test suites in four phases. Coding agents first identify endpoint behaviors and their dependencies, then plan stateful executions, synthesize executable cross-layer tests, and iteratively refine them against the live reference application. Together, these phases scale strict, high-coverage executable evaluation across diverse real-world websites.

We construct WebProgramBench from 20 real-world open-source websites spanning four application domains, 11 language categories, and nearly one million lines of code. Across these websites,

our automated pipeline generates 1,254 executable cross-layer tests comprising 20,851 fine-grained assertions across the interface, service, and persistence layers.

We conduct comprehensive experiments across five leading open-weight and proprietary LLMs and various agent harnesses. Our results demonstrate that frontier LLMs remain far from complete real-world application reconstruction, as even the best-performing LLM fully reconstructs only one application and reaches the almost-solved threshold on only three. More fundamentally, we find that current LLMs lack the compositional capability to integrate behavior both vertically across system layers and horizontally across state-dependent workflows. In summary, our primary contributions are as follows:

- We introduce WebProgramBench, the first benchmark to require coding agents to reconstruct production-scale stateful web applications from scratch by inferring behavior from live reference applications. This task establishes black-box behavioral reconstruction as a critical frontier for sustained, closed-loop software engineering agents.
- We introduce a novel executable cross-layer evaluation protocol that measures full-stack behavioral equivalence through joint interface, service, and persistence verification, preventing shortcut solutions. We also introduce a novel automated pipeline that generates and iteratively repairs executable tests through live execution feedback, producing high-coverage validated tests at scale without expert annotation.
- We conduct a comprehensive empirical evaluation of five leading open-weight and proprietary LLMs. Our results demonstrate that frontier LLMs remain far from complete real-world application reconstruction.

2 Related Work

Repository-level benchmarks. SWE-bench (Jimenez et al., 2024) established issue resolution in real-world repositories as a central evaluation setting for coding agents. Subsequent benchmarks broadened this setting through visual inputs (Yang et al., 2025), multilingual repositories (Zan et al., 2025), real-world freelance tasks (Miserendino et al., 2025), broader task coverage (Xu et al., 2025), and continuous software evolution (Deng et al., 2026). However, even across longer development horizons, these benchmarks remain anchored to existing implementations and do not evaluate whether agents can autonomously implement and verify complete systems from scratch.

Program and repository generation. Recent benchmarks move beyond localized patching toward long-horizon, holistic software construction. Some evaluate repository generation from textual specifications (Zhao et al., 2025; Ding et al., 2026), while others require agents to reproduce an existing implementation from either a compiled executable or an API specification (Yang et al., 2026; Zhang et al., 2026). WebProgramBench advances this paradigm to real-world application reconstruction through interaction with a live GUI, bringing autonomous exploration, system design, implementation, and verification into a unified software-engineering task.

Application development. Image-conditioned benchmarks have progressed from static webpage generation to interaction inference from coherent screenshot sets (Si et al., 2025; Chen et al., 2026). Beyond interface generation, recent benchmarks broaden the scope to interactive websites and full-stack systems (Lu et al., 2025; He et al., 2026; Lu et al., 2026; Desai et al., 2026). However, even when the resulting artifacts are executable, the target behavior is specified through screenshots or textual requirements rather than discovered from a running application. WebProgramBench instead poses long-horizon reconstruction from a live reference application and applies strict, comprehensive verification that jointly evaluates behavior across the interface, service, and persistence layers. A detailed benchmark comparison is provided in Appendix Table 6.

3 WebProgramBench

3.1 Task Formulation

As shown in Figure 1, given access to a live reference website, a concise task description, and endpoint and database contracts, a coding agent must build a deployable full-stack application from

scratch within an isolated implementation workspace to reproduce the functional behavior of the reference website. The contracts expose only the minimum structural information needed for executable evaluation: required HTTP methods and paths, together with database table and column names, data types, and primary keys. The reference application's source code and database remain inaccessible. The complete task prompt is provided in Appendix L.

Our evaluation targets behavioral equivalence between the candidate and reference applications. Given matched initial states, semantically equivalent user actions should produce equivalent functional interface outcomes, HTTP exchanges, and persistent database effects. To enable rigorous and reproducible evaluation, we introduce executable cross-layer tests. Each test exercises one endpoint behavior through browser interaction and jointly verifies its effects across the interface, service, and persistence layers. Each reference website is paired with an ordered suite of such tests designed to cover its endpoint behaviors and state-dependent workflows. The entire test suite remains hidden from the agent throughout task execution.

3.2 Benchmark Construction

Automated Task Construction. Given a selected reference repository, a coding agent uses framework-specific tools to extract the endpoint metadata and database schema of the reference application. The coding agent then converts the extracted metadata into sanitized endpoint and database contracts. These contracts expose the structural constraints required for reconstruction while withholding the reference implementation. After sanitization, WebProgramBench packages the contracts with a concise task description and the required account credentials into a complete task, whose full specification is provided in Appendix L. The extracted endpoint metadata and database schema are also reused as the structural basis for subsequent behavior analysis and executable evaluation.

Executable Cross-Layer Test Suites. WebProgramBench evaluates each reconstructed application using an ordered sequence of cross-layer tests. The ordering preserves the identities, resources, and persistent state required by subsequent tests. Each executable test targets an endpoint behavior, triggers it through browser interaction, and jointly verifies its effects across three layers: *1) Interface layer.* We formulate interface verification as a step-wise agentic procedure composed of ordered semantic browser interactions and observable functional checks. At evaluation time, a GUI-agent verifier executes the ordered procedure and records the outcome of each step. Unlike a goal-driven protocol that provides only a terminal judgment and may accept unintended paths to the same outcome, step-wise verification constrains the intended workflow, yields fine-grained evidence of functional correctness, and improves evaluation reproducibility. Moreover, the semantic step specification decouples verification from implementation-specific test code, allowing the same procedure to evaluate different frontend implementations. *2) Service layer.* For service-layer evaluation, we define the HTTP exchange induced by each browser interaction as the observable boundary of backend behavior. Service-layer assertions jointly verify the request and response, evaluating endpoint-level input-output equivalence without constraining the internal implementation of the reconstructed application. Request assertions cover the method, path, query parameters, and body, while response assertions cover the status, headers, and body. This protocol-level abstraction supports uniform evaluation across diverse endpoint behaviors, from conventional form submissions and structured API calls to binary file uploads and downloads. *3) Persistence layer.* The persistence-layer assertions verify the database state produced by the same UI interaction. Specifically, the assertions verify not only whether records are created, updated, or deleted, but also whether relevant column values and relations to other records match the expected state. Together, the executable assertions across all three layers form a joint behavioral oracle. The evaluation harness executes each test and records the outcome of every assertion. Each assertion returns `passed`, `failed`, or `skipped`. When an earlier failure makes a dependent property unevaluable, the corresponding assertion is marked as `skipped` rather than `failed`, avoiding cascaded failures from a single root cause. A cross-layer test passes only when all required assertions pass. This joint pass criterion prevents a candidate application from passing by reproducing visible interface behavior without implementing the corresponding service or persistence logic.

Automated Test-Suite Construction. To construct ordered cross-layer test suites at scale without expert-authored tests, WebProgramBench uses a fully automated four-phase pipeline. Each phase transforms the output of the preceding phase into the input required by the next: *1) Static analysis.*

For each reference repository, a coding agent analyzes the extracted endpoint metadata, database schema, and reference implementation to identify endpoint behaviors and the persistent resources they provide and require. Each endpoint behavior forms one node in a behavior-dependency graph, while each directed edge encodes a prerequisite relation between two behaviors. A topological sort of this graph produces the initial behavior order used in the next phase. *2) Stateful execution planning.* Following the initial behavior order, a coding agent traverses the behavior-dependency graph and instantiates each endpoint behavior as one or more concrete test executions. If the preceding executions do not establish the application state required by a target behavior, the coding agent inserts the necessary provider executions before that target. The resulting prerequisite relations determine the final execution order of the complete test suite. *3) Cross-layer test synthesis.* Following the final execution order, a coding agent constructs one executable cross-layer test for each concrete execution. The resulting tests form an initial executable cross-layer test suite that has not yet been validated against the reference application. *4) Reference-grounded iterative refinement.* The evaluation harness executes the initial suite against the reference application and records assertion-level evidence for every test. A coding agent uses this evidence to diagnose and repair failing tests, after which the reference state is reset and the refined suite is executed again. After each round, validated execution coverage measures the fraction of planned executions that pass reference validation. Refinement continues until the acceptance criterion is met, and only validated executions enter the final benchmark. Appendix A provides formal definitions and implementation details, while Algorithm 1 formalizes the complete end-to-end construction procedure.

3.3 Source Leakage Prevention with Internet Access

WebProgramBench preserves Internet access to reflect realistic software development and allow agents to obtain dependencies, documentation, and external assets. However, this access creates a direct shortcut because agents may retrieve the reference source code instead of reconstructing the application. To address this risk, WebProgramBench introduces an anti-cheating system whose Deep Research Agent discovers public repositories and source distributions associated with each task. During model inference, the resulting task-specific blocklist denies and logs requests to these sources while preserving access to unrelated Internet resources. The system therefore performs large-scale deep research to identify and block a broad range of source-retrieval paths, making source-code retrieval an impractical form of benchmark hacking compared with implementing the task directly. The Deep Research Agent configuration is provided in Appendix E.

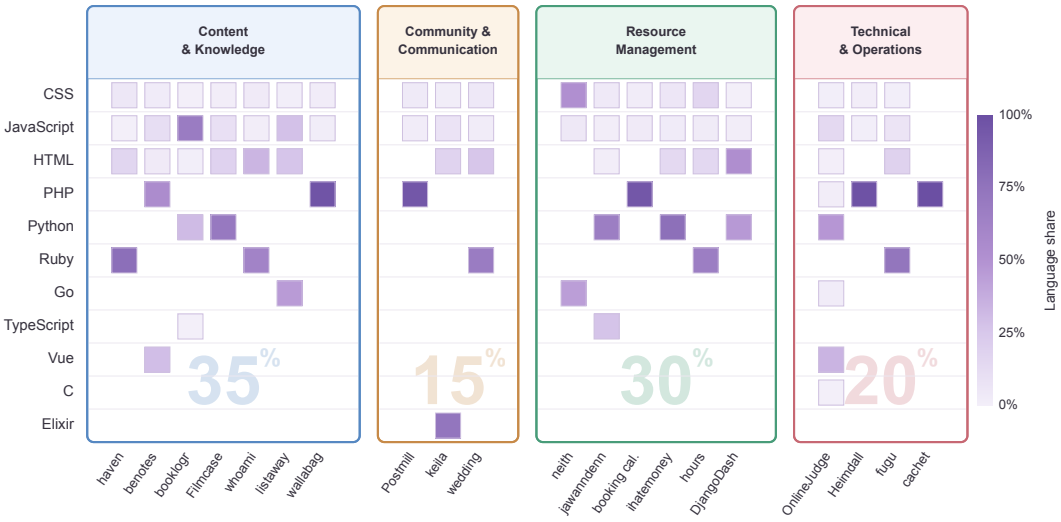

Figure 2: Distribution of 11 language categories and four application domains.

3.4 Benchmark Composition

WebProgramBench comprises 20 reconstruction tasks drawn from real-world open-source websites. We select stable releases that support reliable deployment, reset, and contract extrac-

tion. Each website provides observable browser workflows backed by server logic and persistent state. As shown in Figure 2, the selected websites span four application domains and 11 language categories. Overall, WebProgramBench covers 876,842 lines of code across its 20 applications. The accompanying task-specific blocklists contain 8,867 source-related URL prefixes across these applications. This breadth and scale require agents to reconstruct substantial and heterogeneous software systems rather than isolated interfaces or small application fragments. The final benchmark contains 1,254 executable cross-layer tests. Reference-grounded refinement achieves a macro-average validated execution coverage of 89.3% across websites. The benchmark averages 62.7 tests per website, with counts ranging from 11 to 154, as summarized in Table 1. At a finer granularity, WebProgramBench enforces 20,851 assertions, including 6,881 interface, 8,746 service, and 5,224 persistence assertions. Together, the resulting test suites provide a broad and strictly validated executable evaluation.

Table 1: Cross-layer test-suite statistics.

