A2Z GAMESPEC-BENCH: HOW FAITHFULLY CAN CODING AGENTS GENERATE GAMES FROM GAME DESIGN SPECIFICATIONS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Delegating complete application development to coding agents requires preserving the intended design rather than simply producing plausible outputs through naïve prompting. Game development provides a demanding testbed, as long-form Game Design Documents (GDDs) describe requirements that must work together across game logic, visual rendering, and player interactions. However, existing game-development benchmarks typically use compact specifications and provide limited support for evaluating interdependent requirements across these aspects in long-form GDDs. We introduce **A2Z GameSpec-Bench**, a benchmark of 100 long-form GDDs for evaluating end-to-end game development by agents. We measure ***faithfulness*** by checking whether the game satisfies the GDD requirements and preserves the relationships among them. Each GDD is turned into a DEPENDENCY-AWARE CONTRACT that contains rules, constraints, and prerequisite relations. Following game-development practices, we combine source-code inspection with agent-generated TEST POLICIES for scenario-based replay and adaptive playtesting. The contract remains fixed across agents and revision rounds, while judgments and evidence linked to the same requirements support consistent comparison and failure detection. Our evaluations show that current agents struggle to jointly satisfy interdependent requirements across code implementation and actual play. Requirement-specific feedback improves *GDD Fidelity* by 10.9% relative gain over *self-revision* after two rounds. A2Z GameSpec-Bench assesses end-to-end specification-following ability beyond implementation judgments and provides targeted feedback to support more faithful game development.

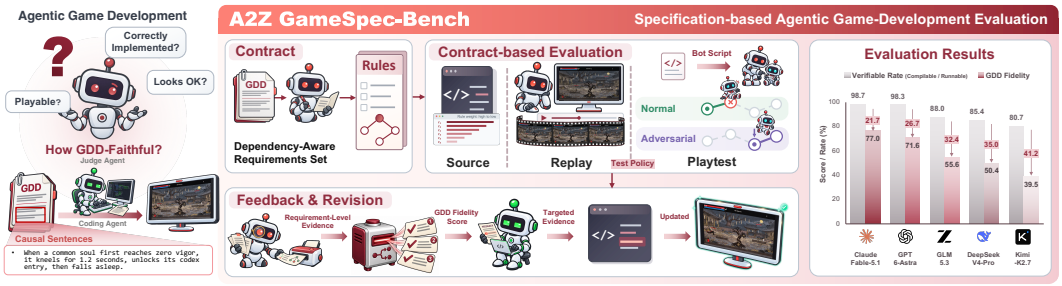
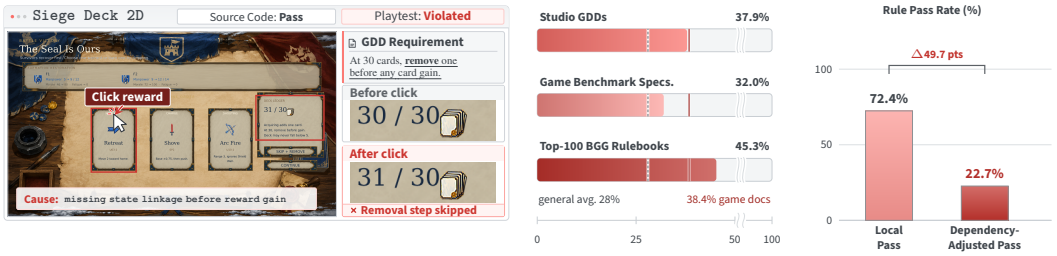


Figure 1: **Overview of A2Z GameSpec-Bench.**

1 INTRODUCTION

Coding agents are increasingly moving from isolated programming tasks to complete application development (Qian et al., 2024; Yang et al., 2026), including playable games from natural-language prompts (Jiang et al., 2026). However, generating a plausible game from a naïve prompt differs significantly from implementing the game that a designer intends. In studio development, delegating implementation requires agents to interpret detailed specifications, connect individual features, and deliver complete games end-to-end without designers repeatedly directing each low-level step. Therefore, specification-driven development is important for retaining design control while scaling the work delegated to the agents. Games provide a demanding testbed for this ability in agents: specified designs must be satisfied across game logic, visual rendering, and player interactions.

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.



(a) Cross-Axis Failure Case **(b)** Causal Sentence Ratio **(c)** Rule Dependencies

Figure 2: **Challenges in specification-driven game development.** (a) In our game `Siege Deck 2D`, the deck-limit requirement gets full source-code credit, but playtesting reveals a reward gain without the card removal. (b) Across three game-design corpora with 353 documents, CiRA (Fischbach et al., 2021) classifies 38.4% of sentences as causal on average, compared with 28% for general documents (Frattini et al., 2023) (Appendix B.1). (c) Source-code pass rates decrease when all upstream rules must also pass (Appendix B.7).

In game development, long-form specifications called Game Design Documents (GDDs), including studio documents (Hamilton, 1995), often serve as a shared reference for implementing and verifying the intended design (Callele et al., 2005; Salazar et al., 2012). Although their formats vary, their requirements provide the basis for assessing implementation and commonly involve causal dependencies (Figure 2b). In practice, developers verify the results using code-level checks, visual testing of specific scenarios, and quality assurance (QA) playtesting (Kisel, 2016; Epic Games, n.d.). These checks must follow the same requirements and account for relations in which one rule's effects satisfy another's conditions. They should also identify requirement-specific mismatches that guide revisions toward the intended design. Thus, faithfulness requires preserving the specified rules and their relationships across implementation, rendered behavior, and actual play. This raises the main question: *How **faithfully** can coding agents implement design specifications as a complete game?*

Existing game-development benchmarks assess specified behavior through replay rubrics (Luo et al., 2026), state-initialized tests (Jia et al., 2026), or source-code inspection (Chi et al., 2026). However, their input specifications are substantially compact, and they provide limited support for assessing long-form GDDs through a common representation of requirement dependencies across implementation, rendering, and continuous play (Table 1). As shown in Figure 2a, a single evaluation axis used in existing benchmarks such as source-code inspection can miss design violations that our playtests detect during runtime. This implies that evaluating fidelity to long-form GDDs needs tracking dependencies and checking the same requirements through code, visual, and execution evidence.

We introduce **A2Z GameSpec-Bench**, a benchmark of 100 long-form GDDs for evaluating end-to-end specification-driven game development. Each task asks a coding agent to deliver a source project and playable build from a GDD to measure its faithfulness defined as *GDD Fidelity*. The corpus contains 50 *Small* designs with compact scope and 50 *Big* designs with extensive content. We construct these GDDs from game briefs through an agentic authoring pipeline that checks for omissions and inconsistencies (Appendix A). For evaluation, each GDD is turned into a DEPENDENCY-AWARE CONTRACT that records individual rules, constraints, and prerequisite relations between rules. Constructed from the GDD and kept fixed across agents and revision rounds, the contract makes these relationships explicit rather than treating requirements as an independent checklist.

Then, we organize evaluation from code-level checks and visual assessment to QA playtesting. Alongside source-code inspection, agents construct requirement-specific TEST POLICIES that determine how to exercise the game and collect evidence against the contract. Scenario replays reproduce specified situations, while trace-guided frame selection retrieves visual evidence of the expected responses. For adaptive playtesting, CODE-AS-POLICY (Liang et al., 2023) bots select player inputs based on the current state. Normal play checks how behaviors connect from the initial state, while targeted adversarial tests establish preconditions for unverified requirements and check their subsequent effects. Consequently, agents construct and execute tests to verify each game based on our contract as summarized in Figure 1. The resulting judgments and evidence are linked to the same requirements for GDD Fidelity scoring and further revision feedback.

As shown in Figure 2c, our code-level evaluation shows that locally passing rules can still depend on failed prerequisites. Across 100 GPT-5.6-Sol games, the mean pass rate for dependency-linked rules is 72.4% from local source-code judgments but 22.7% when all upstream rules must also pass, show-

Table 1: **Comparison of agentic game development benchmarks.** A2Z GameSpec-Bench fixes a dependency-aware evaluation contract from long-form GDDs independently of generated outputs and links judgments from complementary evidence channels to the same requirements.

Benchmark	Benchmark Setup			Requirement Evaluation				Evaluation Channels		
	Full-Game Generation	Input Type	# of Spec. Tokens	Evaluation Target	Relations	Predefined	Cross-Axis	Source Code	Rendered Behavior	Adaptive Play
GameDevBench	✗	Task + project	185	Task-specific tests	✗	✓	✗	✓	✗	✗
GameEngineBench	✗	Spec. + project	–	Tests + LLM judge	✗	✓	✗	✓	✗	✗
OpenGame-Bench	✓	Game spec.	1,830	Build / visual / intent	✗	✓	✗	✗	✓	✗
WebGameBench	✓	Structured spec.	–	Runtime quality	✗	✓	✗	✗	✓	✓
GameCraft-Bench	✓	Game spec.	1,547	Predefined rubric	✗	✓	✗	✗	✓	✗
PlaytestArena	✓	Short prompt	131	Behavior rubric	✗	✓	✗	✗	✓	✓
GameGen-Verifier	✗	Spec. + project	7,998	Sparse keypoints	✓	✓	✓	✓	✓	✗
GameXpert-Bench†	✓	Design brief	84	Post-hoc event rubric	✗	✗	✓	✓	✗	✓
A2Z GameSpec-Bench	✓	**Long-form GDD**	**20,191**	**GDD contract set**	✓	✓	✓	✓	✓	✓

† We report the GameGen track; human ratings are omitted from this table. Spec. tokens are rounded mean lengths of accessible specification using `o200k_base` without project code; PlaytestArena is author-reported. – means that specs. cannot be publicly obtained. For *Adaptive Play*, ✓denotes closed-loop evaluator interaction in which the next action is selected online based on the current observation or runtime state.

ing a graph-based reachability proxy (Appendix B.7). By checking the same requirements through replay and playtests, A2Z GameSpec-Bench further exposes design mismatches in visual output and execution. With the same initial builds and revision, requirement-specific feedback improves *GDD Fidelity* over revision by the agent itself. A2Z GameSpec-Bench supports systematic comparison of coding agents' specification-following ability and targeted revision toward the intended design.

2 Related Work

Specification-Based Evaluation. Evaluating generative outputs against detailed specifications often starts through decomposing the specifications into concrete criteria. In text-to-image evaluation, TIFA (Hu et al., 2023) and VPEval (Cho et al., 2023) decompose text prompts into fine-grained visual checks and evaluate them using visual question answering and specialized modules, respectively. In text-to-CAD generation, MUSE (Dong et al., 2026) pairs design instances with structured specifications and evaluates generated CAD models through code execution, geometric validation, and VLM-based design-intent alignment. For games, these criteria may not be independent: an unmet prerequisite can prevent downstream behavior from being exercised or observed at runtime.

Structured Evaluation Contracts. Davidsonian Scene Graph (Cho et al., 2024) links prompt-derived questions through prerequisite entities, while ComplexBench (Wen et al., 2024) aggregates instruction-following judgments according to constraint composition. DafnyCOMP (Xu et al., 2026) studies whether specifications generated for individual functions remain valid when functions interact. This provides a formal-verification analogue, not direct evidence of failures in generated games. For web artifacts, WebRISE (Meng et al., 2026) represents requirements as Interaction Contract Graphs over observable states, transitions, and predicates. WebGrader (Chen et al., 2026a) first plans required flows from the request, then uses source code and the live DOM to ground executable Flow Contracts and evidence-based verdicts. It develops these graders for reinforcement learning. Our contracts apply related principles to GDDs and preserve the distinction between local tests under supplied preconditions and evidence of the specified gameplay paths.

Agentic Game Development and Evaluation. Benchmarks for agentic game development range from scoped game-engine tasks to complete game generation. GameDevBench (Chi et al., 2026) and GameEngineBench (La et al., 2026) evaluate agents on scoped tasks in existing game projects. At the full-game level, OpenGame-Bench (Jiang et al., 2026) scores browser-native games on build health, visual usability, and intent alignment, while WebGameBench (Zhang et al., 2026) evaluates games from specifications through browser interaction. GameCraft-Bench (Luo et al., 2026) replays traces and assesses fixed-rate sampled frames against a hidden multi-modal rubric. PlaytestArena uses a GUI agent to play browser games against behavior rubrics, and Play2Code uses its feedback for iterative game revision (Huang et al., 2026). Orak (Park et al., 2026), GameWorld (Ouyang et al., 2026), and OmniGameArena (Lin et al., 2026) instead evaluate game-playing agents in predefined games, which has limitations on adaptive runtime test for newly generated games. GameGen-Verifier (Jia et al., 2026) extracts verifiable keypoints from a game specification, initializes their required run-

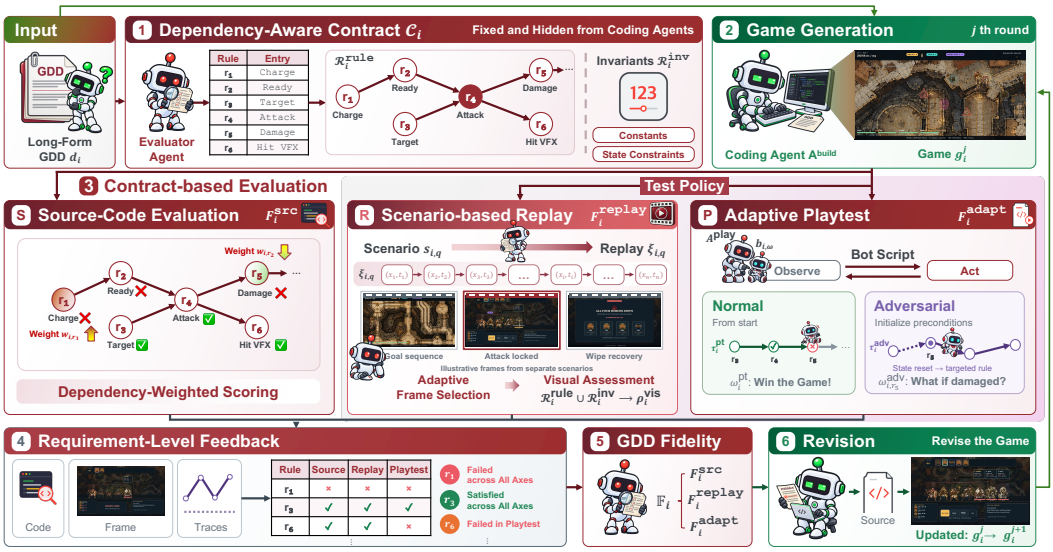


Figure 3: **Overall evaluation and revision pipeline of A2Z GameSpec-Bench.**

time states, and tests them through bounded interaction. GameXpert-Bench (Chen et al., 2026b) combines code inspection and live validation in its GameGen track using an event rubric derived from cross-model outputs. A2Z GameSpec-Bench instead fixes a dependency-aware contract from each long-form GDD before inspecting generated builds. The contract guides source-code inspection and agent-generated test policies for scenario-based replay and adaptive playtesting. Judgments and evidence remain linked to the same requirements across agents and revision rounds for consistent assessment of specification faithfulness and targeted revision.

3 A2Z GameSpec-Bench

Our benchmark measures how faithfully coding agents implement a long-form GDD as a complete game. Rather than considering the GDD as a single free-form rubric, we first convert the GDD into a hidden Dependency-Aware Contract that defines which requirements and dependencies must be verified. Then, our Test Policies specify how to exercise the game and collect evidence for them. Together, source-code inspection, scenario-based replay, and adaptive playtesting provide complementary judgments linked to the same requirements for *GDD Fidelity* scoring and revision feedback, as illustrated in Figure 3. We use a 2D single-player Phaser setting for controlled comparison, while the evaluation framework can be extended beyond it, as discussed in Appendix I.

3.1 Benchmark Setting

Let $\mathcal{D} = \{d_i\}_{i=1}^{N}$ be a corpus of N GDDs, with $\mathcal{U}_i$ denoting the requirements stated in d_i. These include conditional game rules, constraints on states and values, and expected visual responses. Given d_i and a development environment Ω, a game-building agent A^{build} produces

$$g_i^0 = A^{\text{build}}(d_i; \Omega), \tag{1}$$

where g_i^0 contains the source project and executable build. The agent receives the GDD d_i, while its implementation is assessed against a hidden evaluation contract $\mathcal{C}_i$ constructed from that document. We write g_i when evaluating a single build and introduce revision rounds in Section 3.4.

Game Specifications and GDD Generation Protocol. We construct 100 GDDs in two stages: each game brief is first expanded into a creative vision (CV) defining the intended design, and then into a detailed GDD. Then, an agentic authoring pipeline checks each GDD for completeness and consistency. We split the GDDs by implementation scope: 50 *Small* designs with compact scope and 50 *Big* designs with broader content and interacting systems. *Small* GDDs average 14,085 tokens, while *Big* GDDs average 26,297 tokens. Further details of the generation procedure, validation, dataset statistics, and representative examples are in Appendix A.

