GAMEASG-BENCH: BENCHMARKING AUTONOMOUS SOFTWARE GENERATION FOR GAME DEVELOPMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Autonomous software generation (ASG) aims to turn human requirements into executable applications, but delivering these applications does not necessarily establish that their interacting components satisfy the specified behavioral requirements. We introduce GAMEASG-BENCH, a benchmark that makes behavioral testability part of the generation task for game development. Our design declares an evaluation interface specification before generation, fixing legal starting scenarios, player-level actions, stable snapshots, rejection behavior, and invariants while leaving private implementations open. Concretely, we include: (i) static L1 checks that assess source-level compliance; and (ii) browser-executed L2 checks that combine semantic observations with real input and runtime evidence. We implement this protocol as 47 browser-native game-generation tasks spanning 12 primary genres and both 2D and 3D interaction, each with executable checks and an independently verified reference implementation. Experiments show that our predefined interface enables more accurate and efficient behavioral evaluation. We evaluate nine agent stacks on this benchmark and find that the highest observed mean L2 check pass rate is 93.2%, yet the highest observed strict task success rate, requiring all L1 and applicable L2 prerequisite and core requirement checks, is only 55.3% (26/47 tasks). Further comparisons examine the effects of tool access and harness choice. Together, these evaluations support the predefined-interface design and reveal task-level compliance gaps that high average check pass rates can obscure.

1 INTRODUCTION

Autonomous software generation (ASG) aims to translate human requirements into complete, executable software artifacts with minimal human intervention, extending the scope of agentic coding from local implementation tasks to end-to-end development (Liu et al., 2026). Modern coding agents pursue this goal by combining large language models (LLMs) with iterative implementation, execution, and revision (Yang et al., 2024). Success on bounded programming problems or repository-level edits, however, does not establish that an agent can deliver an application whose interacting components collectively satisfy the original requirements (Liu et al., 2026; Zhang et al., 2026c). We therefore ask: *how can we systematically evaluate whether frontier LLMs, operating through contemporary coding-agent loops, can autonomously generate a complete, executable application whose runtime behavior satisfies the specified behavioral requirements?* We study browser-native game generation as a bounded and behavior-dense proxy for autonomous requirement-to-artifact generation. The resulting benchmark measures compliance with specified evaluation requirements.

Browser-native games provide a bounded setting in which these integration requirements are directly observable. Even compact games must coordinate player input, state transitions, rendering, and gameplay rules within one executable artifact (Zhang et al., 2026c; Luo et al., 2026). Plausible individual mechanics can still produce an unusable game when these components fail to work together (Chen et al., 2026). Games therefore offer a concrete setting for assessing requirement-to-artifact generation, but an aggregate "playable" judgment provides limited evidence about which requirements remain unmet. Evaluation must connect observed behavior to individual requirements to distinguish partial implementation from successful delivery and diagnose specific failures (Jia et al., 2026; Xu et al., 2026).

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.

These integration requirements pose challenges for both generation and evaluation. During generation, the agent must translate natural-language requirements into coherent game logic, coordinating player input, state transitions, and visible feedback (Luo et al., 2026). Evaluation must then establish whether these components jointly realize the specified behavior: successful compilation alone is insufficient, while open-ended playtesting can miss conditions requiring long or precise interaction sequences (Liu & Tatar, 2026; Jia et al., 2026). This calls for evaluation that can reach relevant gameplay conditions and connect player actions to observable outcomes, providing evidence of compliance with individual requirements.

Recent work advances behavioral evaluation of generated games through mechanisms for controlled execution and observation. WebGameBench tests game behavior through real-browser interaction (Zhang et al., 2026c); GameCraft-Bench uses scenario initialization and replayable demonstrations to obtain multimodal evidence (Luo et al., 2026); and GameGen-Verifier decomposes specifications into precondition-interaction-postcondition checks, using runtime state injection to establish test conditions in each generated implementation (Jia et al., 2026). Together, these approaches demonstrate ways to reach relevant gameplay conditions and obtain evidence of the resulting behavior.

Building on these advances, we ask how generation can establish consistent access for behavioral evaluation across different implementations. We introduce GAMEASG-BENCH, which makes behavioral testability part of the generation task: task authors define the evaluation interface, its semantics, and executable checks *before generation*, and coding agents implement the interface together with the game. This enables fixed checks to combine semantic observations with real browser input and runtime evidence to assess requirement compliance. Our contributions are:

Contribution 1. We make behavioral testability part of the generation task by specifying an evaluation interface and its semantics before generation (Section 2). The agent implements this interface together with the game, enabling fixed checks to assess different private implementations through semantic observations, real player input, and browser runtime evidence. Note that this design has been verified by experiments to enable more accurate and efficient behavioral evaluation.

Contribution 2. We instantiate this design as GAMEASG-BENCH, a benchmark of 47 browser-native game-generation tasks spanning 12 primary genres and both 2D and 3D environments (Section 3). Each task includes specifications, executable checks, and an independently verified reference implementation, supporting evaluation from requirements to complete, executable games.

Contribution 3. Our evaluation of nine agent stacks reveals a gap between average check performance and full requirement compliance: the highest observed mean L2 check pass rate is 93.2%, while the highest strict task success rate is only 55.3% (26/47 tasks). Controlled comparisons further examine how tool access and harness choice affect performance, with trace analysis highlighting differences in workflow emphasis (Section 4).

2 BENCHMARK DESIGN AND EVALUATION PROTOCOL

2.1 TASK DEFINITION AND OVERVIEW

Each GAMEASG-BENCH task requires a coding agent to produce a self-contained browser-native game in `index.html` that satisfies both player-visible requirements and an evaluation interface specification. Three documents define the task:

- **Generation prompt (`target.md`):** specifies the workspace, delivery protocol, and a concise gameplay brief.
- **Gameplay design requirement (`game-spec.md`):** defines the playable loop, mechanics, player feedback, and completion conditions.
- **Evaluation interface specification (`tdd.md`):** defines scenario setup, action and observation semantics, rejection behavior, and invariants.

Task authors prepare all three documents and executable checks before generation. All documents are available to the agent; the harness passes `target.md` directly, and the agent consults the two specifications to implement the game and interface together. The evaluator verifies delivery and

applies fixed source-level (L1) and behavioral (L2) checks, recording individual outcomes and strict task success. The specifications constrain observable behavior while leaving internal implementation choices open. Execution details and access boundaries appear in Section 4.1 and Appendix E.