Statistic	Value
Cross-layer tests	1,254
Tests / website	62.7
Validated coverage	89.3%
Required assertions	20,851
Assertions / website	1,042.6

4 Experiments

4.1 Experimental Setup

Our evaluation includes five leading models, including GPT-5.6 Sol, Claude Opus 5, Gemini 3.1 Pro, GLM-5.2, and DeepSeek-V4.1 Flash, each using its officially recommended inference settings. To isolate underlying model capabilities, we use mini-swe-agent, a deliberately minimal harness without model-specific prompting. The harness is equipped with Chrome DevTools MCP[1] for interacting with and debugging both the live reference application and its reconstructed candidate. Under this common setup, we use four primary metrics to measure complete and near-complete reconstruction and to quantify fine-grained behavioral correctness. **Solved Rate** reports the fraction of fully reconstructed websites, while **Almost-Solved Rate** reports the fraction reaching at least 95% Test Pass Rate, including fully solved websites. **Cross-Layer Test Pass Rate** measures the fraction of executable cross-layer tests passed under strict joint verification, while **Cross-Layer Reward** captures partial assertion-level correctness within each test. **Interface**, **Service**, and **Persistence Assertion Rewards** further attribute partial correctness to the corresponding verification layers. Appendix E provides complete inference settings, Appendix F describes the isolated Task and evaluation environments, and Appendix K formalizes the metric definitions and aggregation rules.

4.2 Main Results

The main results in Table 2 reveal two central findings. **1) Frontier LLMs remain far from complete real-world application reconstruction.** Claude Opus 5 achieves the strongest overall performance, yet reaches the almost-solved threshold on only three applications, only one of which is fully reconstructed. Its 56.35% Cross-Layer Test Pass Rate further shows that even the best model fails nearly half of the executable cross-layer tests, while three of the five models fail to reconstruct even a single application completely. These results demonstrate that current LLMs remain far from achieving reliable end-to-end reconstruction of real-world applications. **2) Current LLMs reconstruct behavioral fragments, not coherent systems.** Cross-Layer Reward exceeds Cross-Layer Test Pass Rate for every model by 15.28 to 32.35 percentage points. This gap shows that models frequently satisfy individual assertions but fail to preserve correctness jointly across the interface, service, and persistence layers within a complete test, revealing a severe limitation in their ability to integrate behavior vertically across system layers. This partial correctness deteriorates further at the application level. Although models pass between 9.01% and 56.35% of the cross-layer tests, each model fully reconstructs at most one live application across the entire benchmark. This disparity shows that models can reproduce individual behaviors but remain severely limited in their ability to coordinate those behaviors horizontally across state-dependent workflows. Current LLMs therefore lack the compositional capability to integrate application behavior both vertically across system layers and horizontally across state-dependent workflows.

[1] https://github.com/ChromeDevTools/chrome-devtools-mcp/

Table 2: Main results, with best and second-best values bolded and underlined.

Model	Application Level		Test Level	Assertion-Level Rewards			
	Solved	Almost	Pass Rate	Cross-Layer	Interface	Service	Persistence
GPT-5.6 Sol	0.00	0.00	23.90	56.25	65.51	47.12	50.85
Claude Opus 5	**5.00**	**15.00**	**56.35**	<u>71.63</u>	<u>74.41</u>	**69.64**	**67.88**
Gemini 3.1 Pro	**5.00**	<u>5.00</u>	9.01	28.17	32.29	23.84	23.98
GLM-5.2	0.00	0.00	37.88	65.03	70.21	61.82	58.78
DeepSeek-V4.1 Flash	0.00	0.00	<u>48.46</u>	**71.70**	**74.98**	<u>69.46</u>	<u>64.81</u>

4.3 Ablation Studies

Partial-Layer Evaluation Overestimates Reconstruction Success. To quantify false acceptance under incomplete evaluation, we recompute the Test Pass Rate using all seven nonempty combinations of the interface, service, and persistence layers. For each combination, a test passes only if all assertions in the selected layers pass, while full three-layer verification serves as the reference for measuring false acceptance. As shown in Figure 3, the mean Test Pass Rate is 35.12% under full three-layer verification. Removing interface, persistence, or service verification increases it to 37.93%, 40.33%, and 40.28%, respectively, while interface-only and persistence-only evaluation further increase it to 50.01% and 50.40%. Omitting any layer therefore allows behaviors that fail full verification to pass, showing that partial-layer evaluation overestimates reconstruction performance and that all three layers are necessary for strict evaluation.

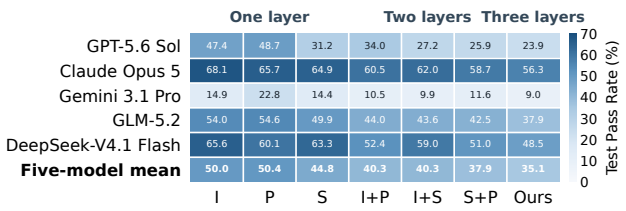

Figure 3: Ablation of cross-layer verification. Each cell reports the Test Pass Rate under the indicated combination of interface (I), service (S), and persistence (P) assertions. Ours jointly verifies all three layers.

Live Reference Access Is Essential for Behavioral Reconstruction. To isolate the contribution of interactive behavioral evidence, we evaluate three models across Postmill, Keila, Heimdall, Jawandenn, and Neith under otherwise identical configurations with and without access to the live reference. As shown in Table 3, removing live-reference access reduces the mean Test Pass Rate from 35.98% to 17.66%. This substantial degradation shows that static endpoint and database contracts enumerate available operations and persistent data structures but do not reveal how functions depend on one another through shared state. Sustained probing of the live reference exposes these dependencies. WebProgramBench requires agents to infer target application behavior through interaction rather than generate code from static specifications.

Agent Harness Ablation. We compare mini-swe-agent, PiAgent, and OpenCode using the same models and applications. As shown in Table 4, no harness consistently dominates across models. mini-swe-agent performs best with DeepSeek-V4.1 Flash, whereas PiAgent improves GLM-5.2 and OpenCode performs better with GPT-5.6 Sol. These reversals show that coding-agent capability emerges from the model–harness interaction. More fundamentally, the next frontier for LLMs is harness adaptability, the capacity to operate effectively across diverse orchestration frameworks rather than depend on a single favorable harness.

Table 3: Live-reference ablation. Each cell reports Test Pass Rate.

Model	Live	No Live
GPT-5.6 Sol	24.76	17.54
GLM-5.2	39.64	15.81
DeepSeek-V4.1 Flash	43.54	19.63

Table 4: Agent-harness ablation. Each cell reports Test Pass Rate.

Harness	GPT-5.6 Sol	GLM-5.2	DeepSeek-V4.1 Flash
mini-swe-agent	24.76	39.64	43.54
PiAgent	12.93	41.96	39.19
OpenCode	26.31	39.29	34.26

5 Analysis

5.1 Automated Test Construction and Evaluator Reliability

Four Refinement Rounds Suffice to Produce High-Coverage Validated Tests. The evaluation harness executes each initial suite against its reference application in the planned order. A coding agent then diagnoses and repairs failures using the resulting assertion-level execution evidence, after which the harness resets the reference application and evaluates the refined suite again. We measure progress using validated execution coverage, the fraction of planned executions that pass reference validation. Its formal definition and the complete construction settings are provided in Appendix A. As shown in Figure 4, refinement increases the mean validated execution coverage across websites from 28.39% for the initial suites to 89.34% for the final suites. All websites reach the 70% acceptance threshold within four refinement rounds, and Appendix Figure 8 reports the cumulative proportion reaching it after each round. Curve termination indicates that no further refinement was executed rather than that coverage had converged. Several websites continue improving after exceeding the acceptance threshold, with three ultimately reaching 100%, showing that additional refinement can produce further gains.

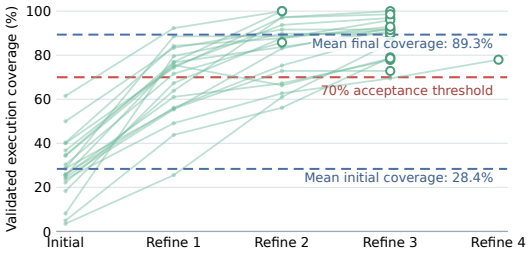

Figure 4: Validated execution coverage over refinement rounds. Each curve represents one website and ends at its last executed round.

Fixed-Candidate Evaluation Is Highly Reproducible. We evaluate the same fixed candidate for each selected website under an unchanged Judge configuration. Table 5 reports aggregate score dispersion and mean pairwise agreement of execution-level pass/fail outcomes, while Appendix Table 10 provides the complete run-level results. Across the five websites, Pass Rate is identical across all five runs for four websites and has a maximum standard deviation of only 2.28 percentage points, while the maximum Reward standard deviation is 1.12 percentage points. Mean pairwise outcome agreement is 100% for four websites and 96.13% for Neith. This stability extends from the low-performing Postmill reconstruction to near-complete candidates. The consistently low variation across applications with widely varying scores demonstrates strong evaluator reproducibility.

Table 5: Fixed-candidate evaluation repeatability.

Website	Pass Rate (Mean ± SD)	Reward (Mean ± SD)	Agreement
Postmill	$1.32 \pm 0.00\%$	$30.35 \pm 0.37\%$	100.00%
Keila	$3.30 \pm 0.00\%$	$15.29 \pm 0.12\%$	100.00%
Heimdall	$7.69 \pm 0.00\%$	$29.56 \pm 0.62\%$	100.00%
Jawanndenn	$91.67 \pm 0.00\%$	$99.43 \pm 0.00\%$	100.00%
Neith	$93.55 \pm 2.28\%$	$97.99 \pm 1.12\%$	96.13%

A Compact Open-Weight Judge Provides Stable Evaluation. Evaluation remains stable from a large-scale Judge to a much smaller 27B Judge. Across websites, scores vary between the two Judges by at most 3.23 percentage points in Pass Rate and 1.91 percentage points in Cross-Layer Reward, as reported in Appendix Table 11. Unlike recent benchmarks that rely on proprietary model-based evaluators (He et al., 2026; Desai et al., 2026), WebProgramBench delivers reliable evaluation with a compact open-weight Judge.

5.2 Agent Trajectory Analysis

We analyze agent trajectories to characterize model-specific development strategies and closed-loop engineering capabilities. Average trajectory length varies by more than an order of magnitude, from 26.0 turns for GPT-5.6 Sol to 268.8 turns for DeepSeek-V4.1 Flash. In the following analysis, we normalize trajectory progress to compare strategy evolution independently of trajectory length. Complete trajectory statistics are provided in Appendix H.

Models Adopt Distinct Development Strategies. As shown in the top row of Figure 5, GPT-5.6 Sol and GLM-5.2 exhibit the clearest staged strategies, with Probe dominating early, Write peaking near the trajectory midpoint, and Verify dominating late. Once they transition to verification, both models rarely return to probing the reference application. Claude Opus 5 and DeepSeek-V4.1 Flash follow the same broad progression but continue probing after verification becomes promi-

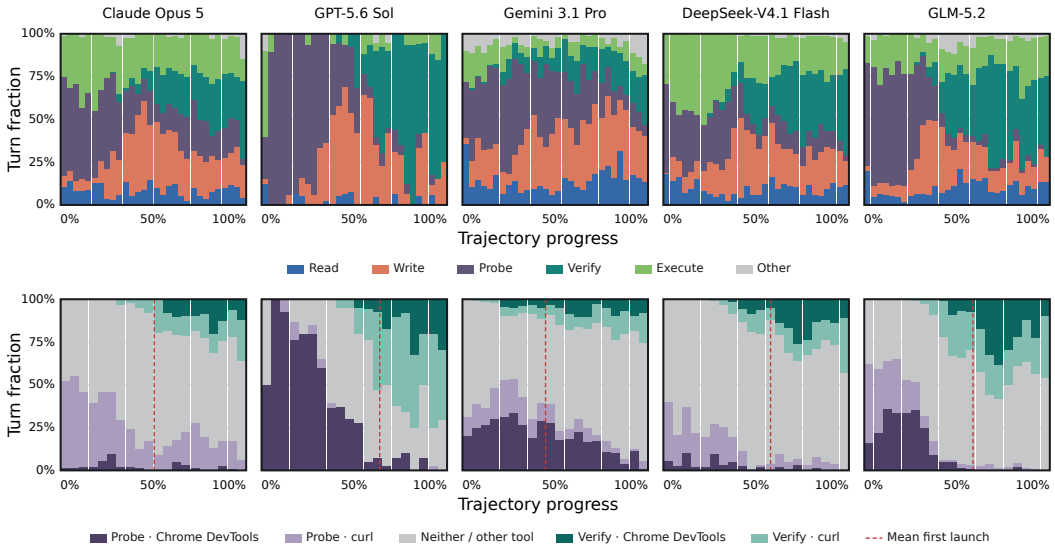

Figure 5: Model-specific development strategies over normalized trajectory progress, showing action composition above and Probe and Verify channels below. Dashed lines mark mean first launch.

nent, producing a more interleaved development process. In contrast, Gemini 3.1 Pro exhibits no clear phase boundary and maintains a mixture of Read, Write, Probe, and Verify throughout its trajectory. Despite these temporal differences, Probe and Verify together account for 48.1% of turns on average across models, as reported in Appendix Table 14. This substantial share demonstrates that WebProgramBench is fundamentally more than a code-generation task, as nearly half of agent turns are devoted to behavioral exploration and verification. The bottom row shows that the first successful candidate launch generally marks the shift from reference probing to candidate verification. Appendix Table 15 reveals opposing channel shifts across models. GPT-5.6 Sol and Gemini 3.1 Pro favor Chrome DevTools during Probe but curl during Verify, whereas Claude Opus 5 and DeepSeek-V4.1 Flash instead increase their Chrome DevTools use during Verify.