3.2 DEPENDENCY-AWARE CONTRACT

We construct an evaluation contract from the requirements $\mathcal{U}_i$ in the GDD d_i:

$$\mathcal{C}_i = (\mathcal{R}_i, \mathcal{E}_i), \qquad \mathcal{R}_i = \mathcal{R}_i^{\text{rule}} \cup \mathcal{R}_i^{\text{inv}}, \tag{2}$$

where rules specify conditions, triggering events, and expected effects, while invariants specify constraints within a stated scope. The directed edges $\mathcal{E}_i$ connect rules when one updates a state referenced by another rule's preconditions or emits an event that triggers it. For example, in our game `traces_left`, placing sensors changes the sensor count required by the 'Run' rule, linking placement to subsequent execution. Detailed contract representations, extraction procedures, and a stability of the contract are provided in Appendix B.

3.3 CONTRACT-GUIDED EVALUATION

On average, each GDD specifies 69 outcome requirements and 32 invariants, and evaluation uses 20 replay scenarios. The games do not share the same internal project structure across agents and many requirements in the $\mathcal{C}_i$ cover rendered or interactive outcomes. Thus, we follow game-development verification practices and evaluate the game g_i through source-code inspection, scenario-based replay, and adaptive playtesting. These complementary axes check the same requirements across implementation, rendered output, and actual play. For the execution-based evaluations, TEST POLICY specifies how to exercise the game to produce evidence for these requirements. In short, the contract defines *what* to verify, while TEST POLICY determines *how* to collect the evidence needed for verification. All judgments remain linked to the same requirements for scoring and feedback.

3.3.1 SOURCE-CODE EVALUATION

First, an evaluator agent inspects the source of g_i against $\mathcal{C}_i$ without executing the game. For each rule, it checks the implementation of the specified conditions, triggers, and effects, assigning a completeness score $f_{i,r}^{\text{src}} \in [0,1]$ with source-code references. Invariants use $f_{i,r}^{\text{src}} \in \{0,1\}$ to record whether their constraints are preserved within the stated scope.

Dependency-Weighted Scoring. Because a failure in one rule can affect requirements that depend on its outputs, we account for each rule's downstream reach when aggregating source-code judgments. For scoring, a dependency link is formed when one rule updates a state that is subsequently evaluated in another rule's condition. Let $D_{i,r}$ count the downstream rules reachable from r via these links, and let $D_i^{\max} = \max_{r \in \mathcal{R}_i^{\text{rule}}} D_{i,r}$. For $D_i^{\max} > 0$, the weight $w_{i,r}$ is defined as:

$$w_{i,r} = \alpha + (1-\alpha)\frac{\log(1+D_{i,r})}{\log(1+D_i^{\max})}. \tag{3}$$

We use $\alpha = 0.5$ and set all weights to 1 when $D_i^{\max} = 0$. Then, the source score combines the weighted rule score with the invariant pass rate:

$$F_i^{\text{src}} = F_i^{\text{rule}} F_i^{\text{inv}}, \quad F_i^{\text{rule}} = \frac{\sum_{r \in \mathcal{R}_i^{\text{rule}}} w_{i,r} f_{i,r}^{\text{src}}}{\sum_{r \in \mathcal{R}_i^{\text{rule}}} w_{i,r}}, \quad F_i^{\text{inv}} = \frac{\sum_{r \in \mathcal{R}_i^{\text{inv}}} f_{i,r}^{\text{src}}}{|\mathcal{R}_i^{\text{inv}}|}. \tag{4}$$

Thus, rules with greater downstream reach receive greater influence on the source score, while invariant violations reduce it. Appendix F.1.2 analyzes this weighting.

3.3.2 SCENARIO-BASED REPLAY ASSESSMENT

Source inspection cannot determine whether the specified visual responses actually appear during execution. Therefore, we use controlled scenario replays to collect rendered evidence for the contract requirements. We derive a canonical scenario set $\mathcal{S}_i = \{s_{i,q}\}_{q=1}^{Q_i}$ from $\mathcal{C}_i$, where each scenario groups requirements with a shared entry condition. After finalizing the build, A^{build} provides a fixed replay TEST POLICY with sequence of player inputs and their timing for each scenario $s_{i,q}$:

$$\xi_{i,q} = [(x_n, t_n)]_{n=1}^{N_{i,q}}, \tag{5}$$

Table 2: **Main results on A2Z GameSpec-Bench.** Overall GDD Fidelity, per-axis scores, and verifiable rates on 100 GDDs. **Bold** represents the best result, and underline indicates the second-best.

Model	Overall GDD Fidelity			Evaluation Axis						Verifiable Rate	
				Source		Replay		Playtest			
	All	Small	Big	Small	Big	Small	Big	Small	Big	Small	Big
Claude-Fable-5.1	**77.0**	**82.8**	**71.1**	**89.7**	**83.5**	**73.5**	**55.1**	**85.2**	**74.7**	98.7%	98.7%
Claude-Opus-5	73.9	80.9	66.8	87.9	78.4	71.2	54.4	83.6	67.7	**100.0%**	99.3%
Claude-Opus-4.8	56.8	67.9	45.6	72.5	49.6	62.5	41.4	68.7	46.0	99.3%	**100.0%**
GPT-6-Astra	71.6	78.5	64.6	84.2	74.4	69.9	52.1	81.6	67.4	99.3%	97.3%
GPT-5.6-Sol	61.8	72.8	50.8	78.5	46.3	63.8	48.3	76.0	57.9	98.7%	98.0%
GPT-5.5	63.2	73.1	53.4	81.0	55.9	60.8	42.5	77.5	61.7	97.3%	98.7%
GLM-5.3	55.6	66.9	44.3	78.6	58.1	57.3	34.5	64.9	40.4	88.7%	87.3%
DeepSeek-V4-Pro	50.4	65.3	35.6	75.7	53.8	52.2	25.9	68.0	27.0	88.0%	82.7%
Kimi-K2.7	39.5	51.4	27.5	60.8	37.3	47.7	19.0	45.7	26.2	75.3%	86.0%

where x_n is the input issued at time t_n. The policy is fixed before execution and does not change with runtime observations. We execute $\xi_{i,q}$ under $s_{i,q}$ and record the full replay with input events.

Following the rubric format and evaluation protocol of GameCraft-Bench (Luo et al., 2026), we express replay-observable rules and invariants as a game-level visual rubric ρ_i^{vis}, preserving their conditions and expected visible outcomes. Each item remains linked to its corresponding requirements. Every canonical scenario is replayed and scored against the fixed rubric using a build-specific input policy, and each judgment must cite observed frame evidence.

Adaptive Frame Selection. Fixed-rate sampling can miss brief responses, while a short temporal window can omit delayed outcomes. Therefore, we keep the full scenario replay and let the judge retrieve evidence relevant to ρ_i^{vis}. Given $s_{i,q}$, ρ_i^{vis}, and $\xi_{i,q}$, the judge identifies relevant trace events and requests their temporal intervals. Then, a deterministic tool maps these requests to recorded timestamps and returns up to M frames. Using these frames, the multimodal judge scores each rubric item in $[0, 1]$ with supporting visual evidence. We combine the item rewards across scenarios using the category weights and aggregation rules of GameCraft-Bench to obtain F_i^{replay}.

3.3.3 Adaptive Playtest

A fixed replay provides controlled visual evidence, but it cannot adjust its inputs to the runtime state of the game. Unlike the fixed replay policy, adaptive playtesting requires the Test Policy to respond to the runtime observations. Therefore, we implement it as an executable program following code-as-policy (Liang et al., 2023). An evaluator-side coding agent A^{play} reads a test objective ω and a shared interface API for observing game states and returning valid player inputs:

$$b_{i,\omega} = A^{\text{play}}(g_i, \omega, \text{API}). \tag{6}$$

The bot uses conditions and loops to select its next input from runtime observations. Executing it from an initial state z produces a trace $\tau_i(\omega, z)$ as below:

$$\tau_i(\omega, z) = [(o_t, x_t)]_{t=1}^{T_i}, \tag{7}$$

where o_t is the observation at step t and x_t is the selected player input. The trace records the states, inputs, events, and timestamps used for requirement-level judgment.

Normal Playtest. First, the objective ω_i^{pt} asks the bot to complete g_i from its default state z_i^0:

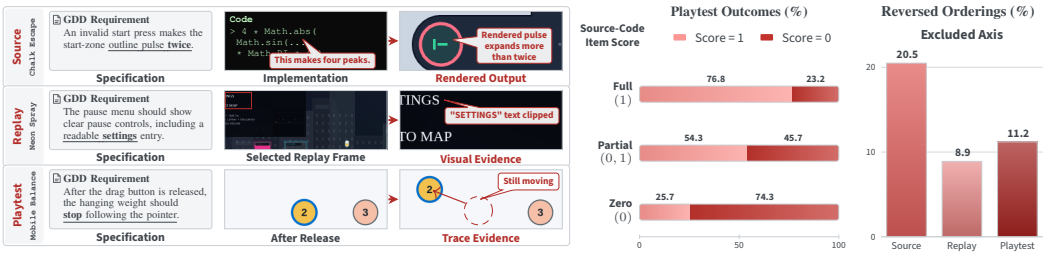
$$\tau_i^{\text{pt}} = \tau_i(\omega_i^{\text{pt}}, z_i^0). \tag{8}$$

Only player inputs are allowed, with no intermediate state initialization. This setting tests which contract requirements can be reached and verified through ordinary gameplay. Let $\mathcal{R}_i^{\text{pt}} \subseteq \mathcal{R}_i^{\text{rule}}$ denote the verified rules with conclusive evidence from this trace.

Adversarial Playtest. Normal play may leave requirements unverified because their preconditions are not reached. Therefore, we apply targeted adversarial tests to $\mathcal{R}_i^{\text{adv}} \subseteq \mathcal{R}_i^{\text{rule}} \setminus \mathcal{R}_i^{\text{pt}}$ for checking edge cases that were not reached during normal play. For each rule r, we initialize the game to a state $z_{i,r}$ satisfying its preconditions and give the bot an objective $\omega_{i,r}^{\text{adv}}$ for testing its expected effects:

$$z_{i,r} \models \phi_r, \qquad \tau_{i,r}^{\text{adv}} = \tau_i(\omega_{i,r}^{\text{adv}}, z_{i,r}). \tag{9}$$

After initialization, the bot uses the same interface as the normal play without further state injection.



(a) Evidence across Axes **(b)** Playtest Confirmation Gap **(c)** Axis Omission

Figure 4: **Complementarity of source-code, replay, and adaptive-play evaluation.** (a) Examples of specification violations identified through different evidence traces. (b) Playtest judgment outcomes grouped by source-code score. (c) Ratio of reversed GDD fidelity rank orderings across all 100 evaluated games built by GPT-5.6-Sol upon omission of individual evaluation axes

Judgment and Scoring. A separate evaluator compares each rule's conditions, trigger, and expected effects with the recorded trace, assigning *satisfied*, *violated*, or *unverified* when evidence is insufficient. Three judgments determine the verdict by strict majority. Conclusive normal-play verdicts are retained, and adversarial tests address only unverified rules. Initialization supplies preconditions but never target outcomes. The adaptive-play score F_i^{adapt} is

$$F_i^{\text{adapt}} = \frac{\sum_{r \in \mathcal{R}_i^{\text{rule}}} f_{i,r}^{\text{adapt}}}{|\mathcal{R}_i^{\text{rule}}|}, \tag{10}$$

where $f_{i,r}^{\text{adapt}} = 1$ only for confirmed satisfaction and 0 otherwise. Each rule contributes once. This score measures confirmed requirement satisfaction under the test protocol; satisfaction after initialization does not establish reachability from the default state. Appendix D.3 details the judgment and aggregation procedures.

3.4 Requirement-Level Evidence And Feedback

For revision round j including the initial build at $j = 0$, we calculate overall *GDD Fidelity* as below:

$$\mathbf{F_i^j} = \frac{F_i^{\text{src},j} + F_i^{\text{replay},j} + F_i^{\text{adapt},j}}{3}. \tag{11}$$

The score $\mathbf{F_i^j}$ supports agent comparison, while the requirement-level judgments and supporting evidence such as code, frames, and traces identify where the implementation differs from the design. These results form feedback h_i^j for the next revision based on g_i^j:

$$g_i^{j+1} = A^{\text{build}}(d_i, g_i^j, h_i^j; \Omega). \tag{12}$$

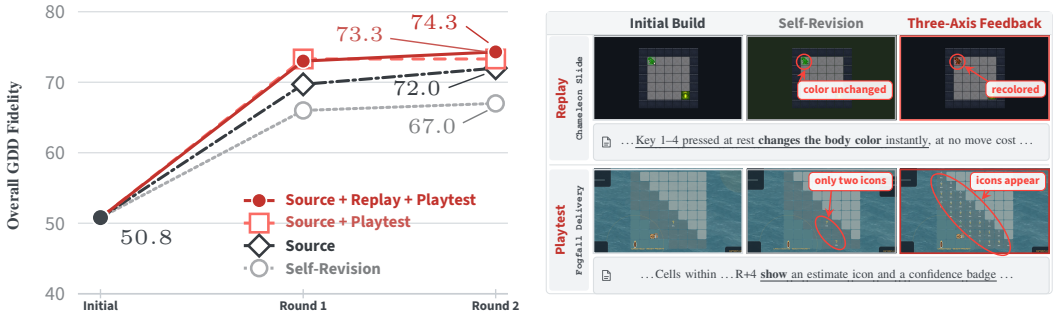
We assess the revised build against the same contract using all three axes, keeping the evaluation target fixed while measuring changes in specification faithfulness.

4 Experiments

4.1 Experimental Setup

We evaluate coding-agent configurations based on Claude-Fable-5.1 (Anthropic, 2026), Claude-Opus-5 (Anthropic, 2026), Claude-Opus-4.8 (Anthropic, 2026), GPT-6-Astra (OpenAI, 2026c), GPT-5.6-Sol (OpenAI, 2026b), GPT-5.5 (OpenAI, 2026a), KIMI-K2.7 (Kimi Team, 2025), GLM-5.3 (GLM-5-Team, 2026), and DeepSeek-V4 (DeepSeek-AI, 2026). Each configuration is evaluated on all 100 GDDs. For each GDD, the same contract and replay scenarios are used across all outputs.

Unless otherwise stated, evaluator-side agents use GPT-5.6-Luna at high reasoning effort. The judge model agent comparisons and cost analyses are provided in Appendix G.2, with additional replay-judge analysis in Appendix F.2. Each configuration generates three builds per GDD, and each build is independently judged three times, with axis-specific aggregation described in Appendix D. We report overall *GDD Fidelity*, the source-code, replay, and adaptive-play scores, and the *verifiable rate* defined as the average compile and runtime pass rate across the three axes. Further details on agent configurations, computational resources, and evaluation settings are provided in Appendix C.



(a) Comparison on GDD Fidelity over iterations **(b)** Qualitative Repairs

Figure 5: **Iterative revision with benchmark feedback.** (a) Overall GDD Fidelity at the initial build and after each revision round. (b) Qualitative comparison between *self revision* and ours.

4.2 Main Benchmark Results

As shown in Table 2, Claude-Fable-5.1 achieves the highest overall GDD Fidelity of 77.0 across all 100 GDDs, followed by Claude-Opus-5 at 73.9. Across the evaluated agents, mean overall GDD Fidelity is 71.1 on *Small* and 51.1 on *Big*, while the corresponding verifiable rates are 93.9% and 94.2%. Although most games can be compiled and executed under our evaluation, high verifiability does not imply faithful implementation of the intended design. In addition, implementing games from *Big* GDDs is consistently more difficult across all three evaluation axes than implementing games from *Small* GDDs. Overall score decreases by 20.0 points from *Small* to *Big*. The average source-code, scenario-based replay, and adaptive playtest scores similarly decrease by 19.1, 20.6, and 20.3 points, which shows that difficulty of broader specifications is not confined to source implementation, visual rendering, or runtime behavior alone. Performance also varies across evaluation axes. On *Small*, GPT-5.5 scores higher than GPT-5.6-Sol in source-code evaluation (81.0 vs. 78.5) and playtest (77.5 vs. 76.0), whereas GPT-5.6-Sol scores higher in replay (63.8 vs. 60.8). Such reversals across axes show that stronger performance under one-axis evidence does not necessarily transfer to the other axis. Further analysis on agent-level comparisons are provided in Appendix E.

4.3 Complementarity of the Evaluation Axes

Figure 4 demonstrates that the three axes capture different failures for the same specification. Figure 4(a) presents violations identified through different evidence channels: an incorrect pulse count in the implementation, clipped menu text in replay frames, and continued object movement after input release during playtesting. As shown in Figure 4(b), even among rules that pass with a full source-code score, only 76.8% are confirmed as satisfied during playtesting, while 23.2% do not receive a full score. In contrast, 25.7% of rules receiving zero source-code score are judged as satisfied during playtesting. These disagreements show that implementation-level judgments and runtime observations provide different evidence on requirement satisfaction. Additionally, Figure 4(c) reports ordering reversals across the 100 games generated by GPT-5.6-Sol when each evaluation axis is omitted. Relative to the full three-axis evaluation, omitting source-code analysis, replay assessment, or adaptive playtest yields reversed-ordering rates of 20.5%, 8.9%, and 11.2%, respectively (Appendix F.4). Additional source-code and runtime discrepancies are analyzed in Appendix F.4.