2.2 Evaluation Interface Specification

The evaluation interface provides consistent ways to prepare scenarios, execute actions, and observe state across different generated implementations of the same task. Each specification defines four methods exposed through `window.__gameTest`: `reset`, `loadScenario`, `input`, and `getSnapshot`. These method names are shared across tasks; scenario names, supported actions, and snapshot fields are defined for each task, with fixed meanings across its implementations. The interface provides two complementary capabilities.

Reproducible starting conditions. The `reset` operation restores the initial state and clears transient effects. `loadScenario` establishes a documented legal state reachable through ordinary play, including rare or late-game preconditions. Scenario setup may adjust resources and positions to establish these conditions, but must not directly produce the outcome being tested. For example, a near-terminal scenario must still require subsequent play to trigger victory or defeat. This design avoids repeatedly executing lengthy preparation sequences while ensuring that the tested outcome still arises from subsequent gameplay.

Actions and observations with stable meanings. The `input` operation executes player-level semantic actions, such as ordering a unit or selecting a target. The `getSnapshot` operation returns a JSON-serializable summary of relevant game state. The specification fixes action and observation semantics, including how invalid actions are rejected and which state properties must remain unchanged. These shared semantics allow pre-authored checks to evaluate different implementations without requiring the agent to expose its private object graph or follow a prescribed code layout.

The specification requires all four methods to remain consistent with the state that drives actual gameplay: `reset`, `loadScenario`, and `input` must operate on that state, and `getSnapshot` must faithfully report the corresponding observations. This requirement connects scenario preparation, action execution, and outcome inspection to the running game. Section 2.3 describes how behavioral checks combine interface observations with real input and browser evidence to assess this consistency.

2.3 Layered Evaluation and Behavioral Evidence

Evaluation combines source-level compliance checks (L1) with browser-based behavioral checks (L2). L1 runs first, but its failures do not prevent L2 execution, allowing both layers to report their outcomes.

L1: source-level compliance. The runner inspects the submitted HTML and its inline or directly linked local scripts using structural and syntax checks, regular-expression assertions, and anti-pattern rules defined in `checks.json`. It records each check's outcome and diagnostic hints.

L2: controlled behavioral execution. The evaluator serves the game locally and runs `checks.js` in headless Chromium, with checks executed sequentially in fresh pages. Behavioral checks follow a prepare–act–observe sequence: they can establish a documented scenario, invoke a semantic action or send real browser input, and compare observed changes with expected outcomes and invariants. Snapshots provide task-specific state observations, complemented by browser activity, rendered output, and runtime errors. Real input and browser observations help assess whether reported state changes are reflected in visible gameplay. Each check returns `PASS`, `FAIL`, or `NOT_APPLICABLE`, with elapsed time and diagnostic details.

2.4 Requirement Priorities and Scoring

Based on our design, L1 and L2 distinguish the evidence layers used by checks. Within L2, requirement priorities distinguish prerequisite checks (P0), core requirement checks (P1), and extended capability checks (P2). P0 covers launch, evaluation-interface availability, and minimum runtime requirements. P1 covers required mechanics, interactions, invariants, and supporting interface requirements. P2 covers additional mechanics and experience completeness. These priorities are assigned during test authoring.

We define strict task success as valid delivery, completed evaluation, and passing every L1 check and every applicable L2 P0/P1 check. P2 checks characterize extended capabilities and are not required for strict success. L2 checks marked `NOT_APPLICABLE` are excluded from the applicable-check set and its pass-rate denominator.

For each planned task g, let $d_g, e_g \in \{0, 1\}$ indicate valid delivery and completed evaluation, respectively. Let $\mathcal{L}_g$ contain all L1 checks and $\mathcal{B}_g$ contain the applicable L2 P0/P1 checks. For each check c, let $p_{g,c} = 1$ if it passes and 0 otherwise. Strict task success s_g and the strict task success rate over the planned task set $\mathcal{G}$ are

$$s_g = d_g e_g \prod_{c \in \mathcal{L}_g \cup \mathcal{B}_g} p_{g,c}, \qquad \mathrm{SR}(\mathcal{G}) = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} s_g. \tag{1}$$

The strict task success rate uses all planned tasks as its denominator, counting generation or evaluation failures as unsuccessful. Check pass rates are computed over tasks that complete evaluation: mean L1 and overall L2 rates average per-task pass rates, while per-priority L2 rates pool applicable checks across tasks. All rates are reported as percentages.

3 Benchmark Construction

We instantiate the design in Section 2 through task selection, specification and test authoring, and quality control, yielding 47 tasks, each with executable checks and a reference implementation.

3.1 Scope and Task Selection

Task selection considers playability, delivery constraints, and evaluation feasibility. Each task must support a complete playable loop in which player actions interact with game rules to produce observable progress toward an objective or a terminal outcome, together with a way to restart. Tasks must fit the self-contained `index.html` delivery protocol; we exclude concepts requiring a backend, user accounts, external databases, paid or private assets, or unbounded multiplayer infrastructure. The behaviors selected for evaluation must also be reachable and observable within bounded browser execution. This scope focuses evaluation on the integration of controls, state transitions, rendering, and gameplay logic.

3.2 Specification and Test Authoring

For each selected game concept, human developers specify the gameplay requirements and combine a gameplay brief with delivery instructions to form the generation prompt. They then define scenarios, actions, and snapshot fields that make the requirements testable under the interface constraints in Section 2.2. Each tested behavior is specified through its starting conditions, player-level action, expected observable outcomes, and invariants, including rejection behavior and preserved state for invalid actions. Based on these specifications, developers author L1 and L2 checks and assign requirement priorities following Section 2.4.