Atomic Commands Reveal What Turn Counts Hide. Conventional trajectory analyses count turns, implicitly treating each model response as a comparable unit of work. As shown in Figure 6, our turn-level parsing reveals that GPT-5.6 Sol packs an average of 9.21 atomic shell commands into only 26.0 turns per task, whereas Gemini 3.1 Pro averages 1.71 commands across 127.6 turns. Despite this nearly five-fold difference in trajectory length, their total command volumes per task remain comparable, as reported in Appendix Table 12. Atomic commands per task correlate substantially more strongly with Cross-Layer Test Pass Rate than turn counts do, with Pearson correlations of 0.95 and 0.74, respectively.

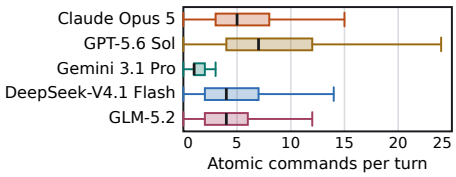

Figure 6: Atomic commands per turn.

6 Conclusion

In this paper, we introduce WebProgramBench, the first benchmark to rigorously evaluate whether coding agents can reconstruct production-scale stateful web applications from scratch through black-box interaction with live reference applications. Our novel executable cross-layer evaluation protocol jointly verifies behavior across the interface, service, and persistence layers, preventing shortcut solutions that merely mock the frontend from passing. Moreover, our fully automated test construction pipeline generates reference-validated, high-coverage executable tests without expert annotation and makes strict evaluation readily scalable to new websites. Our experiments show that frontier LLMs remain far from complete behavioral reconstruction, with even the best-performing model fully reconstructing only one website.

AI USE STATEMENT

In this work, we used generative AI tools for the following tasks with required disclosure: Generate synthetic data sets, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, support qualitative and thematic data analysis, and interpret results. We did not use generative AI tools to help develop theoretical models or conceptual frameworks or to clean and reformat datasets. The tasks "formulate mathematical claims," "provide critical ingredients for proving mathematical claims," and "assist in the writing of proofs" were not applicable to this work.

The benchmark tasks are derived from real-world open-source web applications, while generative AI was used to synthesize the executable tests and related test artifacts. Additionally, we used generative AI tools for software development, scientific figure preparation, literature review, brainstorming, and paper structuring, drafting, and editing.

We reviewed all AI-assisted work. Generated tests included in the benchmark were executed and validated against the reference applications. AI-assisted code used in the reported experiments was reviewed and tested. We checked all reported analyses and claims against the original experimental results and verified all citations using the original papers. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

WebProgramBench is constructed from open-source applications and evaluates reference and candidate systems in isolated Docker environments using synthetic accounts and data. It does not involve human subjects or collect personal data. Behavioral reconstruction may be misused for unauthorized website cloning, but the benchmark is limited to open-source applications and intended for research. Behavioral equivalence also does not establish the absence of security vulnerabilities, which require separate evaluation before deployment.

REPRODUCIBILITY STATEMENT

We will release WebProgramBench to the research community, including task definitions, pinned Docker environments, generated cross-layer test suites, the evaluation toolkit, raw evaluation results, complete inference logs, and all model-generated artifacts. We provide full details of the task formulation, experimental setup, metrics, and aggregation procedures. The released scripts reproduce all reported tables and figures.

REFERENCES

Yutong Bian, Xianhao Lin, Yupeng Xie, Tianyang Liu, Mingchen Zhuge, Siyuan Lu, Haoming Tang, Jinlin Wang, Jiayi Zhang, Jiaqi Chen, Xiangru Tang, Yongxin Ni, Sirui Hong, and Chenglin Wu. You don't know until you click:automated GUI testing for production-ready software evaluation. *CoRR*, abs/2508.14104, 2025. doi: 10.48550/ARXIV.2508.14104. URL https://doi.org/10.48550/arXiv.2508.14104.

Grace Man Chen, Litao Guo, Yifan Wu, Yiyu Chen, Yenchi Tseng, Sicheng Liu, Yuyu Luo, and Ying-Cong Chen. Ui2app: Benchmarking visual interaction inference in executable web application generation. *CoRR*, abs/2607.06306, 2026. doi: 10.48550/ARXIV.2607.06306. URL https://doi.org/10.48550/arXiv.2607.06306.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. SWE-milestone: Evaluating AI agents on continuous software evolution. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=HeRPNfJaod.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Daniel Wang, Derek Chen, Pranav Raja, Albert Liu,

Marek Suppa, Nevasini Sasikumar, Luyang Kong, Erik Quintanilla, Xiangyi Li, Ivan Bercovich, and Steven Dillmann. Swe-marathon: Can agents autonomously complete ultra-long-horizon software work? *CoRR*, abs/2606.07682, 2026. doi: 10.48550/ARXIV.2606.07682. URL https://doi.org/10.48550/arXiv.2606.07682.

Jingzhe Ding, Shengda Long, Changxin PU, Ge Zhang, zhou huan, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, FEI HU, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Chen Qizhi, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Minghao Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Haoran Wang, Zihan Wang, Weihao Xie, Yishuo Yuan, Jiayu Zhang, Yunfei Zhao, He Zhu, LIYA ZHU, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, and Wenhao Huang. NL2repo-bench: Towards long-horizon repository generation evaluation of coding agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=wqQam1muOQ.

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. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=lJpXXwhRRF.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=7o6SG5gVev.

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.

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 Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/6841eed8bb6a2ec49e49235c8115efee-Abstract-Datasets_and_Benchmarks_Track.html.

Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Mingjie Zhan, and Hongsheng Li. Fullstack-agent: Enhancing agentic full-stack web coding via development-oriented testing and repository back-translation. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=xblX2nQrj7.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/miserendino25a.html.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA,*

April 29 - May 4, 2025, pp. 3956–3974. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.NAACL-LONG.199. URL https://doi.org/10.18653/v1/2025.naacl-long.199.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, Kepeng Lei, Yifan Yao, Xinping Lei, Wenqiang Zhu, Zongxian Feng, Han Li, Junqi Xiong, Dailin Li, Zuchen Gao, Kun Wu, Wen Xiang, Ziqi Zhan, Yuanxing Zhang, Wuxuan Gong, Ziyuan Gao, Guanxiang Wang, Yirong Xue, Mengtong Li, Mengfei Xie, Xiaojiang Zhang, Jinghui Wang, Wenhao Zhuang, Zheng Lin, Huiming Wang, Zhaoxiang Zhang, Yuqun Zhang, Haotian Zhang, Bin Chen, and Jiaheng Liu. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models. volume abs/2511.05459, 2025. doi: 10.48550/ARXIV.2511.05459. URL https://doi.org/10.48550/arXiv.2511.05459.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. Swe-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-bench: Can language models rebuild programs from scratch? *CoRR*, abs/2605.03546, 2026. doi: 10.48550/ARXIV.2605.03546. URL https://doi.org/10.48550/arXiv.2605.03546.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/5afa9cb1e917b898ad418216dc726fbd-Abstract-Datasets_and_Benchmarks_Track.html.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In Maria Christakis and Michael Pradel (eds.), *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pp. 1592–1604. ACM, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. Repozero: Can llms generate a code repository from scratch? *CoRR*, abs/2605.07122, 2026. doi: 10.48550/ARXIV.2605.07122. URL https://doi.org/10.48550/arXiv.2605.07122.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

A AUTOMATED TEST-SUITE CONSTRUCTION

As shown in Figure 7, our automated test-suite construction pipeline comprises four phases.

Phase 1: Static analysis. The goal of static analysis is to construct a validated behavior-dependency DAG that provides the prerequisite structure for subsequent dynamic grounding and test ordering. In this graph, each node represents an endpoint behavior, and each directed edge points from a behavior that provides a persistent resource to one that requires it. The test-suite construction pipeline reuses the full endpoint metadata and database schema extracted during automated task construction. Guided by these structural inputs, a coding agent with specialized skills analyzes the reference implementation of each endpoint and produces a structured specification for each behavior. Each behavior specification records its triggering conditions, request semantics, expected response, and required authentication context. The coding agent also identifies the database tables and columns that the behavior reads or writes. Each behavior specification also records the persistent resources it provides for subsequent behaviors and requires from prior behaviors.

We represent dependencies among endpoint behaviors through persistent resources, which capture the application state established by one behavior and required by another. Let $\mathcal{N}$ denote the set of identified endpoint-behavior nodes. For each node $n \in \mathcal{N}$, $\mathrm{Prov}(n)$ and $\mathrm{Req}(n)$ denote the persistent resources it provides and requires. We construct an initial candidate behavior-dependency graph $G_B^{(0)} = (\mathcal{N}, \mathcal{E}_B^{(0)})$, where

$$n_i \to n_j \quad \text{if} \quad \mathrm{Prov}(n_i) \cap \mathrm{Req}(n_j) \neq \varnothing. \tag{1}$$

Dedicated authentication endpoint behaviors are excluded during graph construction because the evaluation harness provisions identities and sessions separately. Endpoint behaviors that require authentication remain in the graph.

The initial graph may contain missing providers or cycles because resource annotations can be incomplete or inconsistent. We flag a missing provider whenever no behavior is annotated as providing a required resource, and detect cycles by identifying nontrivial strongly connected components and self-loops. A coding agent then selectively revises the specifications of the affected endpoint behaviors. For each missing provider, the agent either identifies an omitted provider behavior, removes an incorrect dependency, or confirms that the required resource is created during application initialization. Cycles are resolved by revisiting whether their edges represent genuine prerequisites. Together, these revisions produce the validated behavior-dependency DAG, denoted by $G_B = (\mathcal{N}, \mathcal{E}_B)$.

Phase 2: Stateful execution planning. The static behavior-dependency DAG provides an initial traversal order for dynamic planning, allowing the coding agent to dynamically refine the initial order into a concrete execution plan without searching from scratch. We denote the evolving Execution Dependency Graph by $G_E = (\mathcal{T}, \mathcal{D})$. Each node in $\mathcal{T}$ represents a concrete execution instantiated from an endpoint behavior. Each edge in $\mathcal{D}$ represents a direct prerequisite between two executions. The same endpoint behavior may produce multiple execution nodes as it is instantiated under different application states. A topological ordering of G_E therefore defines a feasible test execution order. Constructing each execution requires determining whether the application state established by preceding executions satisfies the behavior's preconditions. We represent the latent application state relevant to test execution as a Symbolic Resource State $S = \{s_1, \ldots, s_M\}$. Each s_j represents the symbolic state of an application resource that affects whether or how an endpoint behavior can be executed.

Following a topological ordering of the static DAG, the coding agent attempts to instantiate each endpoint behavior n_i given the current Symbolic Resource State S and Execution Dependency Graph G_E. We represent the result as an ordered sequence of concrete executions:

$$\mathbf{t}_i = \mathrm{Instantiate}(n_i, S, G_E). \tag{2}$$

If the current state already satisfies the behavior's preconditions, $\mathbf{t}_i$ consists only of the target execution, $\mathbf{t}_i = \langle t_i^* \rangle$. If additional executions can establish the required preconditions, $\mathbf{t}_i$ consists of the necessary provider executions followed by the target execution, $\mathbf{t}_i = \langle t_i^{(1)}, \ldots, t_i^{(\ell)}, t_i^* \rangle$. If the required preconditions cannot be established, $\mathbf{t}_i$ is empty, $\mathbf{t}_i = \langle\rangle$, and the behavior remains pending. The execution sequence is constructed incrementally. The effects of each newly instantiated execution $t \in \mathbf{t}_i$ are immediately applied to the Symbolic Resource State.

$$S \leftarrow \mathrm{Apply}\left(S, \mathrm{Eff}(t)\right). \tag{3}$$

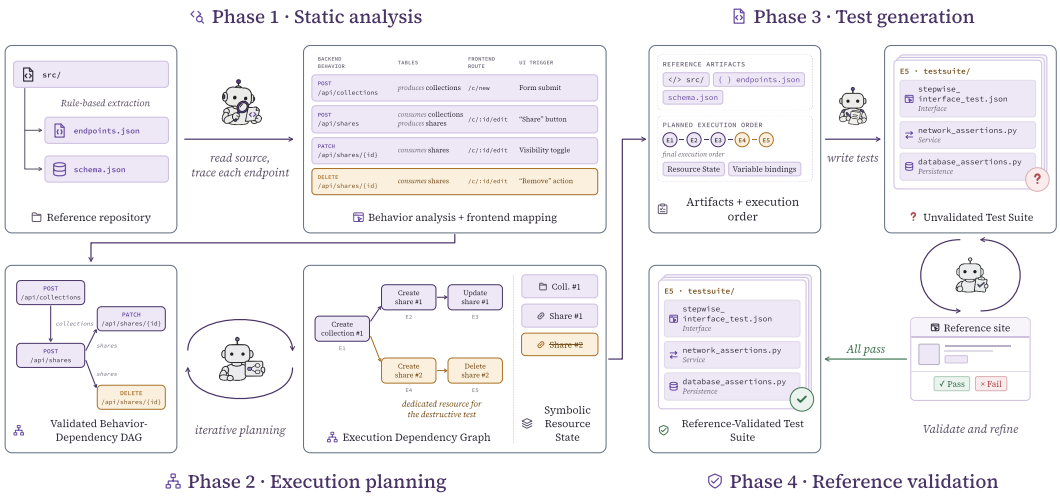

Figure 7: Overview of the automated test-suite construction pipeline. The pipeline comprises four phases, including static analysis, stateful execution planning, cross-layer test synthesis, and reference-grounded iterative refinement. Together, these phases produce an executable cross-layer test suite backed by strict reference validation.