4.4 Iterative Revision with Three-Axis Feedback

We study whether requirement-level feedback based on our contract improves overall GDD Fidelity beyond revision guided by direct GDD and source code, using 50 *Big* GDDs with GPT-5.6-Sol. All conditions start from the same initial builds, use the same revision agent for two rounds, and are evaluated on all three axes against the same contracts. The *self-revision* baseline gets feedback from an evaluator agent that directly reviews the GDD and the source code without using our contract and feedback. We compare it with three benchmark-feedback conditions: *Source* feedback provides contract-based source-code evaluation, *Source* + *Playtest* adds playtest feedback, and *Source* + *Replay* + *Playtest* corresponds to the three-axis feedback.

As shown in Figure 5, *Source* + *Replay* + *Playtest* reaches a mean overall GDD Fidelity of 74.3 after two rounds, exceeding the 67.0 of *self-revision* by 7.3 points (10.9% relative). Under the same revi-

Table 3: **Component analysis.** (a) Mean and maximum absolute changes in source-code score from uniform weighting (Appendix F.1.2). (b) Evidence support for fixed-rate versus adaptive frame selection (Appendix G.3). (c) Mean playtest score and judgment coverage (%) before and after adversarial testing (Appendix F.3). Coverage includes both *satisfied* and *violated* requirements.

(a) Dependency Weighting

Setting (α)	Score Change Mean $\lvert\Delta\rvert$	Score Change Max. $\lvert\Delta\rvert$
Default (0.5)	0.38	3.71
Stronger (0.1)	0.96	19.10

(b) Frame Selection

Method	Evidence Support Item Preference	Evidence Support Game Wins
Fixed-Rate	39.5	13
Adaptive Selection	**60.5**	**35**

(c) Adaptive Playtesting

Playtest Mode	Runtime Evidence Playtest Score	Runtime Evidence Judgment Coverage
Normal	24.4	24.9
Normal + Adv.	**67.0**	**81.8**

sion stage, the larger gains from our feedback support using requirement-level judgment evidence to guide revisions toward the specified design, rather than relying solely on the GDD and source code.

4.5 Component Analysis

Dependency-Weighted Scoring. We examine whether dependency weighting, as described in Section 3.3.1, accurately reflects where implementation failures occur under the same rule judgments. According to results in Section 4.2, the default $\alpha = 0.5$ setting adjusts source scores by an average of 0.38 points in absolute terms, with a maximum adjustment of 3.71 points, as shown in Table 3(a). Applying stronger weighting at $\alpha = 0.1$ increases these changes to 0.96 and 19.10 points, respectively, while model rankings remain unchanged across both splits. In a complementary source-code and playtest analysis, weighting lowers rule scores in 61.1% of builds with confirmed high-reach failures, compared with 37.5% of comparison builds (Appendix F.1.2). Further analysis shows that applying contract dependencies to the judgments with fixed enumerated requirements selects 408 additional inspection targets across 100 games (Appendix G.1.2).

Adaptive Frame Selection. We compare adaptive frame selection with fixed-rate sampling across all 50 *Small* GDD games generated by GPT-5.6-Sol, utilizing identical recorded replays, visual rubrics, and frame counts per replay. A separate agent evaluates how effectively the cited frames support each requirement-level rationale. As shown in Table 3(b), adaptive selection is preferred in 60.5% of comparisons where either method is favored, and receives more preferences in 35 out of 50 games, compared to 13 for fixed-rate sampling, with 2 ties. These results indicate that selecting frames around relevant gameplay events provides stronger support for replay judgments than fixed-rate sampling. Appendix G.3 presents a detailed analysis of these sampling methods.

Normal and Adversarial Playtests. As shown in Table 3(c), on 100 unchanged games generated by GPT-5.6-Sol, adding adversarial tests increases judgment coverage from 24.9% to 81.8% and the playtest score from 24.4% to 67.0%. Both measures retain the full rule set as the denominator. the increases reflect additional execution evidence on the same implementations. In a separate controlled comparison on 86 dependency-linked rules from 50 games, a second normal pass increases coverage from 40.5% to 41.0%, whereas an adversarial second pass raises it to 48.5%, with builds, contracts, test policies, interfaces, and budgets held fixed (Appendix F.3). These results show that prerequisite initialization enables additional downstream checks beyond those obtained by repeating normal play.

5 Conclusion

We introduced A2Z GameSpec-Bench, a benchmark for evaluating coding agents' faithfulness to 100 long-form GDDs in end-to-end game development. Fixed dependency-aware contracts guide both source-code inspection and agent-generated test policies for replay and adaptive playtesting, linking evidence to the same requirements. The results show that runnable outputs may not fully correspond to the intended design and that local source-code passes can coexist with failed prerequisites. On all *Big* GDDs with GPT-5.6-Sol, three-axis feedback yields a 10.9% relative improvement in overall GDD Fidelity compared to *self-revision* after two rounds. By connecting requirement-level evaluation with targeted revision, A2Z GameSpec-Bench enables both assessment and improvement of specification faithfulness in game development.

AI USE STATEMENT

In this work, we used generative AI for synthetic GDD construction and validation, authoring-harness development, evaluation-contract formalization, game generation and revision, test-policy generation, and requirement-level evidence analysis and feedback. These uses are part of the research methodology described in the paper. We additionally used generative AI to assist with interpreting experimental results, drafting and revising the manuscript, LaTeX formatting, and creating illustrative figures. Quality-control procedures included specification validation and repair, programmatic contract checks, evidence-linked judgments, and repeated assessment, with expert calibration limited to the scenario-based replay assessment and adaptive playtest axes. We take responsibility for the final content of this work, including AI-assisted text, claims, code, and figures.

ETHICS STATEMENT

A2Z GameSpec-Bench evaluates coding agents' faithfulness to game design specifications. Human calibration is conducted by co-authors including two technical artists and three game developers, who assess recorded gameplay and interacted with generated games using the protocol in Appendix H. Model-generated specifications and model-based judgments may contain errors or reflect model biases. Fixed contracts, requirement-level evidence, repeated assessments, and expert calibration support reliability but do not guarantee correct or unbiased judgments. GDD Fidelity measures specification conformance under the reported protocol and should not be interpreted as a measure of general game quality, software security, or deployment readiness.

REPRODUCIBILITY STATEMENT

We describe the benchmark construction and validation process, dependency-aware contract generation, experimental environments, model configurations, and evaluation and scoring procedures in the paper and appendix. Appendix A describes GDD construction and validation, and Appendix C reports the execution environments, model configurations, and computational resources. Contract construction and evaluation protocols are detailed in Appendices B and D. Contract stability, evaluator comparisons, and human calibration are reported in Appendices B.5, G.2, and H, respectively.

We provide the GDDs, evaluation contracts, and evaluation code in the supplementary materials. Evaluation contracts remain fixed across coding agents and revision rounds. Reproduction depends on access to the specified hosted models and execution environments; stochastic generation and changes to model services may prevent exact regeneration of individual outputs.

REFERENCES

Anthropic. Claude Code overview. Official Documentation, 2026. URL https://code.claude.com/docs/en/overview. Accessed: 2026-08-29.

David Callele, Eric Neufeld, and Kevin Schneider. Requirements engineering and the creative process in the video game industry. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pp. 240–250. IEEE, 2005.

Boshui Chen, Huiping Liu, and Shaolei Zhang. Webgrader: Training llms for web development with self-evolving programmatic grader. *arXiv preprint arXiv:2608.06474*, 2026a.

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?, 2026b. URL https://arxiv.org/abs/2608.21833.

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.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*, 2024.

Jaemin Cho, Abhay Zala, and Mohit Bansal. Visual programming for step-by-step text-to-image generation and evaluation. *Advances in Neural Information Processing Systems*, 36:6048–6069, 2023.

Jaemin Cho, Yushi Hu, Jason Baldridge, Roopal Garg, Peter Anderson, Ranjay Krishna, Mohit Bansal, Jordi Pont-Tuset, and Su Wang. Davidsonian scene graph: Improving reliability in fine-grained evaluation for text-to-image generation. In *International conference on learning representations*, volume 2024, pp. 15625–15645, 2024.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Xiaoyu Dong, Zhi Li, and Xiao-Ming Wu. Muse: Benchmarking manufacturable, functional, and assemblable text-to-cad generation. *arXiv preprint arXiv:2605.28579*, 2026.

Epic Games. *Automation Test Framework*. Epic Games, n.d. URL https://dev.epicgames.com/documentation/unreal-engine/automation-test-framework-in-unreal-engine. Unreal Engine documentation. Accessed September 23, 2026.

Jannik Fischbach, Julian Frattini, Arjen Spaans, Maximilian Kummeth, Andreas Vogelsang, Daniel Mendez, and Michael Unterkalmsteiner. Automatic detection of causality in requirement artifacts: the CiRA approach. In *Requirements Engineering: Foundation for Software Quality (REFSQ)*, 2021. arXiv:2101.10766.

Julian Frattini, Jannik Fischbach, Daniel Mendez, Michael Unterkalmsteiner, Andreas Vogelsang, and Krzysztof Wnuk. Causality in requirements artifacts: prevalence, detection, and impact. *Requirements Engineering*, 28(1):49–74, 2023.

GLM-5-Team. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Keith R. Hamilton. Race'n'Chase Game Design. Game Design Document Version 1.05, DMA Design Ltd., March 1995. URL https://www.gamedevs.org/uploads/grand-theft-auto.pdf. March 22, 1995.

Yushi Hu, Benlin Liu, Jungo Kasai, Yizhong Wang, Mari Ostendorf, Ranjay Krishna, and Noah A Smith. Tifa: Accurate and interpretable text-to-image faithfulness evaluation with question answering. In *2023 ieee/cvf international conference on computer vision (iccv)*, pp. 20349–20360. IEEE, 2023.

Yixu Huang, Bo Li, Na Li, Zhe Wang, Kaijie Chen, Haonan Ge, Qingyi Si, Yuanzhe Shen, Ruihan Yang, Guangjing Wang, and Hongcheng Guo. 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.

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*, 2026.

Kimi Team. Kimi K2: Open agentic intelligence. arXiv preprint arXiv:2507.20534, 2025. URL https://arxiv.org/abs/2507.20534v1.

Guy Kisel. Running an Automated Test Pipeline for the League Client Update. Riot Games Tech Blog, October 2016. URL https://www.riotgames.com/en/news/running-automated-test-pipeline-league-client-update. Published October 25, 2016.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

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.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Mingxian Lin, Shengju Qian, Yuqi Liu, Yi-Hua Huang, Yiyu Wang, Wei Huang, Yitang Li, Fan Zhang, Zeyu Hu, Lingting Zhu, et al. Omnigamearena: A unified ue5 benchmark for vlm game agents with improvement dynamics. *arXiv preprint arXiv:2606.09826*, 2026.

Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, Fan Bu, Wenyu Du, Xiaotong Zhang, Kyle Li, Shaobo Wang, Linfeng Zhang, Yuxuan Liu, Xin Lai, Chenxin Li, Yiduo Guo, Zhexin Zhang, Xinyuan Wang, Tianyi Bai, Ziniu Li, and Benyou Wang. GameCraft-Bench: Can agents build playable games end-to-end in a real game engine? *arXiv preprint arXiv:2606.17861*, 2026.

Yuxin Meng, Yuhan Suo, Junjie Wang, Yuhan Sun, Yiyao Yu, Ruixu Zhang, Ruining Hu, Yubin Wang, Shouwei Ruan, Bin Wang, et al. Webrise: Requirement-induced state evaluation for mllm-generated web artifacts. *arXiv preprint arXiv:2606.03220*, 2026.

OpenAI. Gpt-5.5 system card, April 2026a. URL https://openai.com/index/gpt-5-5-system-card/. Published by OpenAI Deployment Safety Hub.

OpenAI. Gpt-5.6 system card, July 2026b. URL https://deploymentsafety.openai.com/gpt-5-6/gpt-5-6.pdf. Published by OpenAI Deployment Safety Hub.

OpenAI. Gpt-6 astra: A new generation of intelligence, 2026c. URL https://openai.com/index/gpt-6-astra/. Accessed: 2026-09-16.

Mingyu Ouyang, Siyuan Hu, Kevin Qinghong Lin, Hwee Tou Ng, and Mike Zheng Shou. Gameworld: Towards standardized and verifiable evaluation of multimodal game agents. *arXiv preprint arXiv:2604.07429*, 2026.

Dongmin Park, Minkyu Kim, Beongjun Choi, Junhyuck Kim, Keon Lee, Jonghyun Lee, Inkyu Park, ByeongUk Lee, Jaeyoung Hwang, Jaewoo Ahn, et al. Orak: A foundational benchmark for training and evaluating llm agents on diverse video games. In *International Conference on Learning Representations*, volume 2026, pp. 60684–60744, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 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. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Mario Gonzalez Salazar, Hugo A Mitre, Cuauhtémoc Lemus Olalde, and José Luis González Sánchez. Proposal of game design document from software engineering requirements perspective. In *2012 17th International Conference on Computer Games (CGAMES)*, pp. 81–85. IEEE, 2012.

Bosi Wen, Pei Ke, Xiaotao Gu, Lindong Wu, Hao Huang, Jinfeng Zhou, Wenchuang Li, Binxin Hu, Wendy Gao, Jiaxin Xu, Yiming Liu, Jie Tang, Hongning Wang, and Minlie Huang. Benchmarking complex instruction-following with multiple constraints composition. *Advances in Neural Information Processing Systems*, 37, 2024. doi: 10.52202/079017-4371. URL https://papers.nips.cc/paper_files/paper/2024/hash/f8c24b08b96a08ec7a7a975feea7777e-Abstract-Datasets_and_Benchmarks_Track.html.

Xu Xu, Xin Li, Xingwei Qu, Jie Fu, and Binhang Yuan. Local success does not compose: Benchmarking large language models for compositional formal verification. In *International Conference on Learning Representations*, volume 2026, pp. 118471–118508, 2026.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026.

Wenyu Zhang, Guoliang You, Tianlun, Haotian Zhao, Tianshu Zhu, Haoran Wang, Xiaoxuan Tang, Mingyang Dai, Jingnan Gu, Daxiang Dong, and Jianmin Wu. WebGameBench: Requirement-to-application evaluation for coding agents via browser-native games. *arXiv preprint arXiv:2605.17637*, 2026. URL https://arxiv.org/abs/2605.17637.

APPENDIX CONTENTS

A Game Design Document Dataset . . . **2**
A.1 Brief-to-GDD Pipeline . . . 2
A.2 Harness Optimization . . . 2
A.3 Admission Criteria . . . 3
A.4 Validation and Repair . . . 4
A.5 Dataset Splits and Statistics . . . 5
A.6 Examples of GDDs . . . 7

B Details on Contract Generation . . . **10**
B.1 Dependency Structure in Game Specifications . . . 10
B.2 Process of Generating Rule . . . 10
B.3 Guarantees of the Contract . . . 12
B.4 Example of Contract . . . 12
B.5 Stability of Contract Generation . . . 12
B.6 Value Compatibility . . . 15
B.7 Dependency-Adjusted Rule Pass Rate . . . 15

C Experimental Details . . . **16**

D Details on Evaluation Setup . . . **17**
D.1 Source Code Evaluation . . . 17
D.2 Scenario-Based Replay Assessment . . . 18
D.3 Adaptive Playtest . . . 20

E Additional Analysis on Benchmark Results . . . **23**

F Ablations on Evaluation . . . **27**
F.1 Source-Code Evaluation . . . 27
F.2 Scenario-Based Replay Assessment . . . 29
F.3 Adaptive Playtest . . . 30
F.4 Complementarity of Evaluation . . . 30

G Additional Analysis on Evaluation . . . **33**
G.1 Dependency-Aware Contract . . . 33
G.2 Cost Analysis . . . 35
G.3 Fixed-Rate Sampling vs. Adaptive Frame Selection . . . 37
G.4 Gameplay Agent vs. Adaptive Playtest . . . 40

H Human Calibration . . . **41**

I Scalability of A2Z GameSpec-Bench . . . **43**

J More Visual Results . . . **45**

K Limitations . . . **51**

APPENDIX

A GAME DESIGN DOCUMENT DATASET

A2Z GameSpec-Bench comprises 100 long-form game design documents (GDDs), divided into *Small* and *Big* splits of 50 each. Because the benchmark measures how faithfully a game implements its GDD, the documents define the evaluation targets, and their quality affects the validity of the resulting scores. Sections A.3 and A.4 define the admission criteria and the validation-and-repair procedure. Section A.5 describes the splits and dataset statistics, and Section A.6 illustrates the document structure with one example from each split. Separately, Section B.1 examines causal descriptions in external game-design corpora, motivating the dependency-aware representation introduced in Section 3.2.

A.1 BRIEF-TO-GDD PIPELINE

Each GDD is derived from a short game brief called creative vision (CV) in two stages, as shown in Figure S1. The brief fixes the genre, a one-line premise, the core mechanic, reference games, and the intended scope, under constraints shared by every game: PC, 2D, single player, and keyboard-and-mouse input. A CV generator first expands the brief into a CV with a fixed set of sections: core experience, opening fiction, hook, one pass of the core loop, mechanic intent, and explicit non-goals, plus sections chosen from the game's genre tags. The CV establishes the intended experience and scope. The harness described in Section A.2 then turns the CV into a GDD that specifies the entities, rules, formulas, stages, screens, and assets required for implementation. This separation establishes the design intent before its detailed specification and provides a reference for evaluating the GDD's fidelity to that intent.