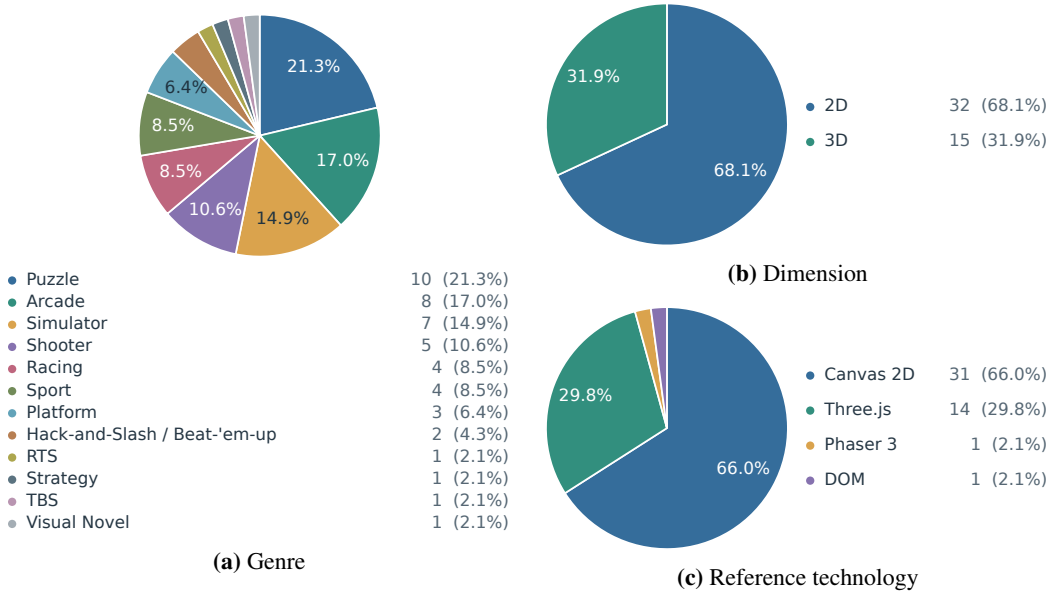
3.3 Quality Control

Quality control proceeds in two stages: manual verification of reference implementations against the task specifications, followed by automated acceptance testing. For each task, manual verification assesses compliance with the gameplay design requirements and evaluation interface specification through real user interactions, core state transitions, termination and restart, and consistency between interface observations and visible gameplay. This establishes a basis for judging the reference implementation beyond its automated check results. The authored L1 and L2 checks are then run on the verified implementation. Acceptance requires passing all L1 checks and all applicable L2 P0/P1 checks. The manually verified implementations thus serve as positive controls for assessing whether the test suite accepts these core-compliant reference games.

3.4 Corpus Composition

The benchmark comprises 47 tasks spanning 12 primary genres, with 32 in 2D and 15 in 3D. Figure 1 summarizes their distributions by genre, dimension, and reference rendering technology.

Across the 47 tasks, the test suite contains 336 L1 checks and 885 L2 checks, with the latter comprising 102 prerequisite (P0), 534 core requirement (P1), and 249 extended capability (P2) checks. Table 6 in Appendix A provides the detailed breakdown.



(a) Genre

(b) Dimension

(c) Reference technology

Figure 1 Composition of the 47 tasks in GameASG-Bench. Each panel covers all 47 tasks.

4 Experimental Evaluation

Our experiments address the following research questions:

(**RQ1**) *Does our evaluation interface enable more accurate and efficient game testing?* (Section 4.2).

(**RQ2**) *How well do contemporary agent stacks satisfy end-to-end task requirements?* (Section 4.3).

(**RQ3**) *How does tool access affect task performance?* (Section 4.4).

(**RQ4**) *How does harness choice affect task performance?* (Section 4.5).

4.1 Experimental Setup

Tasks and configurations. All experiments are based on the task specifications in GameASG-Bench. RQ1 compares evaluation methods on a fixed set of artifacts and behavioral requirements, while RQ2–RQ4 evaluate agent stacks across all 47 tasks using the same task packages. Each generation run starts from a clean workspace, with one run per task–configuration pair. The resulting artifacts are evaluated using the benchmark checks in headless Chromium with a fixed 1280×800 viewport. An *agent stack* is a model paired with its coding harness: Claude Code 2.1.206 or Codex CLI 0.153.4. Model and execution configurations are specified in the corresponding subsections. Additional experiments on turn budget and reasoning effort are reported in Appendices B and C, respectively.

Evaluation metrics. RQ1 measures evaluation completion, judgment accuracy, and preparation and total evaluation time, as detailed in Section 4.2. RQ2–RQ4 measure requirement compliance using strict task success and L1/L2 check pass rates, following the scoring rules in Section 2.4. Strict task success uses all 47 planned tasks as its denominator (Equation 1), while check pass rates cover only

tasks that complete evaluation. We also report delivery and evaluation completion counts, along with available measurements of artifact size, token use, and monetary cost.

Reference-implementation validation. Following the verification procedure in Section 3.3, we evaluated one independently verified core-compliant reference implementation for each of the 47 tasks. All 47 passed all L1 checks and all applicable L2 P0/P1 checks.

4.2 RQ1: Effectiveness of Predefined Evaluation Interfaces

We compare three methods for evaluating the same behavioral requirements on frozen game artifacts: *computer-use* through screenshots and real input, *source adaptation* through implementation-specific preparation and observation code inspired by GameGen-Verifier (Jia et al., 2026), and *predefined interface* (ours), which uses the evaluation interface specified before generation and the original benchmark checks. The experiment covers 8 games and 72 paired instances. Both model-driven methods use GPT-6-Astra with high reasoning effort through Codex CLI. Details are provided in Appendix F. Each test involves two steps: establishing the required preconditions and testing the target behavior. The results are reported in Table 1.

Table 1: Evaluation-method comparison on the same 72 instances. Accuracy uses all 72 instances.

Method	**Preparation**		**Evaluation**		
	Completed ↑ (count)	**Mean time** ↓ (s)	**Completed** ↑ (count)	**Mean total time** ↓ (s)	**Accuracy** ↑ (%)
Computer-use	66	80.165	57	200.939	75.0
Source adaptation	66	104.200	64	189.692	81.9
Predefined interface (ours)	**72**	**4.593**	**72**	**6.548**	**100.0**

The predefined interface achieves the highest evaluation completion rate and judgment accuracy. Our method outperforms computer-use and source adaptation on both metrics (Table 1), indicating that the predefined interface supports completing behavioral tests and correctly determining whether the tested requirements are satisfied.

Reusing predefined access reduces preparation and evaluation time. Our method achieves approximately $30.7\times$ and $29.0\times$ speedups over computer-use and source adaptation, respectively, measured by mean total evaluation time. Predefined access allows the evaluator to reuse state-preparation and observation procedures across implementations, reducing the work needed to reach test conditions or construct implementation-specific access after generation.