Here, $\mathrm{Eff}(t)$ denotes the resources created, modified, or deleted by t. Creation adds a new resource to S, modification updates the state of an existing resource, and deletion marks a resource as deleted without removing it from S. The execution and its direct prerequisites are then added to the Execution Dependency Graph:

$$G_E \leftarrow \mathrm{Extend}\,(G_E, t)\,. \tag{4}$$

As planning proceeds, the Symbolic Resource State and Execution Dependency Graph evolve jointly. Their updates are interdependent because each new execution changes the current state, while the updated state determines which executions can be added next. Behaviors that remain pending after a complete traversal are retried in the next planning round under the same static order. Finally, topologically sorting the completed Execution Dependency Graph produces the final execution order $\Pi = \mathrm{TopologicalSort}(G_E)$ for the entire test suite.

Phase 3: Cross-layer test synthesis. Given the planned execution order, the coding agent generates an executable cross-layer test for each concrete execution. Test construction begins by defining the required runtime variables from the execution's associated resources and the reference source code. To maintain value consistency, the coding agent then synthesizes the three layers around the same runtime variables: a step-wise interaction procedure with functional checks for the interface layer, request and response assertions for the service layer, and database-state assertions for the persistence layer. After generating the three-layer test logic, the coding agent instantiates the shared runtime variables with concrete test data. It produces string values that satisfy the inferred constraints and retrieves suitable public binary artifacts from the Internet, removing the need for manual data curation. To reduce generation complexity, the coding agent synthesizes only behavior-specific browser steps and assertions, while a shared evaluation harness provides reusable infrastructure for GUI-agent orchestration, network capture, runtime-variable propagation, and database access.

Appendix J illustrates this design using a generated cross-layer test for the authenticated profile-update behavior in Online Judge. Together, the generated tests and instantiated data form an initial executable cross-layer test suite $\widehat{\mathcal{T}}^{(0)}$, which is subsequently validated and repaired on the reference application.

Phase 4: Reference-grounded iterative refinement. The initial executable cross-layer test suite is generated by coding agents and may not faithfully capture the behavior of the reference application. Our pipeline uses an iterative loop to validate and refine it on the reference application. In each round, the evaluation harness executes every concrete test and records fine-grained evidence for every required assertion. For each test, the execution record contains the outcome of every interface action, the captured HTTP exchanges, the database checks, the authentication result, and the com-

plete execution log. Using the recorded evidence and relevant reference source code, a coding agent diagnoses each failing test and revises the corresponding behavior specification, runtime variables, test data, or cross-layer test logic. The refinement process preserves the planned execution order. An artifact allowlist confines every edit to the generated test suite. Integrity checks reject any out-of-scope change and restore the previous version. After each round, the evaluation harness resets the reference application and executes the refined tests again. Let $\mathcal{T}_{\text{valid}}^{(k)} \subseteq \mathcal{T}$ denote the concrete executions that pass reference validation after round k. We define the validated execution coverage as

$$C^{(k)} = \frac{\left|\mathcal{T}_{\text{valid}}^{(k)}\right|}{|\mathcal{T}|}. \tag{5}$$

The refinement loop continues until the coverage reaches a predefined acceptance criterion. Only reference-validated executions are retained in their planned order for the final test suite and benchmark score. The resulting suite provides an executable cross-layer evaluation backed by strict reference validation.

B Comparison with Related Benchmarks

Table 6: Comparison with existing coding benchmarks across multiple dimensions. The Languages column counts the language categories represented in the reference implementations. ✓ under Unconstrained Implementation indicates that agents may freely choose the programming languages, frameworks, and system architecture. Backend Verification and DB Verification indicate whether a benchmark verifies service behavior and persistent database state. Joint Verification indicates whether the same execution jointly verifies interface outcomes, HTTP exchanges, and database effects. Fully Automated Test Construction indicates that test suites are generated, validated, and refined without human annotation or repair. △ denotes partial support or separate rather than joint verification. Anti-shortcut denotes an explicit evaluation mechanism that prevents solutions implementing only superficial interface behavior from passing. * denotes the Product Clones subset of SWE-Marathon.

(a) Repository and Program Generation from Scratch

Benchmark	Task Definition	Unconstrained Implementation	Languages	Evaluation	Test Construction
Commit0	Complete a Python library from an API specification and a partial repository	✗	1	Executable unit tests	Repository-derived
NL2Repo-Bench	Generate a complete Python library from a requirements document	✗	1	Executable upstream tests	Repository-derived
ProgramBench	Reconstruct a program from documentation and black-box interaction with a reference executable	✓	5	Executable behavioral I/O tests	Reference-validated synthesis
WebProgramBench	Reconstruct a full stack app from contracts and black-box interaction with a live website	✓	11	Executable cross-layer tests	Reference-validated synthesis

(b) Application Generation and Reconstruction from Scratch

Benchmark	Task Definition	Avg. Tests	Unconstrained Implementation	Verification			Fully Automated Test Construction	Anti-shortcut
				Backend	DB	Joint		
Vision2Web L3	Build a full stack website from screenshots, a PRD, and assets	28.2	✓	✗	✗	✗	✗	✗
FullStack-Agent	Build a full stack website from natural-language requirements	16.2	✗	✓	✓	△	✗	✓
UI2App	Build an interactive frontend from screenshots	–	✗	–	–	–	✗	✗
SWE-Marathon*	Build a product clone from a product specification	16.6†	✗	✓	△	✗	✗	✓
WebProgramBench	Reconstruct a full stack app from contracts and black-box interaction with a live website	62.7	✓	✓	✓	✓	✓	✓

† The SWE-Marathon average follows the five Product Clone tasks reported in its paper, counting each separate UI/UX verifier once. For Slack Clone, we follow the five correctness gates and separate UI/UX verifier in the released code.

C End-to-End Benchmark Construction

Algorithm 1 formalizes the construction of a WebProgramBench instance. For a role ρ, $\mathrm{Agent}_{\rho}(X)$ denotes a coding-agent session initialized with role-specific instructions ρ and context X. Operations nested within the session are executed by that agent using its available tools.

Algorithm 1 End-to-End Construction of a WebProgramBench Instance

Require: R denotes a pinned, deployable reference implementation; τ denotes a validated-execution acceptance threshold
Ensure: Q denotes an agent-facing reconstruction task; L denotes a source blocklist; $\mathcal{T}^*$ denotes an ordered reference-validated executable test suite

— Part I: Automated Task Construction —
with $\mathrm{Agent}_{\mathrm{task}}(R)$ **do**
  $W \leftarrow \texttt{DeployReference}(R)$ // W denotes the live reference website
  $M \leftarrow \texttt{ExtractMetadata}(R, W)$ // M denotes endpoint and database metadata
  $C \leftarrow \texttt{Sanitize}(M)$ // C denotes sanitized contracts
  $Q \leftarrow \texttt{PackageTask}(W, C)$ // Q denotes the reconstruction task
end with
$L \leftarrow \mathrm{Agent}_{\mathrm{blocklist}}(R)$ // L denotes the verified source blocklist
— Part II: Automated Test-Suite Construction —
— Phase 1: Static Analysis —
with $\mathrm{Agent}_{\mathrm{static}}(R, M)$ **do**
  $\mathcal{N} \leftarrow \texttt{AnalyzeBehaviors}(R, M)$ // $\mathcal{N}$ denotes annotated endpoint behaviors
  $G_B^{(0)} \leftarrow \texttt{BuildDependencyGraph}(\mathcal{N})$ // $G_B^{(0)}$ denotes the initial behavior-dependency graph
  $G_B \leftarrow \texttt{ResolveCycles}\big(\texttt{ResolveMissingProviders}(G_B^{(0)})\big)$ // G_B denotes the validated behavior-dependency DAG
end with
— Phase 2: Stateful Execution Planning —
with $\mathrm{Agent}_{\mathrm{planning}}(R, G_B)$ **do**
  $S \leftarrow \{s_1, \ldots, s_M\}$ // S denotes the symbolic resource state
  $G_E = (\mathcal{T}, \mathcal{D}) \leftarrow (\emptyset, \emptyset)$ // G_E denotes the execution-dependency graph
  $\mathcal{U} \leftarrow \mathrm{TopologicalSort}(G_B)$ // $\mathcal{U}$ denotes unresolved behaviors in static order
  repeat
    $\mathcal{U}_{\mathrm{prev}} \leftarrow \mathcal{U}; \mathcal{U} \leftarrow \langle\rangle$
    for each $n_i \in \mathcal{U}_{\mathrm{prev}}$ **do**
      $\mathbf{t}_i \leftarrow \mathrm{Instantiate}(n_i, S, G_E)$
      if $\mathbf{t}_i = \langle\rangle$ **then**
        $\mathcal{U} \leftarrow \mathcal{U} \parallel \langle n_i \rangle$
      else
        for each $t \in \mathbf{t}_i$ **do**
          $S \leftarrow \mathrm{Apply}\,(S, \mathrm{Eff}(t))$
          $G_E \leftarrow \mathrm{Extend}\,(G_E, t)$
        end for
      end if
    end for
  until $\mathcal{U} = \langle\rangle$ **or** $\mathcal{U} = \mathcal{U}_{\mathrm{prev}}$
  $\Pi \leftarrow \mathrm{TopologicalSort}(G_E)$ // Π denotes the final execution order
end with
— Phase 3: Cross-Layer Test Synthesis —
with $\mathrm{Agent}_{\mathrm{synthesis}}(R, \Pi)$ **do**
  $\widehat{\mathcal{T}}^{(0)} \leftarrow \langle\rangle$ // $\widehat{\mathcal{T}}^{(0)}$ denotes the initial executable test suite
  for each $t \in \Pi$ **do**
    $V_t \leftarrow \texttt{DefineRuntimeVariables}(t, R)$ // V_t denotes shared runtime variables
    $A_t \leftarrow \texttt{SynthesizeCrossLayerLogic}(t, V_t, R)$ // A_t denotes joint interface, service, and persistence logic
    $X_t \leftarrow \texttt{InstantiateTestData}(V_t, A_t)$ // X_t denotes concrete test data
    $\widehat{t} \leftarrow \texttt{AssembleExecutableTest}(t, V_t, A_t, X_t)$
    $\widehat{\mathcal{T}}^{(0)} \leftarrow \widehat{\mathcal{T}}^{(0)} \parallel \langle \widehat{t} \rangle$
  end for
end with

Algorithm 1 End-to-End Construction of a WebProgramBench Instance (continued)

— Phase 4: Reference-Grounded Iterative Refinement —
$k \leftarrow 0$
repeat
  $\texttt{ResetReference}(W)$
  $Y^{(k)} \leftarrow \texttt{ExecuteSuite}(W, \widehat{\mathcal{T}}^{(k)}, \Pi)$ *// $Y^{(k)}$ denotes fine-grained execution evidence*
  $\mathcal{T}^{(k)}_{\text{valid}} \leftarrow \{t \in \mathcal{T} : \texttt{Pass}(\widehat{t}, Y^{(k)})\}$
  $C^{(k)} \leftarrow \dfrac{|\mathcal{T}^{(k)}_{\text{valid}}|}{|\mathcal{T}|}$
  accepted $\leftarrow (C^{(k)} \geq \tau)$
  if ¬accepted **then**
    $\widetilde{\mathcal{T}}^{(k+1)}_{\text{invalid}} \leftarrow \text{Agent}_{\text{refinement}}(R, \widehat{\mathcal{T}}^{(k)}_{\text{invalid}}, Y^{(k)})$ *// refine invalid tests only*
    $\widehat{\mathcal{T}}^{(k+1)} \leftarrow \texttt{MergeAllowlistedRepairs}(\widehat{\mathcal{T}}^{(k)}, \widetilde{\mathcal{T}}^{(k+1)}_{\text{invalid}})$
    $k \leftarrow k + 1$
  end if
until accepted
$\mathcal{T}^* \leftarrow \texttt{RetainValidatedTests}(\widehat{\mathcal{T}}^{(k)}, \mathcal{T}^{(k)}_{\text{valid}}, \Pi)$

D LIMITATIONS

We outline potential limitations of WebProgramBench.