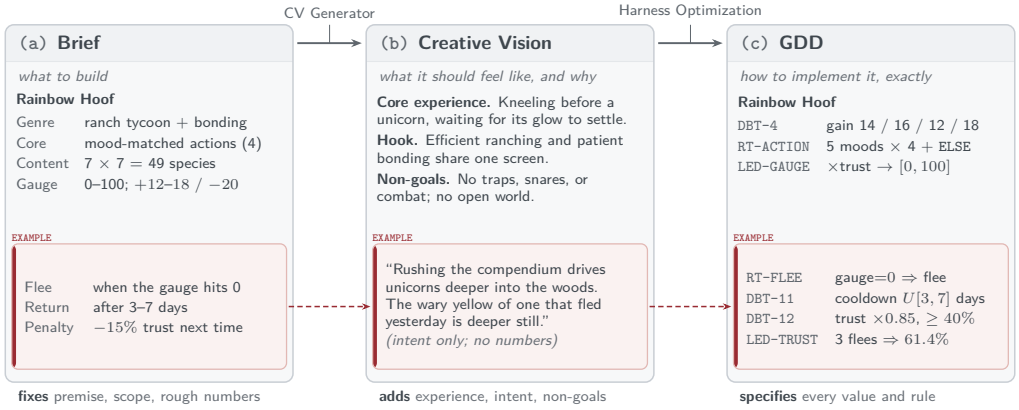


Figure S1: One game traced from brief to GDD. The brief fixes what to build, the CV adds how the game should feel and why (without numbers), and the GDD specifies every value and rule. The highlighted boxes follow one requirement, the cost of letting a unicorn flee, through all three stages: a rough rule in the brief, a design intent in the CV, and a cited rule, constants, and a worked example in the GDD.

A.2 HARNESS OPTIMIZATION

An agent performs the CV-to-GDD step inside a harness: the generator's instructions, the project files it can consult, and its authoring-and-review workflow. Rather than tuning the harness by hand, we refine it with an iterative search inspired by Meta-Harness (Lee et al., 2026).

Search procedure. The search uses a fixed set of N CVs. Starting from a seed harness, each iteration has a proposer agent read the search history, which contains earlier harnesses, their GDDs, the evaluator scores, and the evaluators' feedback. The proposer then produces K candidate harnesses

that modify the instructions, the project files, or the workflow. Each candidate generates a GDD from every CV, and the GDDs are scored as described below. The CVs, the fidelity checklists derived from them, and the evaluator configuration stay fixed across candidates to support a controlled comparison of the harnesses.

Document-quality scores. *CV fidelity* is the fraction of items in a fixed, CV-derived checklist that the GDD satisfies. The checklist covers the CV's mechanics, constraints, design intentions, and any stated quantitative details. The full checklist is always the denominator, so an item the judge omits counts as unsatisfied. *Buildability* is computed over ten disciplines: gameplay, systems, economy, AI, UX, level design, art, audio, narrative, and production. For each applicable discipline, the judge lists the required deliverables and computes the fraction that are specified concretely and consistently. A mechanic or asset that is only named does not count. The score is the unweighted mean over applicable disciplines. The two scores check each other. Fidelity alone would reward restating the CV, and buildability alone would reward detailed specifications that drift from it. With $F_j(h)$ and $B_j(h)$ the scores of harness h on the j-th CV,

$$R(h) = \frac{1}{N} \sum_{j=1}^{N} \big(F_j(h) + B_j(h)\big), \tag{13}$$

which weights both dimensions and all CVs equally. A candidate is scored only if both dimensions are usable for every CV. These scores assess the document and are separate from GDD Fidelity, which assesses the game.

Harness selection. Rewards guide the search, but a candidate replaces the current best harness only if it wins a head-to-head comparison. For each CV, fresh judge sessions compare the two GDDs on every fidelity-checklist item and every applicable buildability discipline. Each item-level comparison is repeated three times independently, and the item-level winner is determined by majority vote; a tie is recorded as undecided. For each dimension, we then determine the CV-level winner by majority vote over the decided item-level outcomes. A tie at this stage is also recorded as undecided. Thus, each CV casts at most one vote per dimension. A candidate must win a strict majority of the decided CV-level votes. If multiple candidates meet this criterion, we select the one with the highest win rate among decided votes.

Configuration. Table S1 lists the search settings and selected checkpoint. All 100 benchmark GDDs are generated with the harness selected at iteration 8. Harness search improves the typical GDD but does not guarantee that any individual GDD is defect-free, so every GDD must still meet the criteria in Section A.3.

Setting	Value
CVs used in the search (N)	13
Candidate harnesses per iteration (K)	3
Repeats per quality judgment / pairwise comparison	3 / 3
Proposer and generator	Claude-Opus-4.8
Evaluator (scoring and pairwise judging)	GPT-5.5
Maximum iterations	20
Selected checkpoint	iteration 8

Table S1: Harness search configuration.

A.3 Admission Criteria

A conformance score such as GDD Fidelity is meaningful only when the GDD specifies requirements clearly enough to evaluate and consistently enough to admit a coherent implementation. Missing or undefined requirements cannot be checked reliably, while contradictory requirements can make full conformance impossible: any implementation may violate at least one. These specification defects can therefore make low scores difficult to attribute to the agent's implementation. A

rule that can never take effect penalizes faithful implementations. A formula open to more than one reading can make the agent and the evaluator compute different correct answers.

We call a document a *Golden GDD* after it passes the validation-and-repair procedure in Section A.4 with no detected violations of the four criteria in Table S2. These criteria target the specification defects described above. G1 and G4 apply to individual statements. G2 and G3 apply to the document as a whole. G2 is static: the text agrees with itself. G3 is dynamic: the rules still hold together when the game runs. A document can pass G2 and still fail G3. For example, two consistent statements can together leave a mechanic that is impossible to reach.

Making the criteria checkable. GDDs use three conventions that support systematic checks of G2–G4. (i) Every tunable value has a canonical definition with a stable identifier (`DBT-`n) in a *design balance table*. Other sections refer to that identifier. Any repeated value must agree with its canonical definition. This makes references and repeated values traceable during G2 checks. (ii) Discrete behavior is written as a *rule table* (`RT-`$\star$). Its rows are evaluated top to bottom. The first match applies, and a final `ELSE` row covers every remaining case, which makes the totality and determinism required by G3 visible. (iii) Any value computed from several sources is written as a *ledger* (`LED-`$\star$). A ledger fixes the evaluation order and the bounds and gives a worked numeric example that a G4 check can recompute. All 100 GDDs use balance tables, 83 use rule tables, and 35 use ledgers. Most ledgers appear in the Big split, where values combine across systems.

Table S2: **Criteria for golden GDDs.** Each criterion excludes one way a GDD can invalidate conformance-based evaluation. Example violations are taken from our validation logs.

Criterion	Requirement	Failure if violated	Example violation
G1 Structural Conformance	• Standalone, all sections present • No empty or `[TBD]` section • No dangling identifier	*The requirement cannot be checked.*	A section cited elsewhere had been removed.
G2 Referential Consistency	• One canonical definition (`DBT-`n) • Every citation resolves • Repeated values agree	*Contradictions; no implementation can satisfy all.*	`bool[56]` declared for 55 species; 1 wave in App. A vs. 3 in App. B.
G3 Behavioral Soundness	• Total, deterministic rule tables • No unreachable or dead mechanic • Difficulty never decreases	*Faithful implementations are penalized.*	A 90 s boss timer never fires; HP decay kills the patient at 40 s.
G4 Executable Formulas	• Variables bound to constants • Fixed evaluation order, bounds • Worked example that recomputes	*Agent and evaluator can compute different answers.*	Kill time ignores the boss's DEF; `ceil(50/4)` written as 12.

These criteria concern the GDD's validity as a specification. A Golden GDD may describe a simple or an unambitious game. The CV fidelity and buildability scores used during harness search (Section A.2) assess alignment with the CV and implementation readiness separately.

A.4 Validation and Repair

Candidate generation. The briefs span a diverse range of genres, core-mechanic families, and production scopes. Each brief expands into a CV and then into a GDD using the harness detailed in Section A.2.

Validation. Validate-Agents check each candidate against G1–G4, with candidates distributed across validators running in parallel. In the initial validation, the system checks every applicable table, formula, and cross-reference. It does not sample within a document. In subsequent rounds, doc-

uments changed during repair are fully re-validated, while unchanged documents are spot-checked for likely regressions. The checks are applied wherever the GDD contains the relevant structure:

- **G1**: no `[TBD]` or placeholder remains; every cited section and identifier exists.
- **G2**: recurring values agree across entity tables, systems, the stage catalog, and level descriptions. Different descriptions of the same stage agree on wave count, objectives, enemy composition, and learning goal.
- **G3**: difficulty arrays are non-decreasing; the first appearance declared for each entity or environment object corresponds to the stage catalog; timers and triggers are reachable given the interacting systems.
- **G4**: boundary tables are recomputed from their formulas and arrays; derived quantities such as $\lceil HP/dmg \rceil \times t_{\text{loop}}$ are re-derived; every exception to a formula is listed.

Repair and acceptance. Each violation is fixed at its canonical definition. When two sections disagree, the more specific one is kept, and the other is changed to match it, and every fix is logged with its criterion. Documents changed in a round are fully re-validated in the next round. Documents that did not change are spot-checked on the properties most likely to regress. A document is accepted only after three consecutive rounds with no detected violations. As an illustration, in one batch of 13 candidates, the logged rounds record 24 fixes, after which the batch passes three clean rounds in a row. Most fixes are numeric values that disagree with the document's own worked examples, mismatches between the stage catalog and the level descriptions, and first-appearance columns that do not match the catalog.

Implications for evaluation. The accepted documents form the 100 Golden GDDs in the benchmark. Passing G1–G4 reduces the risk that low conformance scores arise from detectable specification defects, although evaluator and implementation-interpretation errors can remain. Stable identifiers for constants, rules, and formulas provide fixed anchors for checking individual requirements and locating departures from the specification.

A.5 Dataset Splits and Statistics

Accepted GDDs are organized into *Small* and *Big* splits of 50 each according to the scope of the specified game: the mechanics, content, and interacting systems to be implemented, rather than the length or quality of the document. Document lengths and extracted requirement counts are reported as dataset statistics and are not used as thresholds for assigning the splits.

Small GDDs focus on a single core action or a compact set of mechanics, typically within a single screen and a short session. Each specifies clear success and failure conditions, together with input handling, state transitions, feedback, and boundary cases. Campaigns, progression systems, shops, and narrative content outside the core task are excluded. These restrictions limit the scope of the game without reducing specification detail: each GDD remains self-contained and includes the rules, numerical values, inputs, UI text, content, art, audio, and testing requirements needed for implementation. Small GDDs contain approximately 54 outcome requirements on average, with a range of 9–151.

Big GDDs cover broader content and interacting systems, including combinations of combat, progression, economy, exploration, narrative, and management. The split contains complete 2D single-player designs. Specifications that require 3D or 2.5D presentation, mandatory multiplayer, or further adaptation to the target setting are excluded, as are games that duplicate a Small entry. Big GDDs contain approximately 84 outcome requirements on average, with a range of 10–246.

To limit repetition within the *Big* split, the default selection policy admits at most two entries from the same narrow genre and core-mechanic family. Within a broad genre, designs are treated as distinct families only when their primary inputs, repeated actions, success and failure criteria, and progression structure all differ. This allows several games from one broad genre while limiting repeated instances of the same core task.

The document lengths, requirement composition, and category coverage for the two sets are summarised in Table S3. Outcome requirements count individual promises listed in GDD rule-table rows, including presentation outcomes. A rule in the dependency graph specifies a source row's

conditions, trigger, and state-changing or event-emitting outcomes. On average, the Big GDDs contain about $1.87\times$ as many tokens and $1.55\times$ as many outcome requirements as the Small GDDs. The 50 Small GDDs cover eight task families (Table S4), and the 50 Big GDDs cover 11 commercial game genres (Table S5).

Table S3: **GDD dataset statistics.** Document length and requirement composition of the *Small* and *Big* splits. Outcome requirements count individual promises in GDD rule-table rows, including presentation outcomes.

Statistic	***Small***	***Big***
Documents	50	50
Lines (mean)	629	1,153
Tokens (total)	**704,270**	**1,314,862**
Tokens (mean, range)	14,085 (5,240–24,022)	26,297 (12,489–51,433)
Outcome requirements (total)	**2,710**	**4,202**
Outcome requirements (mean, range)	54.2 (9–151)	84.0 (10–246)
Invariants (total)	1,378	1,862
Scenarios (total)	646	1,385

Table S4: **Task-family distribution of the *Small* GDDs.**

#	**Task Family**	n	**Games**
1	Reaction, Inhibition & Speed Classification	9	`reaction_test`, `no_response_test`, `bloom_or_weed`, `orbit`, `bin_bit`, `odd_even_flash`, `odd_patch`, `same_or_shift`, `three_letter_delivery`
2	Temporal Precision & Rhythm	7	`beat_bistro`, `centerline`, `chord_snap`, `half_beat_bell`, `ooga_ooga_tower`, `ozone_dash`, `rooftop_rumble`
3	Continuous & Precision Motor Control	9	`cursor_pursuit`, `still_cursor`, `chalk_escape`, `stamp_register`, `pocket_curling`, `magnet_dock`, `safe_dial`, `warm_ascent`, `wheel_steps`
4	Short-Term Memory & Attentional Tracking	3	`echo_three`, `hidden_spot`, `key_under_cups`
5	Perceptual Estimation & Psychophysics	4	`blink_count`, `equal_slice`, `mobile_balance`, `whisper_compass`
6	Spatial Reasoning & Mental Transformation	5	`one_fold_letter`, `shadow_pair`, `lighthouse_mirrors`, `tiny_moving_van`, `untangled_cords`
7	Deduction & Constraint Solving	6	`clue_lock`, `mini_nonogram`, `three_switches`, `traces_left`, `one_segment_equation`, `last_match`
8	Planning & Sequential Decision	7	`blackout_dispatch`, `crate_corner`, `color_flood`, `peg_garden`, `pancake_prefix`, `tile_merge`, `greedy_die`

Table S5: **Genre distribution of the *Big* GDD split.**

#	Genre	n	Games
1	Platformer & Metroidvania	7	`afterglow`, `chromashade`, `rainwright`, `ssitgim`, `turning_keep`, `neon_spray`, `the_librarians_hook`
2	Action Roguelite & Survival	4	`abyssal_chain`, `seamline`, `todays_best_spot`, `afterglow_network`
3	Shooter & Bullet-hell	3	`golem_workshop`, `magma_arc`, `snowfall_draw`
4	Rhythm	4	`beat_reroute`, `beatstorm`, `cloud_cast`, `rhythm_of_response`
5	Puzzle	4	`chameleon_slide`, `fable_knot`, `fogfall_delivery`, `u001_toyfit`
6	Strategy & Tactics	5	`ashen_citadel`, `gateline_2d`, `siege_deck_2d`, `chroma_bastion`, `xl001_edens_debt`
7	RPG	4	`four_of_the_depths`, `linkbound`, `dh001_echo_gate_guild`, `borrowed_faces_2d`
8	Simulation & Management	8	`anomaly_factory`, `grand_atelier_2d`, `night_shift_rx`, `rainbow_hoof`, `reef_bistro`, `strata_keepers`, `wham_bam_logistics`, `wind_winter_camp`
9	Idle & Clicker	2	`alias_alchemy_shop`, `hell_casino`
10	Adventure & Narrative	6	`cold_case_archive`, `u003_acting_human_again`, `whispers_of_the_wild`, `harukaze_student_council_casebook`, `last_announcement`, `backwalker_camp`
11	Sports & Racing	3	`last_catch`, `pixel_pennant`, `toybox_rider`

A.6 EXAMPLES OF GDDS

The following examples show how the same document conventions apply to both splits.

Figure S2 shows an excerpt of the Small-split GDD `traces_left`. Its prose describes the game and its systems, while rule-table rows state situations and their required outcomes. Constants such as the 0.800 s and 1.800 s deadlines have canonical definitions that other sections reference. Each labeled row provides a requirement that can be checked and cited: for example, a build that leaves the phase unchanged when `T4`'s stated conditions and trigger hold fails that row's required outcome. Selected rules from these rows appear in the contract graph in Figure S5.