Table 2: End-to-end performance at maximum reasoning effort. Strict task success is shown as *count/47 (percentage)*; remaining values are check pass rates (%).

Model	**Harness**	**Strict task success**	**L1**	**L2**			
				Mean	**P0**	**P1**	**P2**
GPT-6-Astra	Codex CLI	**26/47 (55.3)**	98.0	**93.2**	99.0	**92.7**	**92.4**
Claude-Opus-5	Claude Code	24/47 (51.1)	**99.6**	90.4	91.2	89.9	91.1
GPT-5.6-Sol	Codex CLI	21/47 (44.7)	98.9	91.3	**100.0**	89.7	91.9
DeepSeek-V4-Flash	Claude Code	18/47 (38.3)	99.4	89.3	98.0	86.7	90.2
DeepSeek-V4-Pro	Claude Code	15/47 (31.9)	**99.6**	85.2	85.3	83.9	88.9
Kimi-K3	Claude Code	15/47 (31.9)	97.7	88.5	98.0	86.3	89.7
GLM-5.2	Claude Code	11/47 (23.4)	98.8	84.6	91.2	83.3	83.0
Hunyuan-3	Claude Code	10/47 (21.3)	98.3	81.7	96.1	78.2	82.3
MiniMax-M3	Claude Code	7/47 (14.9)	98.3	70.6	91.2	64.5	75.4

4.3 RQ2: End-to-End Agent Performance

We evaluate nine agent stacks at their maximum reasoning-effort settings with full tool access. DeepSeek-V4-Flash uses Claude Code with a turn budget of 120, specifying the maximum number of allowed agent turns.

All nine agent stacks deliver artifacts and complete evaluation on all 47 tasks. Table 2 reports strict task success and check pass rates at maximum reasoning effort, and Table 3 reports artifact size and resource use. We identify two key observations:

High average check pass rates coexist with incomplete task compliance. Mean L1 check pass rates range from 97.7% to 99.6%, whereas strict task success rates range from 14.9% to 55.3%. GPT-6-Astra with Codex CLI has the highest observed mean L2 check pass rate (93.2%), yet achieves strict task success on only 26/47 tasks (55.3%). Strict success requires passing both source-level and behavioral checks: only 40 of its 47 artifacts pass every L1 check. Reporting both check pass rates and strict task success therefore distinguishes partial compliance from satisfying all required checks.

Table 3: Mean per-task artifact size and resource use across 47 tasks for each agent stack.

Model	**Harness**	**Artifact size** (KB)	**Input tokens** (k)	**Output tokens** (k)	**Monetary cost** (USD)
GPT-6-Astra	Codex CLI	78.82	896.1	55.5	4.4445
Claude-Opus-5	Claude Code	68.52	12,632.6	142.4	11.8500
GPT-5.6-Sol	Codex CLI	64.67	1,382.5	45.7	1.7867
DeepSeek-V4-Flash	Claude Code	85.21	13,038.4	203.6	0.2461
DeepSeek-V4-Pro	Claude Code	90.34	12,944.3	202.9	0.7431
Kimi-K3	Claude Code	57.51	1,263.9	72.3	2.5518
GLM-5.2	Claude Code	60.33	1,067.1	47.0	0.6184
Hunyuan-3	Claude Code	52.88	2,474.5	88.9	0.1779
MiniMax-M3	Claude Code	81.76	4,918.9	71.1	0.4069

Larger artifacts and higher token usage do not consistently correspond to more strict task successes. GPT-6-Astra records 55.5k mean output tokens and achieves 26 strict task successes, compared with approximately 203k output tokens for each DeepSeek variant and 18 and 15 strict task successes for Flash and Pro, respectively. MiniMax-M3 produces larger artifacts than Claude-Opus-5 (81.76 versus 68.52 KB) but achieves 7 rather than 24 strict task successes. DeepSeek-V4-Pro produces the largest artifacts (90.34 KB) and achieves 15 strict task successes. These contrasts show that larger artifacts or higher reported token usage do not consistently correspond to more complete task compliance.

4.4 RQ3: Effect of Tool Access

We vary tool access for DeepSeek-V4-Flash at maximum reasoning effort. The full-tools baseline uses Claude Code with a turn budget of 120, as in Section 4.3. The no-tools condition uses serialized API input instead of the file-based harness workflow.

Table 4: Tool configurations for DeepSeek-V4-Flash. All 47 tasks complete evaluation in each setting. Syntax checking permits Bash for `node --check`, but no browser.

Tool setting	**Strict task success**	**L1**	**L2 P0**	**L2 P1**	**L2 P2**
No tools (serialized API)	7/47 (14.9)	98.5	83.3	65.3	80.1
File read/write only	6/47 (12.8)	98.8	87.3	66.2	73.4
Files + syntax checking	9/47 (19.1)	98.2	86.3	63.4	69.3
Full tool access	**18/47 (38.3)**	**99.4**	**98.0**	**86.7**	**90.2**

Full tool access achieves the most strict task successes and the highest check pass rates. Full tool access achieves 18 strict task successes, compared with 6–9 under restricted configurations (Table 4). Its P1 check pass rate reaches 86.7%, versus 63.4%–66.2% for restricted configurations, while L1 rates remain close to 99% throughout. The configurations thus differ substantially in core behavioral compliance despite similar source-level pass rates. Appendix D.2 complements these aggregate results with concrete traces of tool-assisted testing and repair.

4.5 RQ4: Effect of Harness Choice

We compare DeepSeek-V4-Flash under Claude Code and Codex CLI at maximum reasoning effort with full tool access. The Claude Code condition reuses the 120-turn baseline from Section 4.3. We compare the harnesses as complete execution environments, including their system instructions, context management, tool schemas, command execution, and endpoint protocols.

Table 5: Harness comparison for DeepSeek-V4-Flash at maximum reasoning effort. Strict task success is shown as *count/47 (percentage)*; remaining values are check pass rates (%).

Harness	Strict task success	L1	L2 P0	L2 P1	L2 P2
Claude Code	**18/47 (38.3)**	**99.4**	**98.0**	86.7	90.2
Codex CLI	**18/47 (38.3)**	99.2	96.1	**87.1**	**92.2**

Equal aggregate success conceals differences in outcomes and resource use. Both harnesses achieve strict success on 18/47 tasks (38.3%; Table 5), but only ten tasks succeed under both. Codex CLI also produces smaller artifacts on average (69.5 versus 85.2 KB) and uses fewer output tokens (112.8k versus 203.6k). The equal success rate thus summarizes runs with different successful task sets and resource profiles, motivating closer inspection of their workflows.