Visual fidelity. WebProgramBench evaluates functional interface outcomes but does not score visual fidelity. We deliberately separate behavioral and stylistic equivalence because functional reconstruction should be established before appearance is assessed. Combining them could allow visually plausible outputs to obscure behavioral failures and introduce bias from visual metrics or model judgments. We will extend the benchmark with an independent visual fidelity evaluation after behavioral verification.

Performance equivalence. WebProgramBench evaluates behavioral equivalence but does not require candidates to match the runtime performance of the reference application. A behaviorally correct reconstruction may therefore exhibit higher latency, lower throughput under concurrent workloads, or greater resource consumption. Beyond functional equivalence, stronger coding agents should also produce efficient, scalable, and resource-conscious implementations. Evaluating the performance of model-generated software therefore represents the next frontier in coding-agent evaluation.

Benchmark scale. WebProgramBench currently contains 20 websites. Adding more website categories would further broaden its empirical coverage. The current benchmark already spans four domains, 11 language categories, and approximately one million lines of code. Each task requires reconstructing a production-scale application through several hours of agent interaction. More importantly, our test construction pipeline is fully driven by coding agents and requires no expert-authored tests, providing a practical path to scaling the benchmark to new websites.

Validated execution coverage. Test construction currently uses 70% validated execution coverage as its minimum acceptance threshold. Some planned executions may therefore remain outside the final test suite. However, the planned executions target the core endpoint behaviors and state-dependent workflows of each website, so this threshold covers its primary functionality. Our refinement experiment in Section 5.1 demonstrates that coverage continues to increase across rounds and reaches 100% on several websites. The current threshold therefore reflects a practical construction budget rather than convergence of the refinement process. Future benchmark releases will allocate additional refinement resources and adopt a higher acceptance threshold.

E EXPERIMENTAL DETAILS

Main experiment settings. Table 7 reports the model-specific inference configurations, while Table 8 summarizes the shared execution settings. We evaluate every model using mini-swe-agent without model-specific prompting, planning modules, or orchestration. Each agent is equipped with

Chrome DevTools MCP[2] to interact with and debug both the live reference application and its reconstructed candidate through browser actions and inspection of page states, screenshots, console messages, and HTTP traffic. Each run receives a 12-hour wall-clock budget with no explicit total token limit, while individual model responses use the model-specific output-token limits reported in Table 7 and each shell command is limited to 600 seconds. Every run is conducted in an isolated Docker environment with independently initialized reference and candidate states. Agents may access the live reference application and general Internet resources but cannot access the reference source code or evaluation tests. Requests matching the task-specific source blocklist are denied and logged during inference. All models receive identical task instructions, tools, execution environments, and computational budgets, making the model backend the only experimental variable. For interface verification, we equip the OpenCode harness with Chrome DevTools MCP and use the open-weight DeepSeek-V4.1 Flash as the underlying model of the GUI-agent verifier. This open evaluation stack supports independent reproduction without relying on proprietary evaluator models.

Table 7: Model-specific inference settings for the main evaluation. A dash denotes an unsupported parameter.

Model	Temperature	Top-p	Top-k	Reasoning	Max Output Tokens
GPT-5.6 Sol	–	–	–	Default	128,000
Claude Opus 5	–	–	–	Default	128,000
Gemini 3.1 Pro	1.0	0.95	64	Default	65,536
DeepSeek-V4.1 Flash	1.0	0.95	Default	Default	131,072
GLM-5.2	1.0	Default	Default	Default	131,072

Table 8: Shared execution settings for the main evaluation.

Setting	Value
Agent harness	mini-swe-agent
Browser interaction tool	Chrome DevTools MCP
Execution environment	Isolated Docker environment
Wall-clock limit	12 hours per task
Shell command timeout	600 seconds
Total token budget	No explicit limit
Internet access	Enabled with anti-cheating controls
Interface-verification harness	OpenCode
Interface-verification model	DeepSeek-V4.1 Flash

Automated test-suite construction settings. Table 9 summarizes the configuration used to construct the executable cross-layer test suites. GLM-5.2 operates through the OpenCode harness to generate and refine the tests. A generated suite is accepted once its validated execution coverage reaches the 70% threshold. Appendix A provides the complete construction and refinement procedure.

Table 9: Settings for automated test-suite construction.

Setting	Value
Construction model	GLM-5.2
Agent harness	OpenCode
Acceptance threshold	70% validated execution coverage

[2] https://github.com/ChromeDevTools/chrome-devtools-mcp/

All Test Suites Reach the Threshold within Four Rounds. As shown in Figure 8, none of the initial test suites reaches 70% validated execution coverage, whereas all 20 suites reach this acceptance threshold within four refinement rounds. After one refinement round, 11 suites reach the threshold, increasing to 16 within two rounds and 19 within three rounds. The cumulative threshold-attainment curve demonstrates that reference-grounded refinement efficiently converts imperfect initial suites into accepted evaluators within only a few rounds.

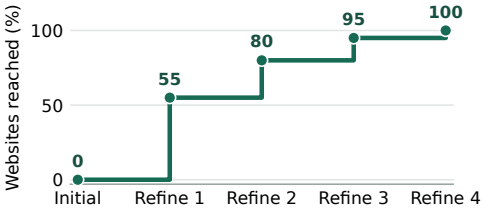

Figure 8: Cumulative percentage of websites reaching 70% validated execution coverage by each refinement round.

Source-leakage discovery settings. For each task, the Deep Research Agent uses DeepSeek-V4.1 Flash with the OpenCode harness to discover public repositories and source distributions associated with the reference application. The verified source locations are converted into a task-specific blocklist that is enforced during model inference.

F Task and Evaluation Environments

Each reconstruction run is conducted in an isolated Task Docker environment. Its internal network exposes the live reference application only through its deployed URL and prevents direct access to the reference container, database, source code, and hidden tests. Within this environment, the agent explores the live reference application and develops its reconstruction in the submission workspace. After the run terminates, only the submission workspace is archived as the candidate artifact. The artifact is then transferred to a separate, clean evaluation Docker environment, where the candidate is redeployed from scratch and the evaluation state is initialized independently of the Task run. Hidden tests are introduced only in the evaluation environment and executed after deployment, while the Task container and its residual runtime state do not participate in scoring. This separation ensures that evaluation measures an independently deployable reconstruction rather than artifacts or state retained from model inference.

G Evaluator Reliability Analyses

We evaluate reliability along two complementary dimensions. Run-to-run repeatability measures variation under a fixed Judge configuration, while Judge-model sensitivity measures variation across open-weight Judges of different scales.

Run-to-Run Repeatability. For all five websites, the fixed candidates were generated by GLM-5.2, and DeepSeek-V4.1 Flash served as the Judge model across all five evaluation runs. Table 10 reports the complete run-level results. Pass Rate and Cross-Layer Reward are reported as percentages.

Table 10: Run-level results for fixed-candidate evaluation repeatability.

	Postmill		Keila		Heimdall		Jawanndenn		Neith	
Run	Pass Rate	Reward	Pass Rate	Reward	Pass Rate	Reward	Pass Rate	Reward	Pass Rate	Reward
1	1.32	30.51	3.30	15.35	7.69	28.98	91.67	99.43	93.55	98.57
2	1.32	29.94	3.30	15.16	7.69	30.62	91.67	99.43	96.77	99.64
3	1.32	30.90	3.30	15.42	7.69	29.45	91.67	99.43	90.32	96.77
4	1.32	30.12	3.30	15.16	7.69	29.40	91.67	99.43	93.55	97.49
5	1.32	30.30	3.30	15.35	7.69	29.34	91.67	99.43	93.55	97.49

Judge-Model Sensitivity. Using the same fixed candidates, we compare DeepSeek-V4.1 Flash with the much smaller Qwen3.8-27B Judge over three independent evaluation runs.

Table 11: Judge-model sensitivity across open-weight evaluators of different scales. Values are mean ± sample SD over three runs, in percent.

(a) Test Pass Rate

Judge Model	Params.	Postmill	Keila	Heimdall	Jawanndenn	Neith
DeepSeek-V4.1 Flash	552B	1.32 ± 0.00	3.30 ± 0.00	7.69 ± 0.00	91.67 ± 0.00	93.55 ± 3.23
Qwen3.8-27B	27B	1.32 ± 0.00	3.30 ± 0.00	6.67 ± 0.89	88.89 ± 4.81	90.32 ± 0.00

(b) Cross-Layer Reward

Judge Model	Params.	Postmill	Keila	Heimdall	Jawanndenn	Neith
DeepSeek-V4.1 Flash	552B	30.45 ± 0.48	15.31 ± 0.13	29.68 ± 0.84	99.43 ± 0.00	98.33 ± 1.45
Qwen3.8-27B	27B	30.09 ± 0.20	15.25 ± 0.13	30.49 ± 0.26	98.87 ± 0.98	96.42 ± 0.00

H Additional Agent Trajectory Analyses

This appendix supplements the trajectory analysis in Section 5.2. All comparisons use the same set of tasks across the five evaluated models. For analyses over development progress, each trajectory is normalized to the interval $[0, 1]$, and statistics are first computed per task and then macro-averaged across tasks.

Aggregate Trajectory Statistics. Table 12 reports trajectory length, within-turn command density, and task-level command volume across models. These statistics expose differences in execution granularity and total interaction volume that turn counts alone obscure.

Table 12: Trajectory length and atomic-command statistics. Atomic Commands/Turn summarizes turn-level counts, while Atomic Commands/Task summarizes task-level totals. Except for Max. Turns, values report mean ± population SD.

Model	Turns / Task	Max. Turns	Atomic Commands / Turn	Atomic Commands / Task
GPT-5.6 Sol	26.0 ± 5.2	36	9.21 ± 7.29	239.3 ± 73.6
Claude Opus 5	246.6 ± 89.0	387	6.60 ± 5.14	$1{,}628.7 \pm 596.5$
Gemini 3.1 Pro	127.6 ± 61.1	263	1.71 ± 1.23	217.9 ± 95.8
GLM-5.2	252.7 ± 67.1	382	5.02 ± 4.74	$1{,}268.1 \pm 282.2$
DeepSeek-V4.1 Flash	268.8 ± 54.1	383	5.73 ± 4.69	$1{,}540.4 \pm 550.0$

Table 13 reports additional efficiency, launch, and execution statistics. The models differ substantially in wall-clock duration, generation cost, candidate-launch timing, tool use, and execution failures. These statistics complement the normalized trajectory analysis by revealing variation hidden by relative progress.

Table 13: Additional trajectory statistics. First-Launch Progress excludes tasks without a successful launch.

Model	Duration / Task (h)	First-Launch Progress	No-Launch Rate	Chrome-Active Turn Share	Failed-Turn Rate	Tokens / Task	Cost / Task ($)
GPT-5.6 Sol	0.62	63.7%	0%	47.4%	1.3%	22,550	2.66
Claude Opus 5	2.34	50.5%	10%	7.6%	1.3%	229,055	262.86
Gemini 3.1 Pro	0.81	44.8%	15%	26.5%	3.1%	23,109	5.83
GLM-5.2	4.83	58.6%	0%	22.1%	5.1%	129,728	13.30
DeepSeek-V4.1 Flash	2.35	57.8%	0%	10.9%	0.9%	201,003	1.63

Pareto-Optimality Depends on How Agent Effort Is Measured. Figure 9 compares Pareto-optimal models when resource use is measured by mean cost, turns, or atomic commands per task.

Under cost, DeepSeek-V4.1 Flash and Claude Opus 5 are Pareto-optimal, whereas under turns, GPT-5.6 Sol and Claude Opus 5 form the Pareto frontier. When effort is measured by atomic commands per task, all five models are Pareto-optimal: no model achieves a higher Cross-Layer Test Pass Rate with fewer commands, and each occupies a distinct interaction-volume–reconstruction-performance trade-off. Because atomic commands reflect executed work rather than model-specific response packaging, this frontier provides a more faithful view of agent efficiency.