Figure S3 shows a Big-split GDD, `chameleon_slide`, under the same conventions. The form doesn't change, but only the reach of each statement does. The pillars and the player card describe one action by citing several rule tables at once — a slide is stopped by `RT-STOP`, hides the player by `RT-CAMO`, and advances the world by `RT-TURN` — and the rule shown, `RT-COLOR K1`, is one that a build must honor in the middle of that chain.

```
# Place Sensors, Find the Cause -- Game Design Document

## 1. Overview
Place Sensors, Find the Cause is a single-player abstract inference game on one fixed directed
graph. Three labeled cause ports, A/B/C, have known deterministic routes through relay nodes L/R to
a shared output O. Before seeing a traversal, the player places exactly two sensor tokens on L, R,
or O. The player then runs the hidden cause once, observes which placed sensors latched LIT, and
chooses A, B, or C.
The one puzzle uses hidden cause B. Its route visits both relay nodes and the output. The graph and
all three possible routes are visible before placement; only the active cause is concealed. A
session permits exactly one traversal. After Run, sensor placement locks permanently until Retry
starts a new identical session.

## 2. Design Pillars
1. **Placement creates information.** Exactly two of three eligible nodes must be instrumented
before the single traversal. The useful decision happens before observation.
2. **Rules are visible, cause is not.** Every graph edge and all three route definitions are
readable on the playfield. Traversal never animates a hidden path; only placed sensors reveal
visits.
3. **One run means one commitment.** Activating Run locks placement before any signal event. There
is no reset, replay, third sensor, or second traversal inside the session.
4. **Inference is explicit.** After traversal, the player selects A/B/C directly. The Result
identifies the hidden cause and compares the observed signature with the full route table.
5. **Determinism supports reasoning.** Fixed cause B, exact route timing, exact latches, and fixed
event priorities make every outcome reproducible.
6. **Small visual grammar.** Ports, relay squares, one output diamond, arrowed edges, sensor rings,
and LIT/DARK labels are the entire playfield language.

### End conditions
- **Solved:** `RT-INFER` I1 when selected cause is B.
- **Not Solved:** I2 when selected cause is A or C.
- **Abandon:** Esc during PLACE or INFER returns to TITLE without a Result.
[··· Core Loop table, section 4 and 5.1-5.3 omitted ···]
### 5.4 Trace Resolver
`TRACE` performs exactly one traversal.
- **Entry transaction:** assert exactly two sensors and traceUsed=false; set traceUsed=true; set
  both latches DARK; clear visit flags; set traceElapsed=0; enter TRACE. If any assertion fails,
  enter DIAGNOSTIC rather than partially running.
- **Relay event at 0.800:** obtain relay visit set from hidden route. B yields `{L,R}`. For each
  placed sensor in that set, set LIT and trigger pulse/tone once.
- **Output event at 1.800:** O is visited. If sensor O is placed, set LIT and trigger pulse/tone
  once.
- **Completion at 2.600:** leave latches visible and enter INFER. Any placed sensor still DARK
  remains DARK.

### 5.5 Inference Resolver
- **Selection:** click a cause button or press A/B/C. The first valid selection writes selectedCause
  and compares against `DBT-CAUSE` through `RT-INFER`.
- **Outcome:** selected B is SOLVED; A or C is NOT_SOLVED. Sensor placement quality does not itself
  determine outcome.
- **Ambiguity:** the game permits non-distinguishing placement. It neither warns nor blocks it. The
  player may still choose, but only L+R separates all three possible routes.
[··· 5.6, RT-PHASE, RT-PLACE omitted ···]
### RT-TRACE -- deterministic visits
| Row | Situation | Single outcome |
|---|------|--------|
| T2 | reliable time crosses 0.800 s and relayVisited=false | set flag; obtain hidden relay set from
  `DBT-ROUTES`; latch/pulse each placed visited relay; schedule node tones |
| T3 | reliable time crosses 1.800 s and outputVisited=false | set flag; visit O; if O sensor
  placed, latch/pulse it and schedule O tone |
| T4 | reliable time reaches 2.600 s after required events executed | leave final latches visible;
  play Trace Complete; enter INFER |
[··· RT-INFER, RT-FOCUS and sections 7-13 omitted ···]
```

Figure S2: **A *Small* golden GDD example.** Excerpt of `traces_left`; omitted sections are marked.

```
# Chameleon Slide -- Game Design Document

## High Concept
A 2D top-down stealth puzzle. You are a small chameleon that can only move by **sliding**: press a
direction and it slides in a straight line at a fixed speed until it hits something, then stops.
While standing still you **change your body color** (keys 1-4); if your color matches the floor tile
you are resting on, sight beams from robot guards and cameras pass right over you. Each stage is a
hand-built grid puzzle scored 1-3 stars on **how few slides** you used. Six themed worlds of ten
stages each (plus a bonus world) introduce one new element at a time -- patrolling sentries, rotating
cameras, color-shifting floors, pits and bumpers, moving walls -- and stack them toward 15x15
brain-benders.

## Design Pillars
*1 -- One press, one committed slide.** Movement is a single directional tap. The chameleon travels
at `DBT-2` (480 px/s) in a straight line until an obstacle stops it (`RT-STOP`); you never steer
mid-slide. Because the board is static during your slide, the landing tile is fully predictable, so
every move is a deliberate plan, not a reflex.
*2 -- Color is safety, chosen before you leap.** Matching body color to the tile you STOP on hides
you from every sight beam (`RT-CAMO`). You can recolor only while stationary (`RT-COLOR`), and your
color is locked from the instant you commit a slide until you next stop -- so you must pick the one
color that keeps you hidden both the moment you land AND after the guards take their step (`RT-TURN`
A/B).
*3 -- Clearing is easy; fewest moves is the game.** Every stage awards 1-3 stars on completed slides
alone (`RT-STARS`). Reaching the exit is a gentle floor; the 3-star line is a routing optimization
that rewards re-planning the whole path. Mastery, replay, and "one better run" live here.
*4 -- Deterministic, legible stealth.** Everything that moves -- guards, cameras, color-shifting
tiles, moving walls -- advances exactly one step per player move, from fully visible state. Sight
beams are always drawn; tiles are always color-coded (with shape cues, `DBT-4`). You lose because of
a plan you can see, never because of timing you could not.

## Core Loop
One "move" is one slide plus the world's single-step response. Target **3-5 s**.
Macro loop: solve stage → see star result vs par → replay for a better star, or advance. Meta loop:
clear a world → unlock the next; bank stars → unlock the bonus world at `DBT-17`.
[··· Glossary, RT-TURN, RT-STOP, RT-VISION, RT-CAMO omitted ···]
### RT-COLOR -- when the chameleon may recolor
| Row | Situation | Outcome |
|---|------|-----|
| K1 | Key 1-4 pressed while **at rest** | Body color set to that color instantly (`DBT-12`); free;
 does **not** advance the world or the move counter |
| K2 | Key 1-4 pressed while **sliding** or during the world step (S-e) | ignored |
| K3 | Key pressed for the color already worn | no-op (no VFX/SFX) |
[··· RT-DETECT, RT-STAGE, RT-STARS omitted ···]
### Chameleon (Player)
The only player-controlled entity. Occupies exactly one tile; always axis-aligned to the grid.
- `V1 Slide` -- press W/A/S/D at rest → slide N/S↑... (up=W, left=A, down=S, right=D) at `DBT-2` (480
 px/s, constant velocity, no steering). Where it stops is owned by `RT-STOP`; the full turn it
 triggers is owned by `RT-TURN`. Reach: unlimited within the lane until `RT-STOP` fires. Hitbox: a
 56x56 px square centered in its 64 px tile (`DBT-1`), used only for the Pit/Exit/Bumper
 center-overlap tests; slide blocking is grid-cell based, not pixel based. Edge case: a direction
 that yields `RT-STOP R0` does nothing and costs no turn.
- `V2 Recolor` -- press 1/2/3/4 at rest → set body color; owned by `RT-COLOR`. Free, instant
 (`DBT-12`), does not advance the world.
- `V3 Camouflage` -- passive; whether beams can see the chameleon is owned by `RT-CAMO` and consumed
 by `RT-DETECT`. No input.
- `E1` edge: while sliding, color input is ignored (`RT-COLOR K2`) and directional input is handled
 by `RT-TURN T0` (slide buffering).
[··· remaining Actor and Tile Cards omitted ···]
## Progression & Scoring
- **Content:** 6 main worlds x 10 stages + 1 Bonus world x 10 stages = **70 stages** (`DBT-16`); max
 **210 stars** (3 per stage, `RT-STARS`).
[··· world roster, Art, Input, Audio, Levels omitted ···]
```

Figure S3: **A *Big* golden GDD example.** Excerpt of `chameleon_slide` GDD; omitted sections are marked.

B Details on Contract Generation

B.1 Dependency Structure in Game Specifications

As summarized in Figure 2b, we measure how often sentences in external game-design documents are classified as causal, on three corpora separate from the benchmark GDDs: 10 publicly available studio design documents ranging from short pitch documents to long-form production specifications (Table S6; 11,741 sentences, 37.9% causal), 246 specifications from GameCraft-Bench (Luo et al., 2026) and GameGen-Verifier (Jia et al., 2026) (141 and 105 documents), and 97 English-language primary rulebooks of BoardGameGeek's top-100 titles covering setup, play, and end conditions, without player aids, FAQs, or fan summaries; in total 353 documents and 73,700 sentences.

We normalize the extracted text, remove formatting artifacts and non-prose content, segment the prose into sentences, and label each sentence causal or non-causal with the published CiRA classifier (Fischbach et al., 2021); a corpus's causal rate is the share of retained sentences labeled causal. As shown in Table S7, the rate ranges from 32.0% to 45.3% across the three corpora, with a corpus macro average of 38.4% (the value in Figure 2b) and a pooled sentence-level rate of 42.0%, against roughly 28% reported for general requirements documents (Frattini et al., 2023); causal statements thus form a substantial portion of game-design specifications across sources.

Table S6: **Studio game-design documents used in the causality analysis.** Sentence counts are computed after text normalization and non-prose filtering; causal rates are CiRA predictions over the retained sentences.

Document	**Sentences**	**Causal (%)**
The Sky Above, the Sky Below	2,069	38.3
Frontier Pharmacist	1,996	42.5
Leisure Suit Larry 5	1,701	38.3
Love for Sail	1,551	37.5
Shape Up or Slip Out	1,444	44.6
Claw	1,428	29.6
Leisure Suit Larry's Casino	797	31.2
Planescape: Torment	483	34.8
Diablo (pitch)	143	35.0
Grand Theft Auto	129	29.5
Total	**11,741**	**37.9**

Table S7: **Causal-sentence estimates in external game-design corpora.** Rates are CiRA predictions on retained prose. The pooled rate weights corpora by sentence count, while the macro average gives equal weight to the three corpora.

Corpus	**Documents**	**Sentences**	**Causal** (%)
Studio GDDs	10	11,741	37.9
Game benchmark specifications	246	11,441	32.0
Board game rulebooks	97	50,518	45.3
Pooled	353	73,700	42.0
Macro average over corpora	–	–	**38.4**

B.2 Process of Generating Rule

We construct the contract $\mathcal{C}_i = (\mathcal{R}_i, \mathcal{E}_i)$ from the GDD d_i before inspecting any generated build. The procedure separates identifying the requirements from formalizing their conditions and effects: it first fixes the terminology and source entries, then uses them to constrain rule generation. The six stages in Figure S4 combine model-based interpretation with deterministic extraction, assembly, and validation.

A rule $r = (t_r, \phi_r, F_r)$ specifies an event t_r, the preconditions ϕ_r under which it applies, and the expected effects F_r, expressed as state updates or event emissions. An invariant $r = (\varphi_r, \sigma_r)$ instead specifies a condition φ_r that must hold throughout its scope σ_r. Dependencies in $\mathcal{E}_i$ connect a rule's effects to the conditions or triggering events of other rules. These links are derived from the rule fields rather than generated as a separate list of relationships.

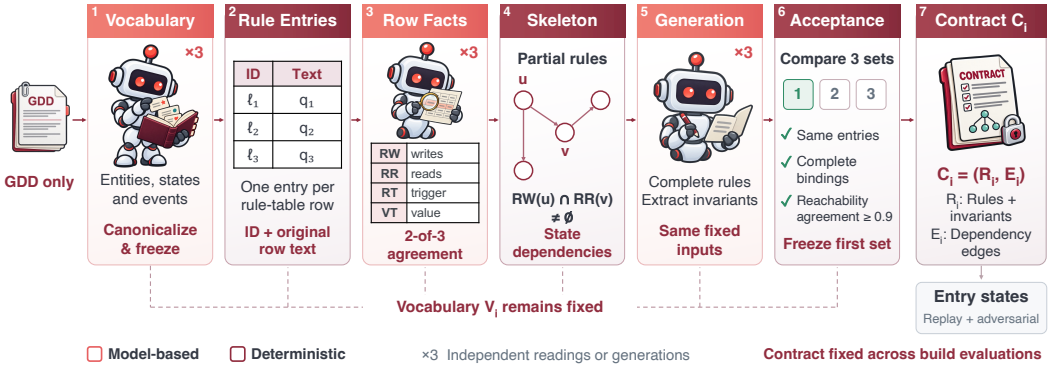


Figure S4: **Contract construction from a GDD.** The pipeline fixes vocabulary and rule entries before completing rule definitions and validating the contract. Coral boxes denote model-based stages, while dark-red boxes denote deterministic stages without model calls. The $\times 3$ labels indicate independent repetitions: row facts are consolidated by two-out-of-three voting, whereas generated contracts are compared for structural consistency before one is frozen for evaluation. The dashed line indicates that the vocabulary V_i remains fixed throughout subsequent stages.

Stage 1: Vocabulary (Model). We establish a shared vocabulary V_i for the entities, state attributes, and events described in the GDD. Three independent readings collect candidate terms, after which synonymous terms are grouped and assigned a common name. The resulting vocabulary is fixed for subsequent stages. This lets requirements that refer to the same game state use the same identifier, even when the GDD describes that state in different ways.

Stage 2: Rule Entries (Code). Code extracts candidate rule entries L_i from the GDD's labeled requirement tables. Each entry $\ell = (\mathrm{id}_\ell, q_\ell)$ retains its identifier and original row text. Generated rules remain linked to these entries through the provenance map $\pi : \mathcal{R}_i^{\mathrm{rule}} \to L_i$. The source entries therefore determine which requirements are to be formalized, rather than leaving each generation to choose a new set of evaluation targets.

Stage 3: Row Facts (Model). For each entry, the model identifies four facts using the fixed vocabulary: the state attributes it changes, $\mathrm{RW}(\ell)$; the attributes its conditions inspect, $\mathrm{RR}(\ell)$; its triggering event, $\mathrm{RT}(\ell)$; and the operation and value associated with each update, $\mathrm{VT}(\ell)$. A condition-only rule has no triggering event. Each extraction is repeated three times independently. Read and write sets retain elements supported by at least two readings, while triggers and update values require two matching answers. A trigger or value without a majority remains unresolved and is completed from the source text during generation.

Stage 4: Skeleton (Code). Given the fixed row facts, code assembles partial rule definitions and a state-dependency graph. Two entries are linked when one updates an attribute inspected by the other, that is, $\mathrm{RW}(\ell_u) \cap \mathrm{RR}(\ell_v) \neq \varnothing$. Through the source-entry associations, these links determine the state dependencies used to compute the downstream counts in Equation 3. Each partial rule also records its known trigger, the attributes used by its preconditions, and the attributes changed by its effects. The next stage completes these fields rather than reconstructing the rule set. Event dependencies are derived from emitted events and matching triggers once the full rule fields are available.

Stage 5: Generation (Model). The model completes each partial rule using its source text q_ℓ, writing precondition predicates and effects consistent with the fixed read/write facts and update specifications. Any unresolved trigger or value is determined from the same source text. When a

value cannot be expressed as a literal or supported symbolic reference, its document description is retained as an *opaque* value instead of inventing a constant. Invariants are extracted separately from the GDD's state and constraint specifications, with supporting quotations. This stage produces three independent rule and invariant sets under the same fixed inputs.

Stage 6: Acceptance (Code). Programmatic checks validate the generated fields and their source associations, and recompute the dependency graph from the rule definitions. The three generations are compared for source-entry consistency, provenance binding, and dependency reachability. The acceptance check requires the same source-entry set, complete rule-to-entry binding, and reachability agreement of at least 0.9. Once accepted, the first generation is frozen as $\mathcal{C}_i$ for all subsequent build evaluations. This fixes the detailed rule definitions as well as the evaluation targets and dependencies; Section B.5 examines the variation that remains between independent generations.

Entry States. Requirements sharing an entry condition are grouped into the canonical scenario set $\mathcal{S}_i$ used for replay evaluation. Each scenario specifies the state a build must establish before the replay begins. For targeted adversarial playtests, the harness similarly initializes the preconditions of a rule left unverified by normal play. Initialization supplies the starting conditions, not the expected outcome: the subsequent behavior must still be exercised and observed. These entry specifications support execution of the contract and are kept separate from the rules and dependency edges in $\mathcal{C}_i$.

B.3 Guarantees of the Contract

The contract accepted after Stage 6 provides a fixed representation for build evaluation. First, it separates triggers, preconditions, and effects, giving each judge explicit fields against which to organize its evidence. Second, dependencies are derived from the rule fields using a fixed vocabulary. State dependencies connect rules when one updates an attribute that another inspects, while event dependencies connect an emitted event to a matching trigger. Reach, roots, and leaves are computed from the resulting graph, which identifies downstream requirements that may be affected by a failed rule. Third, the evaluation targets are fixed before any build is inspected. Each accepted rule retains its source association with the GDD, and values that cannot be formalized are preserved as document descriptions rather than replaced with invented constants. Every build is evaluated against the same accepted contract.

B.4 Example of Contract

Figures S5 and S6 follow `traces_left` from its GDD to a judged normal playtest. The GDD is shown in Figure S2. Each source row describes a situation and an outcome, which are formalized as the rule's trigger, preconditions, and effects. For example, `P2` adds a sensor when an eligible node is empty, the phase is PLACE, and fewer than two sensors are placed. It therefore both reads and updates `sensorNodes`.