The same model shows different workflow emphases. Qualitative inspection of matched traces suggests that Codex CLI emphasizes contract checks and targeted repairs to state transitions, boundary conditions, and simulation logic. Claude Code also performs such repairs, but places additional emphasis on repeatable gameplay trajectories and late-stage refinement of animation, audio, and visual feedback. Both commonly begin with a substantial implementation and prioritize API-level behavior before adding separate input checks. These observations suggest different emphases within a shared implement–test–repair workflow. Appendix D.2 illustrates these differences with paired *Sortie* traces.

4.6 Diagnostic Analysis

We examine two diagnostic cases to illustrate the evaluator's ability to detect subtle behavioral failures that can escape isolated self-tests. Specifically, real-input checks expose failures in player interaction, while checks combining scenario preparation with real input reveal errors in subsequent runtime behavior.

Real-input checks expose failures in player interaction. In the GPT-6-Astra/Codex CLI implementation of *Diner Dasher*, the artifact passes L1 and L2 P0/P2 checks, yet real mouse and touch drags fail to advance the service workflow. The L2 P1 checks detect this failure by requiring player inputs to produce observable service progress. This case illustrates why passing checks for other requirements cannot substitute for directly verifying core player interactions.

Scenario-based checks expose failures in combined operations. In the DeepSeek-V4-Flash/Claude Code implementation of *Turbo Smash Beast*, reset or scenario loading stops natural-time simulation, preventing subsequent driving inputs from advancing the vehicle. The agent's isolated self-tests miss this interaction: evaluation-interface tests explicitly advance time, while a successful driving test starts from a freshly loaded page. By combining scenario preparation, real input, and natural time progression, the evaluator exposes acceleration and coasting failures that neither test covers. This case demonstrates the value of checking whether individually tested operations remain correct when executed together (Appendix D.1).

5 Related Work

5.1 Evaluating Autonomous Software Generation

Language-model code synthesis and execution-guided learning (Chen et al., 2021; Le et al., 2022) underpin longer development workflows. Coding agents combine synthesis with inspection, execution, and revision, making the delivered artifact a product of both the model and its development environment (Yang et al., 2024; Zhang et al., 2024; Qian et al., 2024). Evaluation covers bounded programs (Hendrycks et al., 2021; Liu et al., 2023), changes to existing repositories (Jimenez et al., 2024; Li et al., 2025), and complete applications with broader integration requirements (Lu et al., 2026; Liu et al., 2026; Pan et al., 2026; Ren et al., 2026). CodeFlowBench and KoCo-Bench extend coverage to reuse, dependencies, and domain knowledge (Wang et al., 2026; Jiang et al., 2026a), while visual and interactive evidence further extends application-level assessment (He et al., 2026; Guo et al., 2026). Complementing artifact-generation benchmarks, WebArena evaluates agents' ability to complete tasks in existing web environments (Zhou et al., 2024).

Games provide a concrete setting for evaluating such integrated artifacts. Game-generation systems study joint rule-and-level synthesis (Hu et al., 2024) and organize development through specialized roles, conversation, or engine-grounded workflows (Chen et al., 2023; Hong et al., 2025; Yin et al., 2026). Execution feedback supports iterative construction and learning (Ma et al., 2026; Huang et al., 2026; Zhang et al., 2026a; Jiang et al., 2026b), with verification also used in model distillation and training (Serapio et al., 2026; Zhou et al., 2026). Benchmark tasks range from scoped development in existing engine projects (Chi et al., 2026; La et al., 2026) to project generation and completion (Sun et al., 2026) and text-to-game generation (Zhang et al., 2026b).

5.2 Behavioral Evaluation of Generated Games

Evaluating generated games requires both criteria for judging behavior and a way to exercise the relevant conditions. Related work on web applications addresses complementary aspects: VideoVIBE evaluates failure diagnosis from human-operated recordings with source-code context (Xu et al., 2026), while FlowCheck compiles user-visible information-flow constraints into deterministic CodeQL analyses (Vir et al., 2026). Game evaluators combine play, runtime, structural, and visual evidence (Peng et al., 2026; Huang et al., 2026; Liu & Tatar, 2026; Jiang et al., 2026b). Their criteria take different forms: GameCraft-Bench scores replayable demonstrations against a hidden rubric (Luo et al., 2026); GameGen-Verifier extracts precondition–interaction–postcondition keypoints (Jia et al., 2026); and the generation track of GameXpert-Bench constructs shared event rubrics after generation by pooling events from completed artifacts and incorporating human review (Chen et al., 2026).

Exercising these criteria also requires reaching the relevant states and connecting actions to observable outcomes. WebGameBench uses specification-guided browser interaction and permits candidate state preparation before the final user-level action (Zhang et al., 2026c). GameCraft-Bench supports scenario initialization for replayable demonstrations (Luo et al., 2026), while GameGen-Verifier grounds its keypoints in each generated implementation through runtime state injection (Jia et al., 2026). GameXpert-Bench verifies events through code inspection and live interaction (Chen et al., 2026). These mechanisms provide different ways to establish test conditions and obtain behavioral evidence.

6 Conclusion

In this paper, we introduce GameASG-Bench, which makes behavioral testability part of autonomous game generation through an evaluation interface specification declared before generation. Its 47 browser-native tasks combine source-level checks with browser-executed checks using semantic observations, real input, and runtime evidence. Our experiments support defining evaluation interfaces before generation to enable more accurate and efficient behavioral evaluation. Across nine agent stacks, the highest observed mean L2 check pass rate is 93.2%, yet the highest observed strict task success rate is only 55.3%, exposing the gap between passing most checks and satisfying every required check. Comparisons of tool access and harness choice further reveal configuration-dependent outcomes. The benchmark provides a common protocol and task-level diagnostics for studying requirement compliance.

AI USE STATEMENT

We used generative AI tools to refine wording and grammar and polish existing text, assist with literature discovery and information retrieval, draft portions of the manuscript including the abstract and introduction, and generate synthetic datasets used in this study. All AI-generated synthetic data were manually verified. The authors take full responsibility for the final content of the paper.