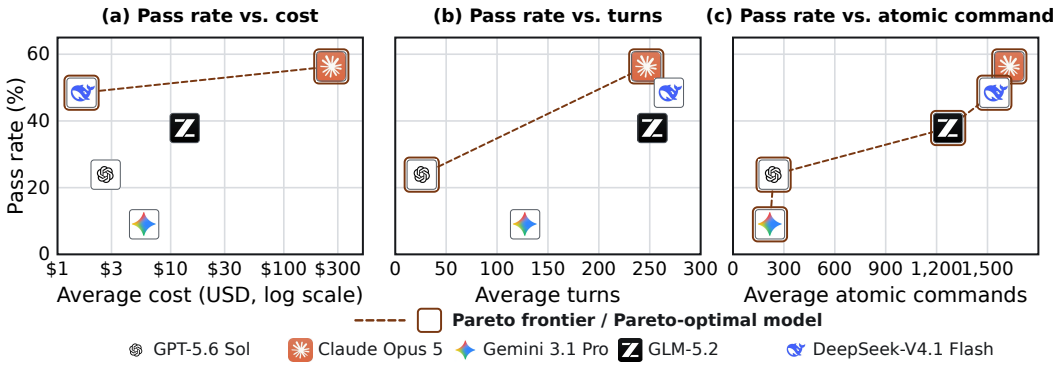

Figure 9: Pareto-optimal models under alternative measures of agent effort. Each panel compares Cross-Layer Test Pass Rate against mean cost, turns, or atomic commands per task, with higher Pass Rate and lower resource use preferred.

Aggregate Action Composition. Table 14 shows that Probe and Verify account for more than 40% of turns for four of the five models, demonstrating that WebProgramBench captures a genuinely compound engineering process in which reconstructing a complex system requires sustained investigation and debugging rather than one-pass code generation. GPT-5.6 Sol devotes 66.1% of its turns to Probe and Verify but only 5.9% to Execute, whereas DeepSeek-V4.1 Flash has the lowest combined share of Probe and Verify at 39.2% and the highest Execute share at 29.0%. Gemini 3.1 Pro allocates the largest shares to both Read (14.2%) and Write (28.0%).

Table 14: Overall action composition for the five evaluated models. The fixed action priority is Probe, Verify, Write, Execute, Read, and Other.

Model	Read	Write	Probe	Verify	Execute	Other	Probe+Verify
GPT-5.6 Sol	2.4%	24.8%	38.8%	27.4%	5.9%	0.8%	66.1%
Claude Opus 5	8.7%	22.7%	27.6%	14.3%	25.1%	1.6%	41.9%
Gemini 3.1 Pro	14.2%	28.0%	31.0%	12.1%	9.3%	5.5%	43.0%
GLM-5.2	9.9%	18.1%	24.2%	26.1%	18.6%	3.1%	50.4%
DeepSeek-V4.1 Flash	9.7%	21.1%	19.9%	19.4%	29.0%	0.9%	39.2%
Model mean	9.0%	22.9%	28.3%	19.9%	17.6%	2.4%	48.1%

Probe and Verify Interaction Channels. The bottom row of Figure 5 shows that Probe is concentrated before the first successful candidate launch, whereas Verify becomes prominent afterward. The launch therefore marks the transition from one-way reference exploration to an executable feedback loop. Table 15 further decomposes these activities by interaction channel. GPT-5.6 Sol and Gemini 3.1 Pro sharply reverse their channel preference, using Chrome DevTools for 92.2% and 62.5% of Probe turns but switching to curl for 70.0% and 83.1% of Verify turns, respectively. Claude Opus 5 and DeepSeek-V4.1 Flash exhibit the opposite shift, increasing their use of Chrome DevTools from 9.1% to 37.5% and from 16.3% to 38.8%, respectively. These opposing shifts show that interaction-channel choice depends jointly on the model and development activity rather than reflecting a fixed tool preference.

Table 15: Interaction channels used for Probe and Verify turns. Tasks without the corresponding action are excluded, and values may sum to less than 100% when other tools are used.

	Probe		Verify	
Model	curl	Chrome DevTools	curl	Chrome DevTools
GPT-5.6 Sol	6.3%	92.2%	70.0%	29.0%
Claude Opus 5	84.6%	9.1%	50.3%	37.5%
Gemini 3.1 Pro	32.7%	62.5%	83.1%	14.2%
GLM-5.2	50.8%	37.7%	54.5%	40.6%
DeepSeek-V4.1 Flash	50.8%	16.3%	43.0%	38.8%

Model Output Dynamics. Figure 10 shows that model output is often concentrated in pronounced bursts rather than distributed uniformly throughout development. GPT-5.6 Sol peaks near the trajectory midpoint, Claude Opus 5 peaks earlier, and GLM-5.2 and DeepSeek-V4.1 Flash exhibit similar mid-trajectory concentrations. These patterns suggest episodic implementation phases, although token counts are interpreted descriptively because tokenization differs across model providers.

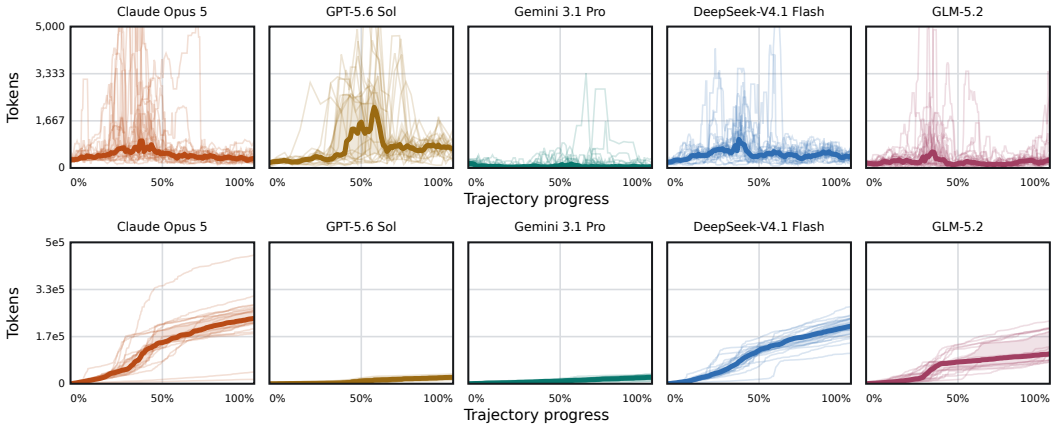

Figure 10: Model output dynamics over normalized trajectory progress. The upper row shows completion tokens per turn, while the lower row shows cumulative completion-token output. Thin curves represent individual tasks, bold curves show pointwise medians, and shaded regions denote interquartile ranges.

I Ordered Stateful Evaluation for Application Reconstruction

Unlike function-level benchmarks, whose test cases can be executed independently, interactive applications expose behavior through workflows coupled by persistent state, identities, and sessions. WebProgramBench resolves this ordering problem by compiling behavioral dependencies into an execution graph and carrying observed resources through a shared runtime variable pool. A failed execution does not prevent downstream tests from running when their required resources remain available; a test is skipped only when those resources cannot be resolved. As shown in Figure 11, the Postmill evaluation continues through failed registration, forum-creation, and posting tests, and begins skipping only when the comment workflow fails to establish the resource required for bookmarking.

J Generated Cross-Layer Test Example

We present a real cross-layer test generated for the authenticated profile-update behavior in Online Judge. The user edits seven profile fields and saves them through `PUT /api/profile/`. On success, the endpoint returns HTTP 200 with the updated profile, and the interface displays a success notification.

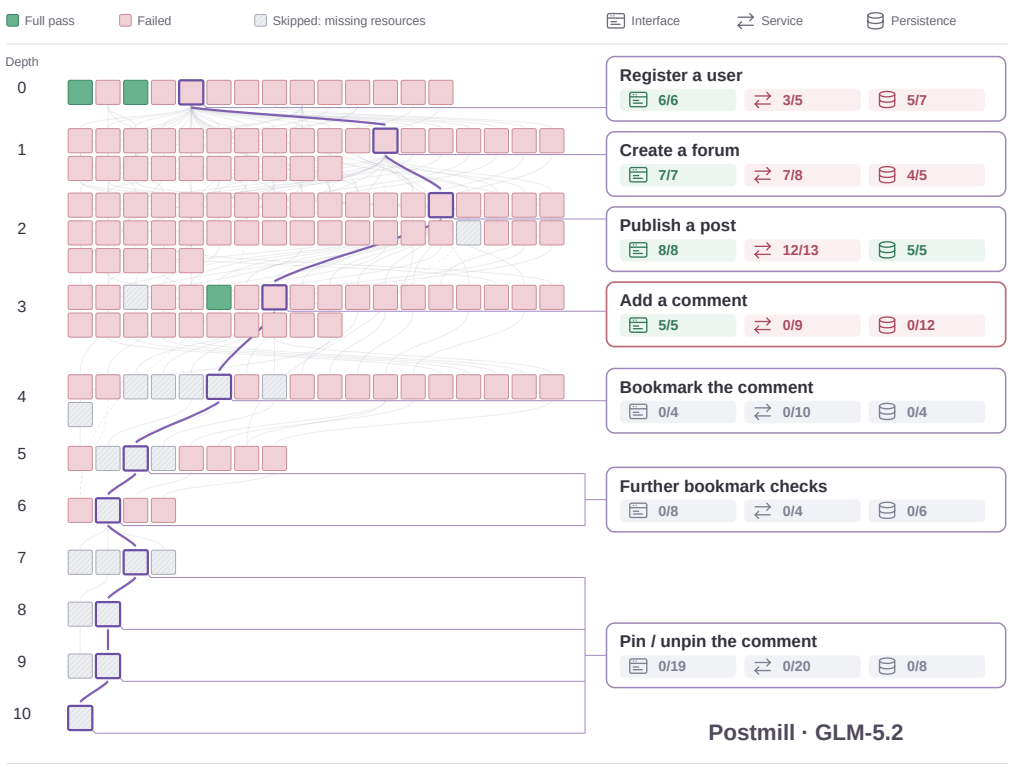

Figure 11: State-dependent evaluation of Postmill reconstructed by GLM-5.2. Failed executions do not block downstream tests whose required resources remain available; hatched nodes are skipped only when required resources cannot be resolved.

Interface test. The interface verifier executes the generated step-wise interface test by filling the seven fields, clicking Save All, and checking for a success notification and the entered real name.

```
{
  "feature": "Update User Profile",
  "steps": [
    {"step": 1, "action": "Navigate to {BASE_URL}/setting/profile"},
    {"step": 2, "action": "Type {GENERATED_USER_REAL_NAME} into the Real Name field"},
    {"step": 3, "action": "Type {GENERATED_USER_SCHOOL} into the School field"},
    {"step": 4, "action": "Type {GENERATED_USER_MAJOR} into the Major field"},
    {"step": 5, "action": "Select {GENERATED_USER_LANGUAGE} in the Language dropdown"},
    {"step": 6, "action": "Type {GENERATED_USER_MOOD} into the Mood field"},
    {"step": 7, "action": "Type {GENERATED_USER_BLOG} into the Blog field"},
    {"step": 8, "action": "Type {GENERATED_USER_GITHUB} into the Github field"},
    {"step": 9, "action": "Click the Save All button"},
    {"step": 10, "action": "Verify the page shows a success notification and the Real Name
        field displays {GENERATED_USER_REAL_NAME}"}
  ],
  "expected_outcome": "The profile settings page shows a success notification confirming the
      updated profile values were saved"
}
```

Listing 1: Generated step-wise interface test for the profile-update test.