Figure S5 is an execution view linked to the fixed contract, rather than the complete static contract graph. Repeated nodes represent occurrences of the same rule during the recorded playtest. Three placements and one removal establish the sensor count required by `P4`. Committing the run sets the phase and initializes the clock used by subsequent timing rules; `T4` completes the trace, after which `I1` records the outcome. The focus-handling sequence is displayed separately for readability, although its rules also share `phase` with the main sequence.

The example illustrates how source-linked rules make execution evidence interpretable in its dependency context. A missing prerequisite can leave downstream behavior unverified, but does not automatically make every downstream rule violated. Section B.5 examines consistency across contract generations, and Section F.1.2 examines how downstream reach affects source-code scoring.

B.5 Stability of Contract Generation

Contract construction is partially deterministic: model-based extraction supplies the vocabulary and row facts, code fixes the rule entries, binds them to their source GDD rows, and assembles the state-dependency skeleton from the extracted read/write sets, and rule generation then completes the preconditions and effects within that structure. To measure what this leaves open, we independently

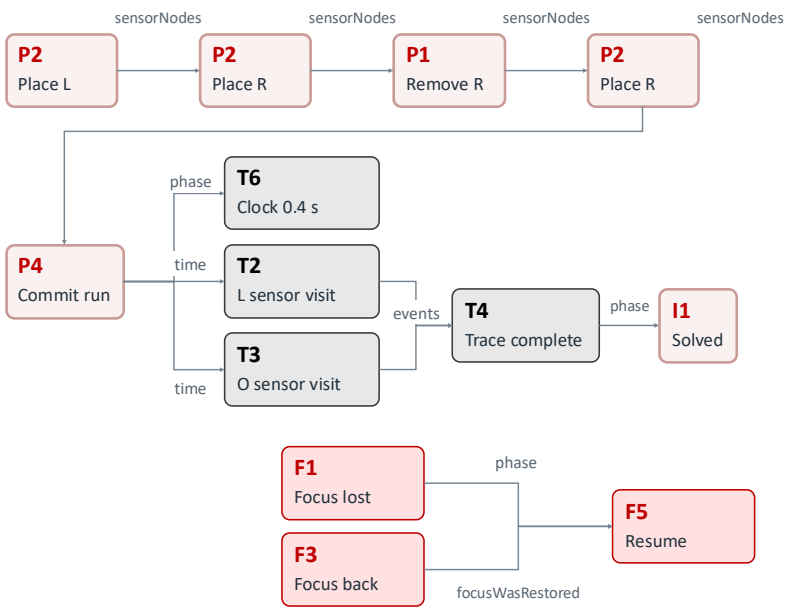


Figure S5: **Contract-linked execution of `traces_left`.** Nodes show rule occurrences in a recorded normal playtest, and labeled links indicate the state attributes connecting them. Three sensor placements and one removal precede `P4`, which commits the run. The fixed contract supplies the rule identities and dependencies; the trace supplies their observed order. Source GDD rows appear in Figure S6.

generate rules three times for each of the 50 *Small* and 50 *Big* GDDs under the same fixed inputs and compare the 300 outputs.

As shown in Table S8, all three generations preserve the same rule entries and source associations, and the dependency graphs are nearly identical, with reachability agreement, dependency-edge Jaccard similarity, and role preservation all at or above 0.999; within each rule, the `trigger` name agrees in 0.94 of cases and the `predicate` structure, the entities, attributes, and operators used in preconditions and effects, in 0.70. Table S9 shows what the remaining variation looks like: the same row encoded with `SET` or `INCREMENT`, or a bound written as a comparison or as a set.

We can check whether this variation changes scores directly, because each generation was also used to judge the same build. Here, the unit is a rule in the dependency graph, which represents a source row's conditions, trigger, and state-changing or event-emitting outcomes. The 869 *Small* and 1,860 *Big* rules are therefore fewer than the individual outcome requirements counted in Table S3. Pairing the three judgments for each rule yields three pairs per rule: 2,607 over the 869 *Small* rules and 5,580 over the 1,860 *Big* rules, with no rule excluded since the generations share the same entries. Re-judging a rule whose predicate structure is identical across two generations changes its score by at least 0.25 in 12.7% of pairs on *Small* and 8.8% on *Big*, a variation that comes with an agent judge and is of a similar magnitude to the status changes of the naive judge on its matched items; when the predicate structure differs, the rate is 17.7% and 11.1%, an increase of 5.0 and 2.3 percentage points (Table S10). The contract therefore removes the variation in what is judged, which the naive judge cannot, while the remaining variation in how a rule is defined raises the rate of score changes only modestly; because that definition still reaches the score, an accepted contract is frozen in full, so that every build and revision round is judged against the same rule definitions and dependencies.

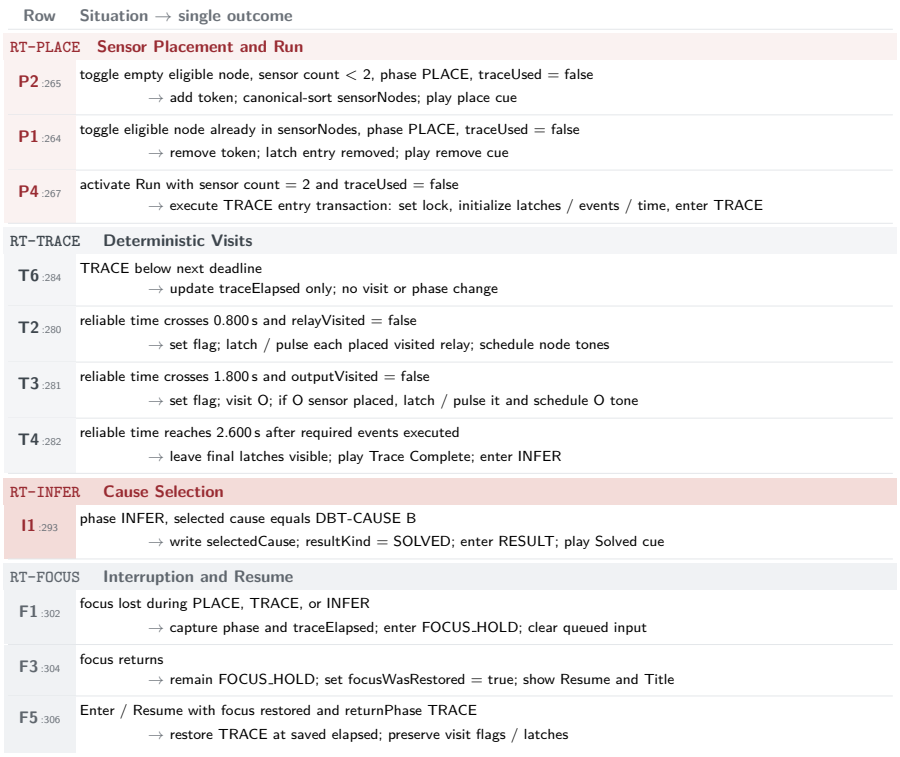


Figure S6: The GDD rows the entries in Figure S5 were derived from, in the four sections of `traces_left` that the run touched, each labelled with its line in the framed GDD. A row states a situation and its required effects (→); the pipeline formalizes the situation as the rule's trigger and preconditions, and the required effects as state updates or event emissions.

Table S8: **Consistency of evaluation targets across repeated generations.** Three rule generations per GDD on 50 *Small* and 50 *Big* GDDs with fixed vocabulary and rule entries, averaged over the two splits; items are matched by their source entries.

Agreement Measure	**Contract**
Matched Items	100%
Dependency Graph	≥ 0.999
`Trigger` Name	0.94
`Predicate` Structure	0.70

Table S9: Rules whose predicate structure differs between two contract generations, with the GDD row they encode and the source-code score $f^{\text{src}}_{i,r}$ each generation received on the same build.

Game, row	GDD row (abridged)	Generation A	Generation B	$f^{\text{src}}_{i,r}$ A / B
`untangled_cords` R8 (*Small*)	All other frames: keep state, keep updating `t` in PLAY_ACTIVE	`timer INCREMENT ⟨dt⟩`	`timer SET ⟨advanced by dt⟩`	1.00 / 1.00
`magma_arc` STAR2 (*Big*)	Not STAR3, and hit events ≤ 3 this attempt: 2 stars	`event <= 3 STAR SET 2`	`event IN [1, 2, 3] STAR SET 2`	1.00 / 1.00

Table S10: Effect of predicate-structure variation on the source-code score. Rule pairs across the three generations, three per rule, judged on the same build; the last column is the share of pairs whose per-rule scores differ by at least 0.25.

Split	**games**	**rules**	**Predicate structure**	**pairs**	$\lvert\Delta f^{\mathrm{src}}_{i,r}\rvert \geq 0.25$
Small	50	869	identical	1,773	12.7%
			different	834	17.7%
Big	50	1,860	identical	4,045	8.8%
			different	1,535	11.1%

B.6 Value Compatibility

In construction, an edge of $\mathcal{E}_i$ is formed at the attribute level: it records that one rule updates a state attribute that another rule's condition or trigger evaluates, without checking the written value against the reading condition, so two rules that write and read the same enumerated attribute with different values are linked as well. The scoring built on the graph is insensitive to this choice: the dependency-weighted rule score F_i^{rule} computed on the reduced graph agrees with the registered one at Spearman 0.99 on *Small* and 1.00 on *Big* with a maximum per-game difference of 0.07, the defining root of each root-defective game remains a root in 17 of 18 games, and the sign counts of Appendix F.1.2 stay at 11 of 18 root-defective games and move from 6 to 7 of 16 comparison games. We therefore keep the attribute-level definition, which is computable from the contract alone, and read its edges as potential rather than guaranteed prerequisites.

B.7 Dependency-Adjusted Rule Pass Rate

Using the source-code judgments and contract graphs of 100 GPT-5.6-Sol games, we compare local and dependency-adjusted rule pass rates. A rule passes locally when its aggregated source-code score is at least 0.8. The dependency-adjusted rate serves as a *graph-based reachability proxy*: a rule counts only when it and all its direct and indirect predecessors pass. Both rates use the same dependency-linked rule set as the denominator within each game and are averaged across games. Figure 2c reports 72.4% locally and 22.7% after this check, a 49.7-point gap. Since edges encode potential prerequisites, exclusion indicates possible blocking rather than demonstrated runtime unreachability. This rule-level diagnostic does not change the benchmark's source-code score or propagate violation verdicts to downstream rules during playtesting.

C EXPERIMENTAL DETAILS

Game Development Environment. The 100 tasks in the main benchmark focus on 2D single-player browser games developed using Phaser. All coding agents start with the same project and runtime-interface specification, using Phaser 4.1, TypeScript, and Vite. Given a GDD, the agent creates the source project along with a runnable web build within this environment. The fixed evaluation contract is kept apart from the game-building instructions. This common development environment enables a controlled comparison across coding agents. Extensions to a Three.js runtime are presented in Section I.

Open-Weight Model Serving and Generation. We generate games with DeepSeek-V4-Pro-0813 (DeepSeek-AI, 2026), GLM-5.3 (GLM-5-Team, 2026), and Kimi-K2.7-Code (Kimi Team, 2025), each served with vLLM 0.28.0 (Kwon et al., 2023) using four NVIDIA B300 GPUs per allocation on a shared `slurm` cluster. Each cluster node has eight NVIDIA B300 GPUs, 128 vCPUs, and about 4 TB RAM. GLM and Kimi use tensor parallelism across the four allocated GPUs, while DeepSeek uses data parallelism with expert parallelism. Generation follows each checkpoint's default sampling parameters: temperature 1.0 and top-p 0.95 for GLM and Kimi, and temperature 1.0 and top-p 1.0 for DeepSeek. For each model, we independently generate a game three times per GDD without best-of-k selection. Serving allocations total approximately 1,100 GPU-hours.

Evaluation Execution and Resources. For runtime evaluation, we serve production builds locally and run them in Chromium via Playwright at each game's declared viewport (1280×720 by default). Scenario-based replay applies recorded input sequences and captures rendered frames, while adaptive playtesting selects keyboard and pointer inputs based on observations of the running game. Generated games expose shared interfaces for state observation and controlled precondition initialization. Section D details the three evaluation axes, including how these interfaces support normal and adversarial playtests. Evaluation workloads run on ten AWS `m7i.4xlarge` instances, each with 16 vCPUs, 64 GiB RAM, and Ubuntu Server 24.04 LTS (noble) on x86-64 hardware, plus one workstation with comparable specifications. The total measurement time is approximately 900 wall-clock hours (source-code evaluation takes 3.1%, scenario-based assessment replay takes 40.1%, and playtest takes 56.8%). Games are rendered and replayed on headless Chromium 149.0.7827.55 (Chrome Headless Shell) via Playwright 1.61.0.

D Details on Evaluation Setup

D.1 Source Code Evaluation

D.1.1 Prompt

Source-Code Judge Prompt

```
[...]
Static reading only: the question is "is this commitment implemented in the code",
     never "did it fire in some normal playtest" (that is a separate, trace-grounded
     measurement).
[...]
- fidelity is a real number 0.00-1.00 -- the degree to which the Rule as stated is
     implemented. Calibration anchors: 1.00 = fully as promised; 0.85 = minor deviation
     that does not change behavior in stated cases; 0.5 = about half the stated
     behavior (a condition unenforced, an effect missing); 0.2 = only traces; 0.00 =
     absent or contradicted. Use any intermediate value that best reflects the degree.
Judge the Rule as specified, not as you would have designed it. A reasonable
     alternative that does NOT do what the Rule states scores 0.
[...]
Judge a Rule EXISTENTIALLY -- does the stated mechanism exist with its stated gates and
     effects. Whether OTHER paths could interfere with or corrupt it (a stale callback,
     a re-entrant handler, an async edge that skips a re-check) is a universal property
     and belongs to the invariant checks, not here
[...]
- ok: true -- the code establishes the property and no reachable path violates it in
     scope.
- ok: false -- a code path can break it in scope, or nothing establishes it at all.
[...]
For every item, `evidence` cites what you actually read -- a file and the relevant
     function/line region, one sentence on how it implements (or fails) the item.
     "Probably implemented" without a location is not evidence; if you could not find
     it, the score is 0 and the evidence says what you searched.
[...]
```

Figure S7: **Excerpt of the source-code judge system prompt.** Text is verbatim, and omitted passages are marked `[...]`.

The source-code judge is one prompted model call per build. Its system prompt states the question, the continuous scale for rules with its calibration anchors, the two-valued outcome for invariants with the frame-boundary and injection-surface exclusions, and the evidence requirement; Figure S7 reproduces the passages that shape the judgment. The task message is assembled by code from three parts: the rule set $\mathcal{R}_i^{\text{rule}}$ in the order to be judged, the invariant set $\mathcal{R}_i^{\text{inv}}$, and the path of the read-only source directory of g_i. The judge answers through a fixed-schema tool call with one entry per rule ($f_{i,r}^{\text{src}} \in [0,1]$ and evidence) and one per invariant ($f_{i,r}^{\text{src}} \in \{0,1\}$ and evidence), and a validator rejects a submission that omits an item, breaks the order, or leaves evidence empty.

D.1.2 Implementation

Inputs and Session. The source-code judge receives the rule set $\mathcal{R}_i^{\text{rule}}$ and the invariant set $\mathcal{R}_i^{\text{inv}}$ for contract $\mathcal{C}_i$, along with read-only access to the `src/` directory of build g_i. For each item, the judge answers a static question: is this commitment implemented in the code? The judge does not execute the game. Each session evaluates a single build and must return exactly one entry per rule and per invariant, in the specified order, each accompanied by a score and supporting evidence. Submissions that omit items, disrupt the order, or lack evidence are rejected and must be resubmitted. Given the density of generated sources, the judge examines entire files rather than inferring absence from line counts, and does not reuse findings as evidence for multiple rules.

Scoring a Rule. Each rule is decomposed into preconditions, trigger, and effects, accompanied by its originating GDD sentence. The judge assigns a continuous score $f_{i,r}^{\text{src}} \in [0,1]$ reflecting the extent

to which the causal commitment is implemented, specifically whether the code gates the trigger on the stated conditions and produces the specified effects. The scoring scale is anchored: 1.00 indicates full implementation as specified, 0.85 denotes a minor deviation that does not alter behavior in the stated cases, 0.5 represents partial implementation (such as an unenforced condition or missing effect), 0.2 indicates only traces of implementation, and 0.00 signifies absence or contradiction. Intermediate values are permitted when they more accurately reflect the degree of implementation. Contract names are mapped to code identifiers by meaning. Rules are evaluated as specified, not as reasonable alternatives, and the assessment is existential; potential corruption by other code paths is addressed through invariant checks.

Checking an Invariant. An invariant is defined as a predicate over the game state that must hold at every frame boundary within its declared scope. The judge assigns $f_{i,r}^{\text{src}} \in \{0, 1\}$: a value of 1 if the code establishes the property and no reachable gameplay path can violate it within scope, and 0 if any path can break it or if the property is not established. Differences in encoding, such as a 0-based index representing 1-based numbering or alternative enum spellings, are not considered violations. The state-injection surface present in every build for the verifier (`setState`, scene mutators, scenario entries) is not treated as a violating path, as it is designed to permit arbitrary state modifications.

Evidence. Every score cites the file and the function or line region the judge read, with one sentence on how that code implements or fails the item; a rule the judge cannot locate scores 0, and its evidence records what is searched. A validator rejects submissions whose evidence strings largely copy one another or that score every rule zero without locating anything.