REFERENCES

Dake Chen, Haoyang Zhang, Hanbin Wang, Yunhao Huo, Yuzhao Li, and Junjie Wang. Gamegpt: Multi-agent collaborative framework for game development. *arXiv preprint arXiv:2310.08067*, 2023.

Kun Chen, Haorong Hong, Peizhong Gao, Jianfeng Lin, Tongxu Luo, Yuxuan Xie, Chenxu Liu, Jieling He, Zhongyuan Liu, and Zeno Zeng. Gamexpert-bench: How far are coding agents from expert game development? *arXiv preprint arXiv:2608.21833*, 2026.

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

Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, et al. Gamedevbench: Evaluating agentic capabilities through game development. *arXiv preprint arXiv:2602.11103*, 2026.

JunJia Guo, Yuhang Yao, Jingdi Chen, et al. Vista: An end-to-end benchmark for visual spec-to-web-app coding agents. *arXiv preprint arXiv:2605.26144*, 2026.

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. *arXiv preprint arXiv:2603.26648*, 2026.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Jiale Hong, Hongqiu Wu, and Hai Zhao. Game development as human-llm interaction. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4333–4354, 2025.

Chengpeng Hu, Yunlong Zhao, and Jialin Liu. Game generation via large language models. In *2024 IEEE conference on games (CoG)*, pp. 1–4. IEEE, 2024.

Yixu Huang, Bo Li, Na Li, Zhe Wang, Kaijie Chen, Haonan Ge, Qingyi Si, Yuanzhe Shen, Ruihan Yang, Guangjing Wang, et al. Gui agents for continual game generation. *arXiv preprint arXiv:2605.28258*, 2026.

Chaobo Jia, Ruipeng Wan, Ting Sun, Weihao Tan, Borui Wan, Yuxuan Tong, Guangming Sheng, and Hong Xu. Gamegen-verifier: Parallel keypoint-based verification for llm-generated games via runtime state injection. *arXiv preprint arXiv:2605.07442*, 2026.

Xue Jiang, Ge Li, Jiaru Qian, Xianjie Shi, Chenjie Li, Hao Zhu, Ziyu Wang, Jielun Zhang, Zeyu Zhao, Kechi Zhang, et al. Koco-bench: Can large language models leverage domain knowledge in software development? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28422–28441, 2026a.

Yilei Jiang, Jinyuan Hu, Qianyin Xiao, Yaozhi Zheng, Ruize Ma, Kaituo Feng, Jiaming Han, Tianshuo Peng, Kaixuan Fan, Manyuan Zhang, et al. Opengame: Open agentic coding for games. *arXiv preprint arXiv:2604.18394*, 2026b.

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

Brian La, Sejoon Chang, Ben Kim, Junyoung Bae, Aamish Ahmad Beg, Sei Chang, Gonzalo Gonzalez-Pumariega, and Kanav Goyal. Gameenginebench: Evaluating coding agents on real c++ runtime environments. *arXiv preprint arXiv:2607.03525*, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. Fea-bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17160–17176, 2025.

Hugh Xuechen Liu and Kıvanç Tatar. Mage: Multi-axis evaluation of llm-generated executable game scenes beyond compile-pass rate. *arXiv preprint arXiv:2605.07342*, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Jingyao Liu, Chen Huang, Zhizhao Guan, Wenqiang Lei, and Yang Deng. E2edev: Benchmarking large language models in end-to-end software development task. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 35032–35068, 2026.

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. *Advances in Neural Information Processing Systems*, 38, 2026.

Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, et al. Gamecraft-bench: Can agents build playable games end-to-end in a real game engine? *arXiv preprint arXiv:2606.17861*, 2026.

Hongnan Ma, Han Wang, Shenglin Wang, Tieyue Yin, Yiwei Shi, Yucong Huang, Yingtian Zou, Muning Wen, and Mengyue Yang. Creativegame: Toward mechanic-aware creative game generation. *arXiv preprint arXiv:2604.19926*, 2026.

Ruwei Pan, Yakun Zhang, Qingyuan Liang, Yueheng Zhu, Chao Liu, Lu Zhang, and Hongyu Zhang. Toward functional and non-functional evaluation of application-level code generation. *arXiv preprint arXiv:2602.03462*, 2026.

Zhiyuan Peng, Wei Tao, Xin Yin, Chenhao Ying, Yuan Luo, and Yiwen Guo. Playcoder: Making llm-generated gui code playable. *Proceedings of the ACM on Software Engineering*, 3(FSE): 2003–2026, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 15174–15186, 2024.

Qingnan Ren, Shun Zou, Shiting Huang, Ziao Zhang, Kou Shi, Zhen Fang, Yiming Zhao, Yu Zeng, Qisheng Su, Lin Chen, et al. Saasbench: Exploring the boundaries of coding agents in long-horizon enterprise saas engineering. *arXiv preprint arXiv:2605.17526*, 2026.

Tyrone Serapio, Arjun Prakash, Haoyang Xu, Kevin Wang, and Amy Greenwald. Distilling game code world model generation into lightweight large language models. *arXiv preprint arXiv:2605.24375*, 2026.

Jianwen Sun, Chuanhao Li, Zizhen Li, Yukang Feng, Fanrui Zhang, Yifei Huang, Yu Dai, and Kaipeng Zhang. Jamer: Project-level code framework dataset and benchmark on professional game engines. *arXiv preprint arXiv:2606.19830*, 2026.

Reya Vir, Lydia Chilton, Zhuo Zhang, and Eugene Wu. Flowcheck: Helping end-users specify and verify intent in vibe-coded web apps. *arXiv preprint arXiv:2608.28880*, 2026.

Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. Codeflowbench: A multi-turn, iterative benchmark for complex code generation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4369–4402, 2026.

Jiajun Xu, Yanghao Zhou, Jingyun Liao, Yu Bai, Jinxing Zhou, Chengliang Liu, Changsen Yuan, Bo Wang, and Qian Liu. Videovibe: A video-grounded diagnostic benchmark for one-shot interactive website generation. *arXiv preprint arXiv:2608.09573*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Lei Yin, Wentao Cheng, Zhida Qin, Tianyu Huang, Yidong Li, and Gangyi Ding. Autoue: Automated generation of 3d games in unreal engine via multi-agent systems. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2341–2364, 2026.