Service test. The evaluation harness captures the HTTP exchange produced by the browser interaction and checks the method and response status. The generated assertions compare all seven request fields and all seven returned profile fields with the supplied values. They also check that the response reports no error and identifies the authenticated user by username.

```
def assert_request_path_params(path_str_without_query_params, check):
    return {}
```

```
def assert_request_query_params(query_params, check):
   return {}


def assert_request_body(body, check, GENERATED_USER_REAL_NAME, GENERATED_USER_SCHOOL,
     GENERATED_USER_MAJOR, GENERATED_USER_LANGUAGE, GENERATED_USER_MOOD, GENERATED_USER_BLOG,
     GENERATED_USER_GITHUB):
   def check_field(key, expected, name):
      actual = body.get(key)
      if actual == expected:
         check(name, "passed", "")
      else:
         check(name, "failed", f"expected {key}={expected!r}, got {actual!r}")

   check_field("real_name", GENERATED_USER_REAL_NAME, "request_real_name_matches")
   check_field("school", GENERATED_USER_SCHOOL, "request_school_matches")
   check_field("major", GENERATED_USER_MAJOR, "request_major_matches")
   check_field("language", GENERATED_USER_LANGUAGE, "request_language_matches")
   check_field("mood", GENERATED_USER_MOOD, "request_mood_matches")
   check_field("blog", GENERATED_USER_BLOG, "request_blog_matches")
   check_field("github", GENERATED_USER_GITHUB, "request_github_matches")
   return {}


def assert_response_body(body, check, GENERATED_USER_REAL_NAME, GENERATED_USER_SCHOOL,
     GENERATED_USER_MAJOR, GENERATED_USER_LANGUAGE, GENERATED_USER_MOOD, GENERATED_USER_BLOG,
     GENERATED_USER_GITHUB, SUPER_ADMIN_USERNAME):
   error = body.get("error")
   if error is None:
      check("response_error_is_null", "passed", "")
   else:
      check("response_error_is_null", "failed", f"expected null error, got {error!r}")

   data = body.get("data")
   if not isinstance(data, dict):
      check("response_data_object_exists", "failed", "response data is not an object")
      check("response_real_name_matches_supplied", "skipped", "response data object was
           missing")
      check("response_school_matches_supplied", "skipped", "response data object was missing"
           )
      check("response_major_matches_supplied", "skipped", "response data object was missing")
      check("response_language_matches_supplied", "skipped", "response data object was
           missing")
      check("response_mood_matches_supplied", "skipped", "response data object was missing")
      check("response_blog_matches_supplied", "skipped", "response data object was missing")
      check("response_github_matches_supplied", "skipped", "response data object was missing"
           )
      check("response_user_object_exists", "skipped", "response data object was missing")
      check("response_user_username_matches_authenticated_account", "skipped", "response data
            object was missing")
      return {}
   check("response_data_object_exists", "passed", "")

   def check_field(key, expected, name):
      actual = data.get(key)
      if actual == expected:
         check(name, "passed", "")
      else:
         check(name, "failed", f"expected data.{key}={expected!r}, got {actual!r}")

   check_field("real_name", GENERATED_USER_REAL_NAME, "response_real_name_matches_supplied")
   check_field("school", GENERATED_USER_SCHOOL, "response_school_matches_supplied")
   check_field("major", GENERATED_USER_MAJOR, "response_major_matches_supplied")
   check_field("language", GENERATED_USER_LANGUAGE, "response_language_matches_supplied")
   check_field("mood", GENERATED_USER_MOOD, "response_mood_matches_supplied")
   check_field("blog", GENERATED_USER_BLOG, "response_blog_matches_supplied")
   check_field("github", GENERATED_USER_GITHUB, "response_github_matches_supplied")

   user = data.get("user")
   if not isinstance(user, dict):
      check("response_user_object_exists", "failed", "response data.user is not an object")
      check("response_user_username_matches_authenticated_account", "skipped", "response user
            object was missing")
      return {}
   check("response_user_object_exists", "passed", "")

   username = user.get("username")
   if username == SUPER_ADMIN_USERNAME:
      check("response_user_username_matches_authenticated_account", "passed", "")
   else:
```

```
        check(
            "response_user_username_matches_authenticated_account",
            "failed",
            f"expected user.username={SUPER_ADMIN_USERNAME!r}, got {username!r}",
        )
    return {}


def assert_response_headers(headers, check):
    return {}
```

Listing 2: Generated service assertions for the profile-update test.

Persistence test. The generated assertions join `user_profile` with `user` and locate the profile by the authenticated username. After checking that exactly one profile is found, they independently verify all seven stored values. If the profile cannot be uniquely identified, the dependent field checks are marked as skipped.

```
from sqlalchemy import select


def assert_database(conn, metadata, check, SUPER_ADMIN_USERNAME, GENERATED_USER_REAL_NAME,
     GENERATED_USER_SCHOOL, GENERATED_USER_MAJOR, GENERATED_USER_LANGUAGE, GENERATED_USER_MOOD
     , GENERATED_USER_BLOG, GENERATED_USER_GITHUB):
    users = metadata.tables["user"]
    user_profile = metadata.tables["user_profile"]

    stmt = (
        select(
            user_profile.c.real_name,
            user_profile.c.school,
            user_profile.c.major,
            user_profile.c.language,
            user_profile.c.mood,
            user_profile.c.blog,
            user_profile.c.github,
        )
        .join(users, user_profile.c.user_id == users.c.id)
        .where(users.c.username == SUPER_ADMIN_USERNAME)
        .limit(2)
    )
    rows = conn.execute(stmt).mappings().all()

    if len(rows) != 1:
        check(
            "super_admin_profile_row_exists",
            "failed",
            f"expected exactly one user_profile row for username={SUPER_ADMIN_USERNAME!r}, found
                 {len(rows)}",
        )
        check("user_profile_real_name_updated", "skipped", "user_profile row was not uniquely
             identified")
        check("user_profile_school_updated", "skipped", "user_profile row was not uniquely
             identified")
        check("user_profile_major_updated", "skipped", "user_profile row was not uniquely
             identified")
        check("user_profile_language_updated", "skipped", "user_profile row was not uniquely
             identified")
        check("user_profile_mood_updated", "skipped", "user_profile row was not uniquely
             identified")
        check("user_profile_blog_updated", "skipped", "user_profile row was not uniquely
             identified")
        check("user_profile_github_updated", "skipped", "user_profile row was not uniquely
             identified")
        return

    check("super_admin_profile_row_exists", "passed", "")

    row = rows[0]

    def assert_field(column, expected, name):
        actual = row[column]
        if actual == expected:
            check(name, "passed", "")
        else:
            check(
               name,
```

```
            "failed",
            f"user_profile.{column} mismatch: expected {expected!r}, got {actual!r}",
        )

    assert_field("real_name", GENERATED_USER_REAL_NAME, "user_profile_real_name_updated")
    assert_field("school", GENERATED_USER_SCHOOL, "user_profile_school_updated")
    assert_field("major", GENERATED_USER_MAJOR, "user_profile_major_updated")
    assert_field("language", GENERATED_USER_LANGUAGE, "user_profile_language_updated")
    assert_field("mood", GENERATED_USER_MOOD, "user_profile_mood_updated")
    assert_field("blog", GENERATED_USER_BLOG, "user_profile_blog_updated")
    assert_field("github", GENERATED_USER_GITHUB, "user_profile_github_updated")
```

Listing 3: Generated persistence assertions for the profile-update test.

Runtime test data. The generated values include profile inputs, execution base URL, and authenticated username.

```
{
  "variables": {
    "BASE_URL": "http://target.website.com",
    "SUPER_ADMIN_USERNAME": "root",
    "GENERATED_USER_REAL_NAME": "Generated User 13b5e287",
    "GENERATED_USER_SCHOOL": "Ocean University 13b5e287",
    "GENERATED_USER_MAJOR": "Computer Science 13b5e287",
    "GENERATED_USER_LANGUAGE": "en-US",
    "GENERATED_USER_MOOD": "Practicing algorithms daily 13b5e287",
    "GENERATED_USER_BLOG": "https://blog-13b5e287.example.com",
    "GENERATED_USER_GITHUB": "https://github.com/coder-13b5e287"
  }
}
```

Listing 4: Generated runtime variables for the profile-update test.

The three artifacts verify one concrete execution from complementary perspectives. The interface test performs the edits and observes the save confirmation, the service test checks the submitted and returned values, and the persistence test checks the stored values for the authenticated account. For example, `GENERATED_USER_SCHOOL` must match the value entered into the School field, the request and response fields, and `user_profile.school`. These shared variables connect browser interaction, HTTP behavior, and database state within one execution.

K Evaluation Metric Definitions and Aggregation

This appendix formalizes the evaluation metrics introduced in Section 4.1. We report three primary outcome metrics that measure reconstruction success at complementary granularities. The **Solved Rate** is the percentage of websites on which a candidate achieves a 100% cross-layer test pass rate, indicating complete behavioral reconstruction. The **Almost-Solved Rate** uses a 95% threshold to capture candidates that reproduce nearly all evaluated behaviors. The **Cross-Layer Test Pass Rate** macro-averages the percentage of executable cross-layer tests passed across websites. To complement these binary outcomes, we report four fine-grained reward metrics. The **Cross-Layer Reward** assigns partial credit according to the assertions passed within each test, capturing behavioral correctness beyond binary test outcomes. The **Interface**, **Service**, and **Persistence Assertion Rewards** separately measure assertion-level performance at each verification layer. Together, these metrics distinguish complete reconstruction from partial progress and localize the remaining behavioral errors across system layers.

We next define their computation and aggregation. Let $\mathcal{W}$ denote the set of websites and $\mathcal{T}_w$ the set of scored executable cross-layer tests for website w. For test t and layer l, let $n_{w,t,l}$ and $c_{w,t,l}$ denote the numbers of required and passed assertions, respectively. Let $\mathcal{L}_{w,t}$ contain the interface, service, and persistence layers for which test t defines at least one required assertion.

Primary outcome metrics. Following the joint pass criterion for executable cross-layer tests, we define the binary outcome of test t as

$$z_{w,t} = \begin{cases} 1, & \text{if } c_{w,t,l} = n_{w,t,l} \text{ for every } l \in \mathcal{L}_{w,t}, \\ 0, & \text{otherwise.} \end{cases}$$

The website-level cross-layer test pass rate is

$$P_w = \frac{1}{|\mathcal{T}_w|} \sum_{t \in \mathcal{T}_w} z_{w,t}.$$

The **Cross-Layer Test Pass Rate** is the macro-average across websites:

$$P_{\text{test}} = \frac{1}{|\mathcal{W}|} \sum_{w \in \mathcal{W}} P_w.$$

The **Solved Rate** and **Almost-Solved Rate** are the fractions of websites attaining complete and near-complete reconstruction, respectively:

$$P_{\text{solved}} = \frac{|\{w \in \mathcal{W} : P_w = 1\}|}{|\mathcal{W}|}, \qquad P_{\text{almost}} = \frac{|\{w \in \mathcal{W} : P_w \geq 0.95\}|}{|\mathcal{W}|}.$$

Cross-layer reward. We first compute the assertion pass fraction of each applicable layer within a test:

$$a_{w,t,l} = \frac{c_{w,t,l}}{n_{w,t,l}}.$$

The reward of a cross-layer test is the mean of its applicable layer fractions:

$$r_{w,t} = \frac{1}{|\mathcal{L}_{w,t}|} \sum_{l \in \mathcal{L}_{w,t}} a_{w,t,l}.$$

We then average test rewards within each website and macro-average the resulting website rewards:

$$R_w = \frac{1}{|\mathcal{T}_w|} \sum_{t \in \mathcal{T}_w} r_{w,t}, \qquad R_{\text{cross}} = \frac{1}{|\mathcal{W}|} \sum_{w \in \mathcal{W}} R_w.$$

This aggregation assigns equal weight to applicable layers within a test, to tests within a website, and to websites within the benchmark.

Layer-specific assertion rewards. For each layer l, we first aggregate its assertions within website w:

$$R_{w,l}^{\text{assert}} = \frac{\sum_{t \in \mathcal{T}_w} c_{w,t,l}}{\sum_{t \in \mathcal{T}_w} n_{w,t,l}}.$$

Let $\mathcal{W}_l$ denote the websites containing at least one required assertion at layer l. The benchmark-level reward for that layer is

$$R_l^{\text{assert}} = \frac{1}{|\mathcal{W}_l|} \sum_{w \in \mathcal{W}_l} R_{w,l}^{\text{assert}}, \qquad l \in \{\text{interface}, \text{service}, \text{persistence}\}.$$

These definitions yield the **Interface**, **Service**, and **Persistence Assertion Rewards**. Unlike the Cross-Layer Reward, which balances layers within each test, each layer-specific reward measures assertion-level performance at one observable layer while retaining equal weight across websites. All rewards lie in $[0, 1]$, while rate metrics are reported as percentages.