Aggregation. The rule component F_i^{rule} is calculated as the dependency-weighted mean of the continuous per-rule scores, using the weights $w_{i,r}$ defined in Equation 3. Here, $D_{i,r}$ represents the number of rules reachable from r via links where one rule updates a state that another rule's condition evaluates. All weights are set to 1 when $D_i^{\max} = 0$. The invariant component F_i^{inv} is the fraction of invariants that hold, and is set to 1 if no invariants are declared for a contract. The overall source-code score is the product of these two components (Equation 4).

Runs. Each build is evaluated in three independent sessions, and the mean score is reported. The GPT-5.6-Luna model is used at high effort for all measurements. Appendix G.2 presents judge comparisons and cost analyses.

D.2 Scenario-Based Replay Assessment

D.2.1 Prompt

The replay judge is two prompted model calls per scenario replay. The frame-selection call receives the rubric ρ_i^{vis}, the executed input trace $\xi_{i,q}$ with the engine's render timestamps, and the observation budget M, and returns a plan of replay positions per rubric item that a deterministic tool resolves to recorded frames; the scoring call receives only the resolved frames and the rubric and returns one score, one rationale, and the cited frame ids per item through a fixed-schema tool call. The scoring prompt is prefixed with the GameCraft-Bench scoring instruction, which defines the 0–1 scale; Figure S8 reproduces the passages of both prompts that shape the evidence.

D.2.2 Implementation

The scenario-based replay assessment, described in Section 3.3.2 keeps the evaluation protocol of GameCraft-Bench (Luo et al., 2026) where the protocol is engine-neutral: the build gate, the four rubric categories with weights $0.15/0.35/0.15/0.35$, the per-item score in $[0, 1]$, the per-item aggregation over replays by max or mean, and the score formula are reused unchanged, and only the Godot replayer is replaced by a Playwright replayer for browser builds, with the same trace format of timed input events. Three parts differ, because the contract rather than the submitter decides what is played, where the evidence is taken, and what is asked.

Scenario Replays. GameCraft-Bench scores the demonstration traces a submitter ships with the project, capped in number and judged on a fixed-length window of each. We replay one fixed policy

Frame-selection prompt (excerpt)

```
[...]
The executed input trace is a temporal index, not visual proof. The pre-trace and
    post-trace context frames are already included as deterministic replay boundaries.
    Map every rubric target to any additional replay intervals, first POST_RENDER
    points after inputs, or replay boundaries needed beyond that context. A frame may
    support several targets, and one target may use several locations.
After duplicate frames are removed, the complete plan must remain within the declared
    maximum observation count. Do not invent time offsets, game-specific event
    categories, visual outcomes, or composite images.
[...]
```

Scoring prompt (excerpt)

```
Only the individual original-resolution images in the user message are visual evidence.
    The frame metadata identifies their replay position and intended targets, but it
    is not proof by itself.
Score every original rubric requirement. Cite at least one observed frame ID for every
    score and explain only what the cited visual evidence supports. Do not treat the
    scenario text, expected outcome, trace metadata, or source code as proof that a
    rendered behavior occurred.
```

Figure S8: **Excerpts of the two replay-judge system prompts.** Frame selection over the recorded replay and scoring from the selected frames. Text is verbatim, and omitted passages are marked `[...]`.

$\xi_{i,q}$ per canonical scenario $s_{i,q} \in \mathcal{S}_i$. For each build, A^{build} declares the inputs and their timing, and a tool compiles the declaration into the trace format, rejecting scenarios outside $\mathcal{S}_i$. Every canonical scenario is replayed and scored. Thus, the scenario set is fixed across builds of the same game, while each build supplies its own fixed input policy for those scenarios.

Frame Selection. The reference judge sees frames sampled at a fixed interval, at most forty per replay and from a random window when the replay exceeds the cap, so the frames are independent of the inputs that produce the behavior being judged. Our replayer records the whole replay together with the executed inputs, the engine's render timestamps, and a context frame before and after the trace; the judge's selection call maps each item of ρ_i^{vis} to the first render after the relevant input or to an interval between inputs, a deterministic tool resolves these requests to recorded frames and removes duplicates within the budget M, and the judge must cite at least one retrieved frame for every item it scores, which the code verifies before accepting the verdict.

Rubric. GameCraft rubrics are hand-written per task. ρ_i^{vis} is generated from the contract under the same schema: each item states a condition and its expected visible outcome with the 0, 0.5, and 1 anchors and stays linked to the rules or invariants it expresses, functional items and presentation items are kept in separate categories rather than sharing one criterion, the number of items per category follows what the contract supports (three to five), and the aggregation rule of each item is chosen by its meaning, max for a capability or a screen that one replay can prove and mean for a quality that must hold across all replays.

Reliability. The judge answers through a fixed-schema tool call with one score, one rationale, and the cited frames per item, so a response cannot omit an item or fail to parse; each replay is independently judged three times on the same frames and averaged, and the per-item scores are aggregated across scenarios by the item's rule and combined by the category weights into F_i^{replay} (judge settings in Section G.2).

D.3 Adaptive Playtest

D.3.1 Prompt

The adaptive-playtest pipeline uses three prompted roles: a normal-playtest author, an adversarial-playtest author, and a separate trace judge. The two authors instantiate A^{play} in Equation (6). Both receive the shared interface API as a fixed system-prompt prefix and the test objective ω as a JSON brief containing the target rules, dependency edges, and source paths. The normal objective ω_i^{pt} asks the bot to complete g_i and collect evidence for $\mathcal{R}_i^{\text{rule}}$; the adversarial objective $\omega_{i,r}^{\text{adv}}$ targets $r \in \mathcal{R}_i^{\text{adv}}$ and its upstream dependencies. Each author returns the bot $b_{i,\omega}$ as a JavaScript module.

The playtest author is confined to player input from z_i^0 and must follow the snapshot contract the judge binds to: every action aimed at a rule is followed by a snapshot labeled with the rule id and its trigger, and a snapshot taken before the action carries the suffix `-before`. The adversarial author establishes $z_{i,r} \models \phi_r$ through a declared scenario entry or a minimal `ctx.setState` at the root precondition, records it as an `ASSISTED_RULE` note, and then uses player input only, so $\tau_{i,r}^{\text{adv}}$ contains no state injection after initialization.

The trace judge restricts evidence to the snapshots, events, and notes of the trace $\tau_i(\omega, z)$, treats absence of evidence as unverified rather than violated, and answers through a fixed-schema call with one entry per requested rule; three such calls are combined by strict majority. Figure S9 reproduces the passages of the three prompts that shape the evidence.

D.3.2 Implementation

Shared Runtime Interface. The interface API is exposed to each JavaScript bot as a `ctx` object; Figure S10 shows excerpts of two recorded bots. `ctx.state()` reads `GameInspector.exportState()` and yields the observation o_t, while keyboard and pointer methods issue the player input x_t to the running game. The bot uses observations in conditions and loops, waits for state changes, and records evidence with `ctx.snapshot()`. Snapshots contain labels, elapsed times, game states, and available events, and together with the input log they form the trace $\tau_i(\omega, z)$ of Equation 7; `ctx.watch()` adds selected live-state observations. Adversarial tests additionally use declared scenario entries or `ctx.setState()` to establish $z_{i,r}$ before observing the specified behavior, whether it follows a player input or elapsed time.

Rule-Level Judgments. A separate evaluator checks each rule's conditions, trigger, and expected effects against the recorded trace, issuing a verdict only when the rule's required situation ϕ_r is established in the trace. The rule is *satisfied* when the evidence supports its expected behavior and *violated* when the observed outcome contradicts it. A promised value is interpreted by its meaning rather than its spelling, but the operation and number of fields must match: a `SET` with a different observed value, or an exposed field that contradicts a multi-field effect, constitutes a violation. Code re-checks every cited `SET` and overrides a satisfied judgment if it finds a contradiction.

When the evidence is insufficient, the rule is *unverified*: its required situation was never established in the trace. This status does not count as a violation. Each verdict retains the rule identifier, supporting execution evidence, and whether the situation was reached through normal play (τ_i^{pt}) or assisted initialization ($\tau_{i,r}^{\text{adv}}$). Source code may help interpret state fields, but does not serve as verdict evidence. Each rule receives three evaluator judgments; the strict majority determines its verdict, and a rule with no majority is treated as unverified.

Combining Results and Aggregation. Conclusive normal-playtest judgments form $\mathcal{R}_i^{\text{pt}}$ and are retained, and adversarial tests address $\mathcal{R}_i^{\text{adv}} \subseteq \mathcal{R}_i^{\text{rule}} \setminus \mathcal{R}_i^{\text{pt}}$. Across repeated adversarial attempts on the same rule, an observed violation takes precedence; otherwise, a supported satisfaction is retained. Each rule contributes once to the final result. Let $n_i = |\mathcal{R}_i^{\text{rule}}|$ and let $n_i^{\text{sat}} = \sum_{r \in \mathcal{R}_i^{\text{rule}}} f_{i,r}^{\text{adapt}}$ and n_i^{vio} count the rules finally judged satisfied and violated. The playtest score is $F_i^{\text{adapt}} = n_i^{\text{sat}}/n_i$, identical to Equation 10, and judgment coverage is $(n_i^{\text{sat}} + n_i^{\text{vio}})/n_i$. Both use the full rule set as the denominator. We report unweighted means of per-game values within each split.

Normal Playtest Author Prompt

```
[...]
- Use only normal player input after the initial page load.
- After every action aimed at one or more Rules, call
    `ctx.snapshot('<RuleId>[-<RuleId>...]-<what-happened>', { targetRule: ['<RuleId>',
    ...], trigger: { kind: 'input' | 'timer' | 'event', detail: '<what you did or
    waited for>' } })`. [...]
- A snapshot taken BEFORE the action (the situation, not the outcome) carries the
    suffix `-before` and no `targetRule`. [...]
[...]
```

Adversarial Author Prompt

```
[...]
- Use `ctx.setState` only when normal input cannot establish a root precondition.
- Record every assistance with `ctx.note('ASSISTED_RULE=<id>: <reason>')`.
Snapshot each attempted Rule before and after its trigger. If an expected transition
    does not occur, record the observed blocker and return normally; do not throw or
    assert.
[...]
```

Trace Judge Prompt

```
[...]
Verdict evidence must come from trace snapshots, trace events, or trace notes.
Absence of evidence is `unverified`, not `violated`. [...] A snapshot label is a
    navigation aid, not behavioral proof.
Submit exactly one entry for every requested Rule id in the supplied order.
[...]
```

Figure S9: **Excerpts of the three adaptive-playtest system prompts.** Text is verbatim, and omitted passages are marked `[...]`.

(a) The Shared ctx Interface

```
export const budgetMs: number;  export default async function run(ctx: Ctx);
interface Ctx {
  state(); live(fn); watch(fn); snapshot(label, extra?); note(msg);   // observation,
     read-only
  tap(key); press(keys); type(text); hold(key, ms?); down(key); up(key);   // keyboard
  click(x, y, button?); move(x, y); mouseDown(); mouseUp(); mouseHold(); drag(points);
     wheel(dx, dy);
  blur(); focus(); hide(); show(); pointerCancel();                    // window and
     document
  wait(ms); waitFrames(n); waitFor(predicate, opts?);                  // waiting
  setState(patch);                                          // adversarial only, refused
     in the normal playtest
}
export const scenario?: string;                             // declared scenario entry,
     adversarial only
```

(b) Normal Playtest Bot magnet_dock, excerpt

```
export default async function run(ctx) {
  const snap = (label, rules, kind, detail) =>
    ctx.snapshot(label, { targetRule: rules, trigger: { kind, detail } });
  ctx.watch(() => {                          // live-state fields attached to every
     snapshot
    const g = window.GameInspector?.exportState?.().game;
    return g && { state: g.state, polarity: g.polarity, settleTicks: g.settleTicks };
  });
  await ctx.snapshot('initial');
  await ctx.down('Space');  await ctx.waitFrames(2);   // one fresh toggle while held ...
  await snap('I5-first-polarity-toggle', ['I5'], 'input', 'fresh Space down');
  await ctx.down('Space');  await ctx.waitFrames(2);   // ... a second down must not
     toggle
  await snap('I3-held-space-repeat-inert', ['I3'], 'input', 'second down while held');
  await ctx.up('Space');
  if ((await ctx.state()).game.state === 'S_RESULT') {
    await ctx.click(640, 492);
    await snap('S13-result-to-title', ['S13'], 'input', 'clicked Title');
  }
}
```

(c) Adversarial Playtest Bot magnet_dock, rule D1, excerpt

```
export const scenario = 'dock_center_at_rest';       // declared scenario entry
export default async function run(ctx) {
  const gameState = async () => (await ctx.state())?.game ?? {};
  if ((await gameState()).settleTicks !== 59)        // pin the final pre-threshold tick
    await ctx.setState({ game: { state: 'S_PLAY', settleTicks: 59, x: 0, v: 0,
      polarity: 'Attract', positionQualified: true, speedQualified: true } });
  const before = await gameState();
  await ctx.snapshot('D1-before');
  let after = before;
  for (let i = 0; i < 8 && after.settleTicks < 60; i++) {
    await ctx.waitFrames(1);  after = await gameState();
  }
  await ctx.snapshot('D1-after', { targetRule: ['D1'], trigger: { kind: 'timer',
    detail: { prior: before.settleTicks, observed: after.settleTicks } } });
}
```

Figure S10: **The playtest runtime interface and two recorded bots for `magnet_dock`.** (a) Every bot receives the same `ctx`. (b) The normal playtest bot reads state, issues inputs, and records labelled snapshots naming the target rule. (c) The adversarial playtest bot enters through a declared scenario, pins the precondition of rule `D1` with `setState`, and observes the outcome.

E ADDITIONAL ANALYSIS ON BENCHMARK RESULTS

Comparison Protocol. We compare six proprietary coding agents using per-GDD scores from the 100-task benchmark: Claude-Fable-5.1 (Anthropic, 2026), Claude-Opus-5 (Anthropic, 2026), Claude-Opus-4.8 (Anthropic, 2026), GPT-6-Astra (OpenAI, 2026c), GPT-5.6-Sol (OpenAI, 2026b), and GPT-5.5 (OpenAI, 2026a). For each evaluation axis, a higher score on the same GDD contributes one win, a tie one half, and a lower score zero. Matrix entries average these outcomes over matched GDDs. We use scores before display rounding, including recorded zeros. Source uses graph-weighted fidelity with $\alpha = 0.5$. Overall compares the arithmetic mean of the three axis scores for each game before aggregating model-pair outcomes.

Elo Ratings. We summarize outcomes against all opponents using Bradley–Terry ratings on an Elo scale (Chiang et al., 2024). For each split and axis, the expected comparison outcome for model m against model n is

$$p_{mn} = \frac{1}{1 + 10^{(r_n - r_m)/400}}. \tag{14}$$

Let w_{mn} be wins plus half-credit ties and n_{mn} the number of matched GDDs. We fit all model-pair outcomes jointly by maximizing

$$\sum_{m<n} \left[\left(w_{mn} + \tfrac{1}{2}\right) \log p_{mn} + \left(n_{mn} - w_{mn} + \tfrac{1}{2}\right) \log(1 - p_{mn}) \right], \qquad \frac{1}{M} \sum_{m=1}^{M} r_m = 1{,}000. \tag{15}$$

Here $M = 6$ is the number of compared models. Symmetric half-win and half-loss regularization keeps ratings finite under complete separation and does not alter the empirical heatmap. Each axis is fitted independently; Overall Elo is fitted from Overall comparisons rather than averaged from axis ratings. Pooled ratings combine Small and Big GDDs. Uncertainty is estimated with 2,000 GDD-level bootstrap resamples, preserving the Small and Big composition and keeping all comparisons from each sampled GDD together.

Results. Figure S11 combines the empirical win-rate matrices and model-level Elo summaries for the six proprietary agents. GPT-5.5 (OpenAI, 2026a) has a win rate of 76.5% against GPT-5.6-Sol (OpenAI, 2026b) in source-code evaluation and 61.0% in Playtest, compared with 12.9% in scenario-based replay assessment. Their relative Elo ordering follows the same directions, while Claude-Fable-5.1 has the highest pooled Elo on all four axes. Figures S12 and S13 provide split-specific comparisons. Mean fidelity measures the level of specification fulfillment. Specifically, win rates and Elo summarize comparative performance across GDDs, with Elo aggregating outcomes against all opponents.