Jiajun Zhang, Yuheng Jing, Zeyu Cui, Hao Zheng, Wentao Chen, Kaixin Li, Jiaxi Yang, Tianbao Xie, Zeyao Ma, Tianyi Bai, et al. Bringing code alive: Optimizing interactive frontend mini-games via automated play and reinforcement learning at scale. In *Forty-third International Conference on Machine Learning*, 2026a.

Wei Zhang, Jian Yang, Renshuai Tao, Linzheng Chai, Shuyue Guo, Jiajun Wu, Xiaoming Chen, Ganqu Cui, Ning Ding, Xander Xu, et al. V-gamegym: Visual game generation for code large language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 5613–5641, 2026b.

Wenyu Zhang, Guoliang You, Haotian Zhao, Tianshu Zhu, Haoran Wang, Xiaoxuan Tang, Mingyang Dai, Jingnan Gu, Daxiang Dong, Jianmin Wu, et al. Webgamebench: Requirement-to-application evaluation for coding agents via browser-native games. *arXiv preprint arXiv:2605.17637*, 2026c.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Chenyu Zhou, Qiliang Jiang, Shuning Wu, and Xu Zhou. The verifier is the curriculum: Precision sets the return on search in code self-distillation. arXiv preprint arXiv:2607.09709, 2026. URL `https://arxiv.org/abs/2607.09709`.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A Benchmark Check Statistics

Table 6 summarizes the checks across the 47 tasks using the evidence layers and requirement priorities defined in Section 2.

Table 6: Check counts for the 47 tasks used in all reported experiments. Layer and priority describe different properties of a check.

L1: source-level checks		**L2: behavioral checks**	
Tool checks	43	Prerequisite checks (P0)	102
Regular-expression checks	288	Core requirement checks (P1)	534
Anti-pattern checks	5	Extended capability checks (P2)	249
Total	**336**	**Total**	**885**

B Effect of Turn Budget

We vary the turn budget for DeepSeek-V4-Flash while retaining full tool access and the other baseline settings in Section 4.4.

Table 7: Turn-budget comparison for DeepSeek-V4-Flash with full tool access. The primary success rate uses all 47 planned tasks; conditional success rates and check pass rates are computed over evaluated tasks only.

Turn budget	**Evaluated / planned**	**Strict task success**		**L1**	**L2 P0**	**L2 P1**	**L2 P2**
		All planned	**Conditional**				
30	10/47	7/47 (14.9)	7/10 (70.0)	100.0	100.0	94.0	97.6
60	32/47	13/47 (27.7)	13/32 (40.6)	99.7	88.6	81.6	88.5
120	47/47	**18/47 (38.3)**	18/47 (38.3)	99.4	98.0	86.7	90.2

Larger turn budgets increase evaluation completion and strict task success rates. At turn budgets of 30, 60, and 120, respectively, 10, 32, and 47 tasks complete evaluation, yielding 7, 13, and 18 strict successes out of all 47 planned tasks (Table 7). Claude Code halts at the configured turn limit, leaving more runs unfinished before delivery under shorter budgets; Appendix D.3 illustrates such cutoffs during verification and finalization. These results indicate that larger turn budgets improve end-to-end task success in this setting, with more runs completing generation and delivery.

C Effect of Reasoning Effort

We compare low, high, and maximum reasoning effort for DeepSeek-V4-Flash across 47 tasks, with all other settings matching its RQ2 baseline. Table 8 reports task performance and mean reasoning-token use as a measure of computational cost.

Higher reasoning effort improves check-level performance, but does not consistently yield more strict task successes. P1 check pass rates increase from 71.7% at low effort to 86.7% at maximum effort, whereas strict task successes rise from 7/47 at low effort to 19/47 at high effort, then fall to 18/47 at maximum effort. High effort also uses 26.9% fewer reasoning tokens than maximum effort. Thus, although maximum effort achieves the highest P0, P1, and P2 pass rates, high effort offers a more favorable trade-off between strict task success and reasoning-token use in these runs. These results suggest that additional reasoning helps satisfy individual behavioral requirements, but does not consistently ensure that the same artifact passes every check required for strict task success.

Table 8: Reasoning-effort comparison for DeepSeek-V4-Flash. Reasoning-token counts are averaged across tasks.

Effort	Strict task success	L1	L2 P0	L2 P1	L2 P2	Reasoning tokens
Low	7/47 (14.9)	98.3	93.1	71.7	80.2	37,699
High	**19/47 (40.4)**	**99.4**	97.1	82.7	88.9	138,232
Max	18/47 (38.3)	**99.4**	**98.0**	**86.7**	**90.2**	189,210

D Supplementary Trace Analysis

This appendix records selected diagnostic examples from evaluation reports and generation traces supporting the experimental discussion. The cases illustrate failure and repair mechanisms. Unless otherwise specified, DeepSeek-V4-Flash cases use the Claude Code baseline with full tool access, maximum reasoning effort, and a turn budget of 120. Strict task success has the benchmark-compliance meaning defined in the main text.

D.1 Runtime and Interaction Failures

The following cases supplement the real-input and scenario-based diagnostics in Section 4.6. They describe the prepared state, the interaction exercised, and the resulting check outcome.

Real-input service progression. In *Diner Dasher* generated by GPT-6-Astra with Codex CLI, two L2 P1 checks load `tray_with_correct_item`, then obtain the tray item's and customer's screen bounds from the snapshot. One check sends a real mouse drag between their centers; the other sends the corresponding touch sequence. Each compares the resulting snapshot with the prepared state and requires an increase in service progress or earnings. Both fail at this progress assertion: neither real drag advances the service workflow. The checks also test tray consumption and visible rendering changes after successful service, but these runs fail before reaching those assertions. The artifact passes L1 and L2 P0/P2, showing why exposing a callable interface and a readable playfield does not establish that the required serving interaction works.

Scenario preparation and natural-time progression. In the DeepSeek-V4-Flash/Claude Code implementation of *Turbo Smash Beast*, reset or scenario loading enables test mode and suppresses natural-time simulation. The generation trace exercises two different paths: self-tests through the evaluation interface advance simulation time explicitly, while a successful real-input driving test starts from a freshly loaded page. The benchmark instead prepares a scenario and then exercises real driving input with natural time progression. In this sequence, the simulation remains stalled, and acceleration, coasting, and related real-input checks fail. The failing sequence combines operations that the separate self-tests do not exercise together.