L COMPLETE TASK PROMPT

```
# Task

You are a full-stack software engineering agent with access to a computer through
a shell, a browser, and harness-provided developer tools.

This is an interaction-grounded web application behavioral reconstruction task.
Build a new, original, runnable full-stack application from scratch in
`/home/agent/workspace` that reproduces the observable functional behavior of
the reference website running at:

`http://target.website.com`

Writing an original full-stack implementation based on systematically observed
behavior is the key point of this benchmark.

Observable behavior includes, but is not limited to:

- browser-accessible pages, controls, forms, and navigation;
- supported URL and page-transition behavior;
- HTTP requests, responses, status codes, and redirects;
- input validation and error behavior;
- authentication, authorization, sessions, and role-dependent behavior;
- state transitions across multiple user actions, page reloads, and application
  restarts.

Website operations must correctly create, update, and delete data in PostgreSQL,
and these changes must remain after page reloads and application restarts.

The reconstructed website must expose equivalent functionality through a real, usable
     interface.

# Task contracts

The benchmark provides two sanitized contracts extracted from the reference
application during benchmark construction:

- `/home/agent/task/endpoints.json` contains the required HTTP methods and
  paths exposed by the reference application.
- `/home/agent/task/database_schema.json` contains the required PostgreSQL
  table names, column names, database types, and primary keys derived from the
  reference application's database schema.

Treat these files as authoritative task contracts for the required HTTP routes
and PostgreSQL database structure. They do not reveal the reference
application's implementation details.

`endpoints.json` does not specify request fields, path or query parameter
semantics, request-body schemas, response schemas, status codes, redirects,
permissions, validation rules, or business rules. Discover these behaviors by
interacting with the reference website and inspecting browser-visible network
activity.

`database_schema.json` does not specify the meaning of each field, bootstrap
values, business invariants, or the state transitions that modify the database.
Discover those behaviors through normal interaction with the reference website.

The endpoint contract is a required HTTP surface, not a complete description of
the website. In particular, browser navigation routes and relationships between
pages and endpoints may not all be explicitly listed.

# Required exploration

Systematic exploration of the reference website is a required part of the task.

Do not stop after reproducing the landing page or a small number of common
workflows. Explore the complete reachable website, including small settings and
secondary features.

You should:

- visit and interact with all reachable pages;
- use the endpoint contract as a coverage checklist;
- inspect links, forms, controls, menus, and browser-visible network traffic;
- create accounts and application data through normal website workflows;
- explore different roles, permissions, and authentication states;
- examine empty, success, validation-error, and permission-denied states;
- test navigation, redirects, refresh behavior, and multi-step state changes;
```

```
- revisit earlier pages after mutations to observe their downstream effects;
- alternate between exploration, implementation, and testing until the clone
  behaves consistently.

You may use the browser, browser developer tools, shell commands, and ordinary
HTTP clients to observe behavior exposed through the public website.

# Required implementation

Your submission must be a genuine full-stack implementation consisting of:

- a functional browser-facing frontend;
- a backend that implements the observed application behavior;
- the required HTTP routes;
- persistent PostgreSQL-backed application state;
- database initialization or migration logic;
- any required bootstrap data;
- an executable startup script.

The required tables, columns, PostgreSQL types, and primary keys from
`database_schema.json` must exist in PostgreSQL. The application must perform
the database mutations required by its implemented workflows.

You may use files or process memory for static assets, uploaded file contents,
caches, sessions, temporary data, and other appropriate implementation details.
They must not replace the required PostgreSQL tables or the persistent
database-backed behavior represented by `database_schema.json`.

A static frontend, a collection of mocked pages, or an interface whose controls
do not execute real backend and database behavior is not a valid solution.

# What is allowed

- Use any programming language, framework, architecture, or combination of
  technologies available in the environment.
- Change your architecture or implementation during development.
- Rewrite or delete code you created and retry as many times as necessary.
- Use browser developer tools to inspect the DOM, accessibility information,
  requests, responses, headers, cookies, storage, console output, and public
  resources.
- Inspect and save resources that the reference website normally serves to a
  public browser, including HTML, CSS, JavaScript bundles, images, icons, and
  fonts.
- Register accounts and create data on the reference website through its normal
  public workflows.
- Write scripts to generate test data and assets such as images, documents, and
  text files.
- Download unrelated public materials for use as test inputs.
- Download publicly served assets from the reference website when useful.
- Browse general technical documentation and package documentation.
- Install and use ordinary third-party libraries and frameworks that are not
  the original application or a deployable copy of it.

# What is forbidden

Any submission that relies on the following approaches is not a valid behavioral
reconstruction and will be scored as a failure.

## 1. Obtaining the original implementation

You must not obtain, inspect, or reuse the reference application's original
source code or non-public implementation artifacts.

This includes, but is not limited to:

- locating or cloning its original source repository;
- accessing forks, mirrors, cached copies, or source archives;
- obtaining the original application through a package registry;
- downloading a prebuilt or deployable distribution of the original
  application;
- searching the internet specifically for its source code or internal
  implementation;
- attempting to access server-side files, containers, internal services, or the
  reference database.

## 2. Attacking the reference website

You must not exploit vulnerabilities, bypass infrastructure isolation, obtain
unauthorized server access, or use the reference website for anything other
than observing its public application behavior.
```

```
Ordinary interaction, HTTP requests, input variation, and validation or
permission testing through the public application interface are allowed.
Security exploitation and attempts to escape that interface are forbidden.

## 3. Wrapping or depending on the reference website

Your reconstructed website must operate independently.

You must not:

- proxy or relay requests to the reference website;
- embed the reference website in an iframe or similar container;
- reuse a live reference session as your backend;
- forward user actions to the reference website;
- make the submitted application depend on the reference website at runtime.

## 4. Faking the website with captured visuals or expected outputs

You MUST NOT use screenshots, screen recordings, videos, captured page
renderings, or image overlays as the implementation of any page or workflow.

You MUST NOT display a captured version of the reference website and place
transparent, invisible, or artificial click targets over it.

Screenshots and recordings may only be used as observational evidence during
development. They must never become part of the reconstructed website's
rendered implementation.

The submitted interface must be rendered by executable frontend code and backed
by real application logic. Any submission that implements the website by
replaying captured visuals is invalid and will be scored as a failure.

You must also not hard-code anticipated benchmark actions or expected outputs in
place of implementing the underlying behavior.

# Development environment

The container includes preinstalled Node.js, Python, Go, Java, Rust, Ruby, PHP,
Chrome, and user-space language package managers.

System packages cannot be added at task runtime. Prefer compatible wheels, pure
language implementations, preinstalled libraries, or dependencies that can be
built as the current user.

All available runtimes are on `PATH` and can be invoked directly:

| Language | Runtime | Package manager |
| --- | --- | --- |
| Python | `python3` (3.14) | `pip`, `uv` |
| Node.js | `node` (24) | `npm`, `npx` |
| Go | `go` (1.26) | `go mod` |
| Java | `java`, `javac` (Temurin 25) | `mvn` |
| Rust | `rustc`, `cargo` (1.96) | `cargo` |
| Ruby | `ruby` (4.0) | `gem`, `bundle` |
| PHP | `php` (8.4) | `composer` |

For Python, a virtual environment at `/home/agent/.pyvenv` is automatically
activated for shell commands.

Google Chrome is installed as `google-chrome-stable`. A headless Chrome instance
is already running with remote debugging available on port 9222.

Harness-specific instructions describe the exact browser and development tools
available to the current agent.

# Submission

Your submission must be a complete and clean deployable version of the
reconstructed website, suitable for a fresh deployment.

The reconstructed website must bind to `0.0.0.0:18000` and be reachable at:

`http://localhost:18000`

Keep all source code, dependency manifests, installed project dependencies,
virtual environments, generated assets, migration logic, and other required
files inside:

`/home/agent/workspace`

Create an executable startup script at:
```

````
`/home/agent/workspace/start.sh`

`start.sh` must initialize the reconstructed website, start all required
services, and keep the website running and reachable at
`http://localhost:18000` for the duration of the evaluation.

Your submission may include tests, test scripts, fixtures, generated test
assets, and development utilities. These development artifacts may remain in
the workspace, but `start.sh` must not automatically load or expose them as
live application data.

When deployed for evaluation, the website must begin in a clean initial
application state. It must not contain development, test, or demo accounts,
user-generated content, uploaded test files, sessions, queued jobs, or other
application state created during development.

The workspace is archived when the development run ends. Language download
caches and other files outside the workspace are not guaranteed to be archived.
Keep vendored runtime dependencies or reproducible installation steps inside the
workspace.

# Database

The reference website uses a private database. You cannot access that database
directly.

A separate PostgreSQL database is provided during development. You may freely
use it to create accounts, content, uploads, and other data while developing and
testing the reconstructed website. The contents of this development database
are not used during evaluation.

Use the following connection information without modifying it:

```text
PGHOST=postgres-agent
PGPORT=5432
PGUSER=agent
PGPASSWORD=agent
PGDATABASE=agentdb
```

During evaluation, the reconstructed website is provided with a new PostgreSQL
service backed by a fresh empty data directory. Before `start.sh` is executed,
this database contains no application tables or application data.

`start.sh` must create all database tables, columns, PostgreSQL types, primary
keys, and essential system or bootstrap records required by the reconstructed
website.

After initialization, the database must be in a clean initial application
state. It may contain the system and bootstrap records required for the website
to operate, but it must not contain development, test, demo, or user-generated
application data.

`start.sh` must also be safe to run when the database has already been
initialized. In that case, it may apply necessary migrations, but it must not
delete, reset, or reseed existing application data.
````

M SELECTED OPEN-SOURCE WEBSITES

Table 16: Overview of the selected open-source websites. We report the frontend and backend frameworks, repository file count and net lines of code, and the numbers of detected web endpoints and database tables. The italic line below each entry summarizes the application.

		Tech Stack		Code Volume		Functionality	
Repository	**Version**	**Frontend**	**Backend**	**Files**	**Net LoC**	**Endpoints**	**Tables**
benotes	`v2.7.1`	**Vue.js**	**Laravel**	306	48,499	36	11
Self-hosted application for organizing markdown notes and saved links in collections, with metadata extraction, sharing, and backups.							
booking-calendar	`59fa91a`	**Twig**	**Symfony**	114	16,774	19	7
Appointment booking system with a public calendar and administrative workflows for slots, bookings, settings, confirmations, and reminders.							
booklogr	`v1.11.1`	**React**	**Flask**	263	50,963	46	11
Personal library tracker for reading lists, progress, ratings, notes, quotes, and optional public profile sharing.							
cachet	`v2.4.1`	**Blade, Vue.js**	**Laravel**	1,241	118,376	134	25
Status page application for publishing service components, incidents, metrics, subscriber notifications, and operational updates.							
DjangoDash	`b1f7450`	**Django Templates**	**Django**	137	8,677	62	13
CRM for managing leads, agents, and categories with dashboards, search, pagination, exports, password reset, and administration.							
Filmcase	`ea0b968`	**Templates, HTMX**	**Django**	705	66,346	54	20
Application for managing Fujifilm camera recipes and browsing a JPEG catalog with EXIF matching, filtering, rating, and recipe cards.							
fugu	`v1.1.5`	**ERB, Hotwire**	**Rails**	224	7,462	35	8
Self-hosted product analytics platform for tracking events, managing projects, and exploring funnels through a web dashboard and API.							
haven	`5b108c5`	**ERB**	**Rails**	295	10,107	98	16
Private blogging platform for multimedia posts, invited readers, and following other blogs through a built-in private RSS reader.							
Heimdall	`v2.8.2`	**Blade**	**Laravel**	402	87,796	69	13
Application dashboard and launcher for organizing links to web applications and services with tiles, search, and enhanced integrations.							
hours	`d695140`	**Haml**	**Rails**	398	12,468	62	13
Project-based time tracker for registering hours by project, category, and tag and analyzing how teams spend their time.							
ihatemoney	`7.2.1`	**Jinja2**	**Flask**	235	55,212	45	11
Shared budget manager for project members, weighted bills, settlement computation, history, statistics, and REST API access.							
jawanndenn	`4.1.0`	**React**	**Django**	160	13,942	31	14

		Tech Stack		Code Volume		Functionality	
Repository	**Version**	**Frontend**	**Backend**	**Files**	**Net LoC**	**Endpoints**	**Tables**
Self-hosted meeting scheduler and polling application with anonymous poll creation, voting, and automatic poll expiry.							
keila	`v0.30.2`	**LiveView**	**Phoenix**	1,067	127,416	151	30
Email newsletter and campaign platform for contacts, lists, forms, templates, mailings, and delivery integrations.							
listaway	`v1.18.2`	**HTML Templates**	**Go net/http**	106	10,983	40	7
List publishing application for individuals or teams to create, share, and publicly publish collections of items.							
neith	`66e32d6`	**templ, HTMX**	**Go net/http**	70	4,148	13	4
Personal milestone tracker for registered users to create milestones, track their status, and manage a profile.							

OnlineJudge	`oj_2.7.5`
	`api.1.6.1`
	`judge.1.6.1`
		Vue.js	**Django**	457	46,456	62	20

Programming contest and judging platform for problems, submissions, users, contests, rankings, and automated code evaluation.							
Postmill	`7ef756f`	**Twig, HTMX**	**Symfony**	1,196	87,369	160	31
Link aggregation and discussion platform for communities, posts, comments, moderation, user accounts, and voting.							
wallabag	`2.6.14`	**Twig**	**Symfony**	794	88,227	149	16
Read-it-later application for saving articles, extracting readable content, tagging, annotating, and exporting e-books.							
wedding	`8a14419`	**ERB, ActiveAdmin**	**Rails**	170	3,810	58	6
Wedding information and guest management application with secret-link RSVPs and an administrative attendee interface.							
whoami	`2.0.0`	**ERB, Hotwire**	**Rails**	369	11,811	62	25
Personal profile and portfolio platform for links, experience timelines, blog posts, public pages, and RSS feeds.							