D.2 Execution-Guided Repairs

These traces supplement the tool comparison in Section 4.4 and the harness workflow analysis by connecting execution feedback to specific edits and subsequent checks.

Feedback under different tool settings. For DeepSeek-V4-Flash's *Diner Dasher*, the file read/write only configuration supports a correction of touch-release coordinates through source rereading; the final artifact passes the corresponding mouse and touch checks. In the run with full tool access, Chromium mouse input exposes a different defect: a hidden completion screen missed by stub-based tests. Correcting the screen-state mapping restores that transition. These runs illustrate the distinct repairs prompted by source inspection and browser interaction.

Syntax repair and behavioral verification. In the *Turbo Smash Beast* run with files + syntax checking, `node --check` catches an unclosed construct. Closing it restores parsing, but acceleration and coasting still fail benchmark evaluation. The syntax feedback resolves a parsing defect while leaving the gameplay failures observable to behavioral checks.

State-progression repair. In the *Garden Gulp* run with full tool access, a browser-executed progression probe using the evaluation interface reveals that the hole's actual size never catches up with its target size. Repairing the update loop allows the probe to complete the level. This feedback links an evolving game-state discrepancy to a concrete update-loop repair.

Browser-specific rendering repair. In DeepSeek-V4-Pro's *Neon Flow*, browser execution exposes a negative-radius Canvas error missed by Node-based tests. The agent repairs the rendering path and verifies it again; the final artifact satisfies strict task success. This sequence illustrates feedback from executing the artifact with the browser's rendering API.

Different repairs under two harnesses. Both maximum-effort DeepSeek-V4-Flash *Sortie* runs achieve strict task success after execution-guided repairs. In Codex CLI, 27 self-tests through the evaluation interface pass before a real mouse drag fails. Inspection reveals that item positions are rendered relative to a layer below the heads-up display while the snapshot reports viewport coordinates; correcting this offset is followed by successful real-input retesting. In Claude Code, self-tests expose stale item states surviving reset because layout reconstruction preserves state indiscriminately. Restricting that preservation to resize is followed by a 148/148 self-test pass. Later edits repair the snap animation and adjust audio initialization before the final regression checks. The paired traces show how both harnesses support testing and repair, with different defects and follow-up edits in these runs. The self-test counts refer to different agent-authored tests.

D.3 Turn-Budget Trace Details

Claude Code stops execution when the configured turn limit is reached. The following cases supplement the turn-budget comparison in Appendix B by showing how this cutoff can interrupt verification or finalization before delivery is complete.

Cutoff before verification executes. *Ancient Beast* reaches the cutoff in both shorter-budget conditions. One 60-turn attempt stops immediately after writing a browser harness, before running it. The trace thus ends after preparing a verification step but before obtaining its execution feedback.

Cutoff after self-tests pass. One 30-turn *Barbie and Ken's Puzzles* attempt continues with screenshot processing after its self-tests pass, then reaches the cutoff without completing delivery. This case shows that passing the agent's self-tests does not itself complete the generation and delivery workflow.

E Generation and Evaluation Implementation Details

Generation execution and delivery preflight. Each attempt uses the original task documents in a clean workspace and a fresh session. Claude Code runs in prompt mode with model and turn-budget arguments and stops when the configured turn limit is reached. Codex CLI uses an ephemeral session with workspace write access and JSON tracing. The host records execution traces, exit status, elapsed time, failure reasons, artifact status, and available usage measurements. The container launcher also monitors elapsed time and periods without trace updates to terminate timed-out runs. Evaluation requires a successful generation-process exit and a regular, non-symlink, nonempty `index.html` containing `</html>`. Artifacts that pass this preflight proceed to L1 and L2 evaluation.

Task access and evaluation isolation. The launcher mounts the generation workspace with read/write access and marks the copied task documents read-only. The coding agent receives the generation prompt, gameplay design requirement, and evaluation interface specification (`tdd.md`); executable checks and evaluation runners are not mounted in the generation container. After generation ends, checks run in a separate evaluation container with the submitted HTML and tests mounted read-only.

F Evaluation-Method Comparison Details

This appendix provides the protocol and additional analysis for the paired evaluation-method comparison in Section 4.2.

Paired evaluation instances. We randomly select eight games: *Diner Dasher*, *Sortie*, *Garden Gulp*, *Ancient Beast*, *City Rider*, *Fruit Smoothie Merge*, *Tankor Arena*, and *Penalty King*. For each game, we use one manually verified positive implementation and two implementations from the

multi-model generation results. Each of these 24 artifacts is evaluated at the same three test points for its game, yielding 72 implementation–test-point instances per method. The test points cover basic interactions, specific intermediate states, and terminal, progression, or state-locking behavior. Each method receives the same frozen artifact and tests the same requirement, preconditions, target operation, and expected outcome.

Evaluation methods. *Computer-use* uses screenshots and real mouse and keyboard input to reach the required state, perform the target interaction, and judge the visible outcome, without reading source code or calling the evaluation interface. *Source adaptation* reads the generated source and writes, executes, and debugs implementation-specific preparation and observation code after generation. It cannot reuse the existing `__gameTest` interface or benchmark check helpers. *Predefined interface* uses the interface semantics specified before generation and the benchmark's original checks, including documented scenarios and any additional legal preparation actions.

Execution protocol. Both model-driven methods use GPT-6-Astra with high reasoning effort through the local Codex CLI. Each instance runs independently, with a 600-second preparation budget and a 900-second total budget.

Metrics and verification. Table 1 reports five metrics. Preparation is complete when the required legal state is reached without prematurely producing the tested outcome. Test completion requires executing the target operation, observing its outcome, and returning definite `PASS`/`FAIL` judgments for all assigned checks; uncertain or missing judgments count as incomplete.

Judgment accuracy is the fraction of all 72 instances that return complete, definite judgments matching the historical reference results for every assigned check. Uncertain judgments, timeouts, and missing results do not count as correct. The comparison results were manually verified.

Preparation time measures the interval from the start of preparation until the required preconditions are first established. Its mean uses only valid timing records: 66 instances for each model-driven method and 72 for the predefined interface. Total evaluation time includes preparation and runs until result submission or termination, including incomplete instances. The reported mean covers all 72 instances per method; for the predefined interface, it is computed from the total batch execution time divided by 72. Waiting time between the two source-adaptation batches is excluded.