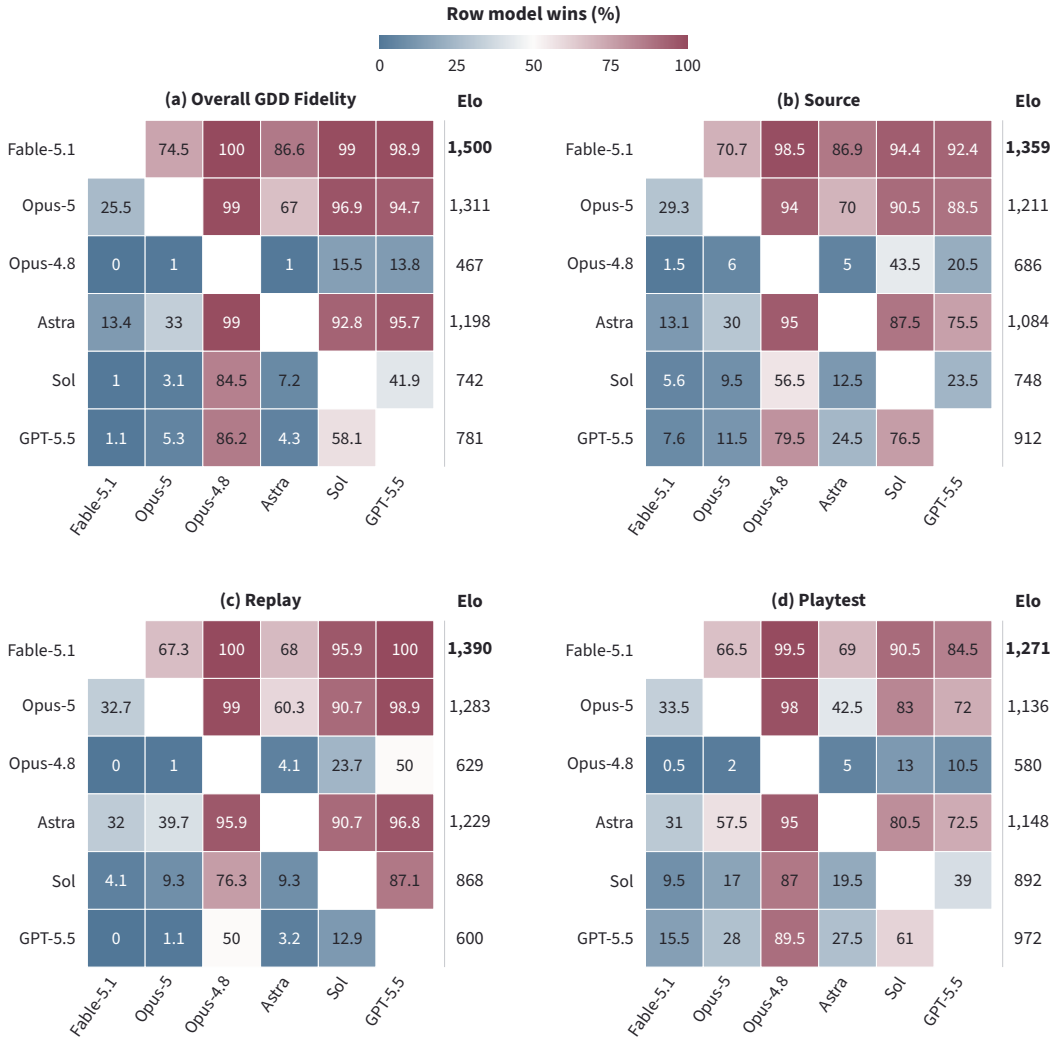


Figure S11: **Pairwise model comparisons and Elo ratings: Pooled GDDs.** Matrix entries give the row model's win rate against each column model, with ties counted as one half. The rightmost column reports the model's Elo rating for that panel. Values above 50% favor the row model; Elo ratings are relative and centered at 1,000 within each panel.

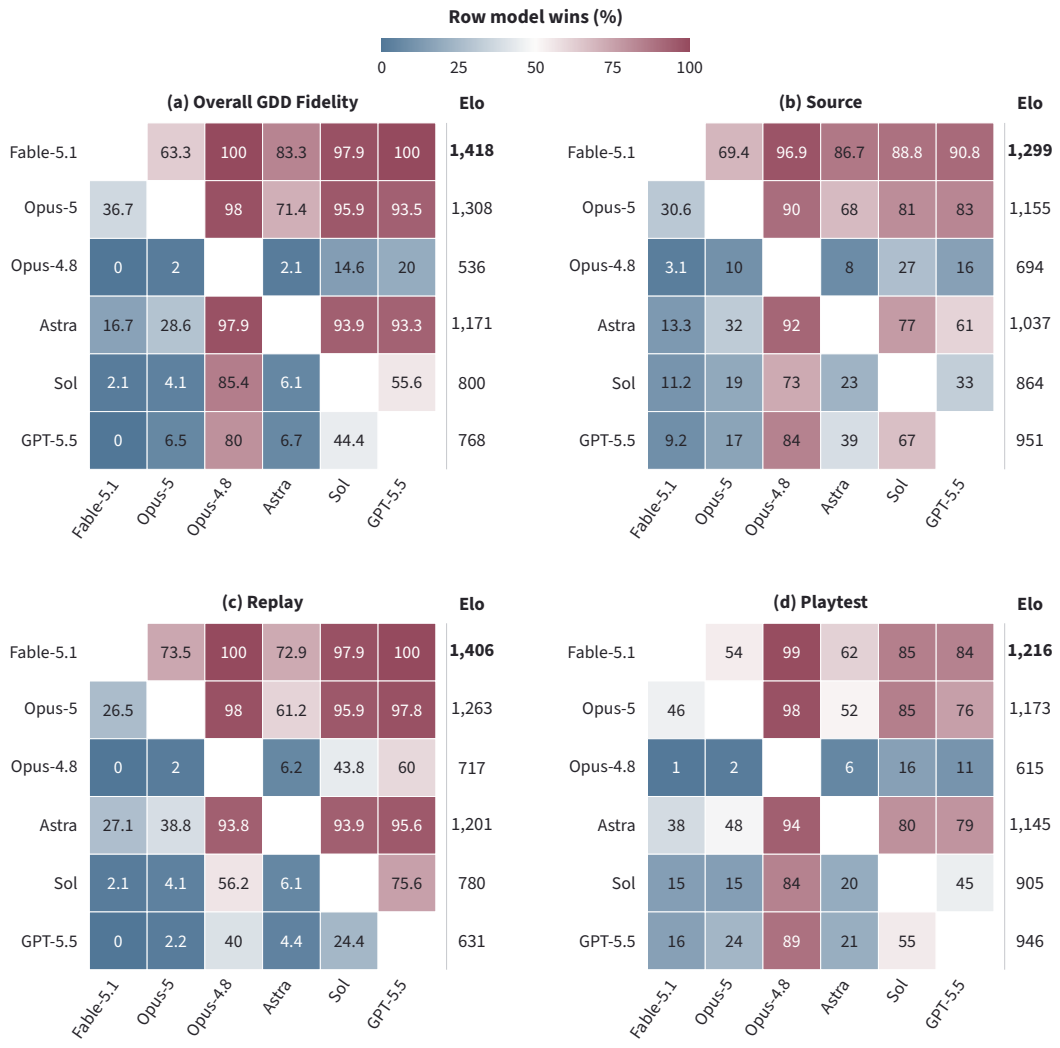


Figure S12: **Pairwise model comparisons and Elo ratings on *Small* GDDs.** The comparison protocol, relative model order, and win-rate color scale follow Figure S11. Each panel includes an Elo column alongside the empirical pairwise matrix.

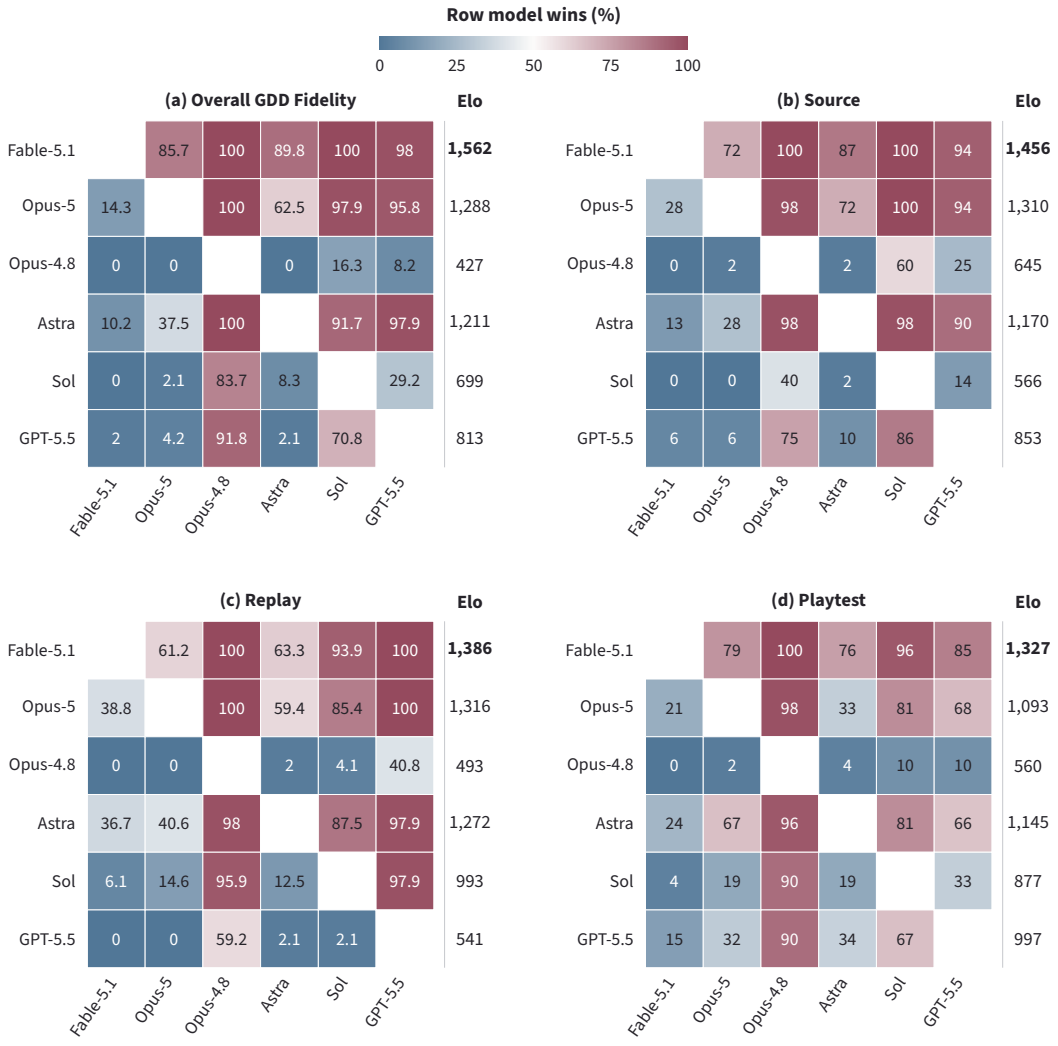


Figure S13: **Pairwise model comparisons and Elo ratings on *Big* GDDs.** The comparison protocol, relative model order, and win-rate color scale follow Figure S11. Each panel includes an Elo column alongside the empirical pairwise matrix.

F Ablations on Evaluation

This section examines choices in the evaluation that the method does not force: whether a judge without a contract keeps a stable requirement set, what dependency weighting changes in the source score, how the replay judge's model and reasoning effort shift its scores, and what adversarial initialization adds over repeated normal play. Each ablation modifies a single factor while keeping other inputs constant, such as builds, scenario replays, rubrics, or traces, and results are read against the self-disagreement of the same judge.

F.1 Source-Code Evaluation

We examine the source-code evaluation in Section 3.3.1 through repeated judgments, changes in judge configuration, and an ablation of dependency-weighted scoring.

F.1.1 Requirement-Set Consistency

The naive GDD-based judge identifies its own requirements from the GDD and source project, while the contract-based judge scores a fixed set of rules and invariants. Only the naive judge reconstructs its targets on each run. We therefore test whether repeated judgments of an unchanged build use a consistent set of requirements.

We run the naive judge five times per unchanged build for all 100 GDDs, using GPT-5.6-Sol at high reasoning effort. We also repeat dependency-aware contract construction five times for each GDD. After the consistency checks pass, the first generated contract is frozen for benchmark evaluation across builds and revision rounds, as described in Section B. Scores vary little: the median gap between a build's highest and lowest scores is 0.05. However, the extracted requirements vary substantially: for the median GDD, item counts vary by 60% of their mean, only 35% of items match across runs at a Jaccard threshold of 0.5, and 9% appear in all five runs. This instability is greater for *Big* GDDs: the median cross-run match is 31%, with 7% of items appearing in all five runs, compared with 38% and 14% for *Small* GDDs. Figure S14 shows requirement-count variation and cross-run item alignment. Thus, a stable score can mask shifting evaluation criteria, undermining requirement-level comparisons. Freezing the contract keeps targets fixed across agents and revision rounds; Section B.5 examines how consistently the contract itself can be constructed.

On the same build g_i, the naive score correlates more strongly with the contract's rule component F_i^{rule} than with its invariant component F_i^{inv}: Pearson correlations of 0.54 versus 0.16 for *Small* GDDs, and 0.87 versus 0.23 for *Big* GDDs. This limited correspondence with invariant pass rates motivates checking invariants explicitly alongside rules in the source-code measure, as in Equation 4.

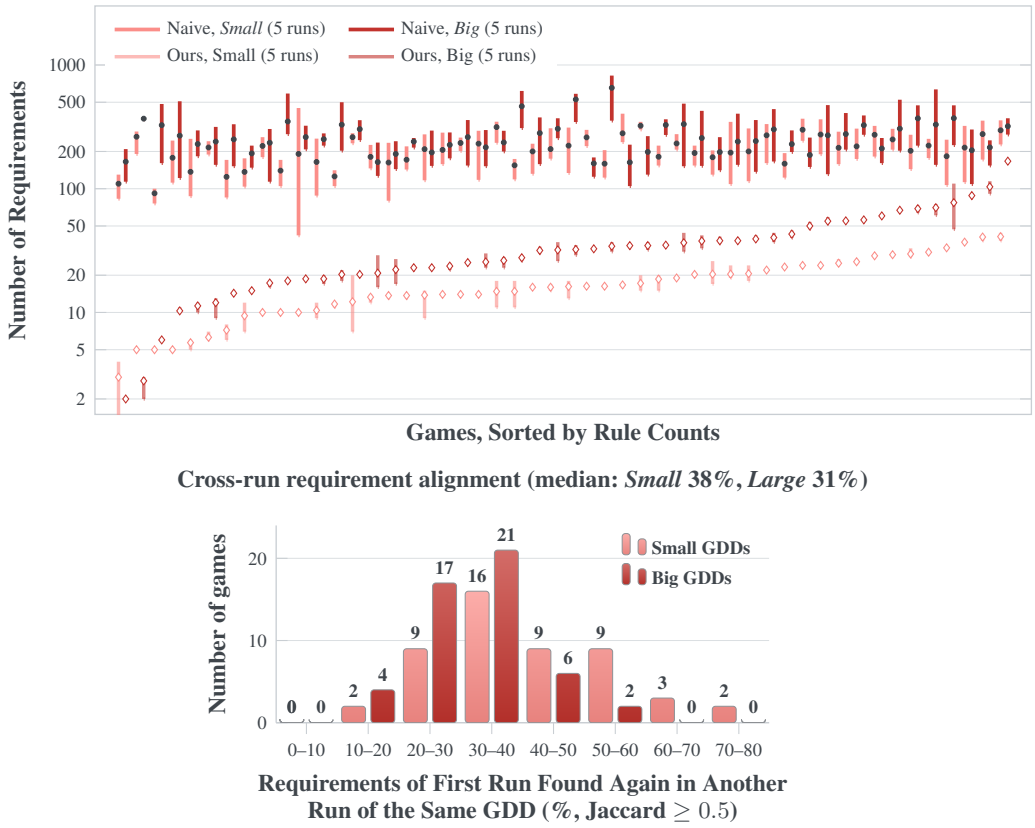


Figure S14: **Requirement consistency across repeated source-code judgments.** The naive judge evaluates each of 100 unchanged builds five times. Top: the minimum, maximum, and mean number of extracted requirements per game, with rule counts shown for comparison. Bottom: the distribution of the fraction of one run's requirements matched in another run at a Jaccard threshold of 0.5.

F.1.2 Ablation On Dependency-Weighted Scoring

Table 3(a) measures the sensitivity of the full source-code score F_i^{src} to dependency weighting. We hold the rule scores and invariant judgments fixed and compare $\alpha = 0.5$ and $\alpha = 0.1$ with uniform weighting ($\alpha = 1$), reporting the mean and maximum absolute score changes. The analysis below separately examines changes in the rule component F_i^{rule} for builds with and without confirmed high-reach failures. Weighting rules by their downstream reach, as defined in Equation 3, gives greater influence to mechanics on which other behaviors depend. We compare the dependency-weighted rule score F_i^{rule} at $\alpha = 0.5$ against its unweighted counterpart at $\alpha = 1$, and denote the difference by $\Delta_\alpha = F_i^{\text{rule}}(\alpha) - F_i^{\text{rule}}(1)$. A negative gap means that weighting lowers the score. This is expected when rules with greater downstream reach receive relatively low implementation scores; conversely, the score can increase when failures concentrate on lower-weight leaf rules.

To examine this effect, we classify a game as *root-defective* when a core root rule, whose downstream reach is at least half the maximum in the game's contract, receives a score of at most 0.5 under both source-code judge settings and is violated in the adaptive playtest. The comparison group contains games with no observed playtest violation among their assessed root rules. Both groups include *Small* and *Big* games. As shown in Figure S16, negative gaps occur in 11 of 18 root-defective games, compared with 6 of 16 comparison games. Thus, dependency weighting lowers the score more often in games with confirmed foundational failures. Positive gaps in the root-defective group arise primarily when failures are even more prevalent among leaf rules, which pull down the unweighted average more strongly. The sign consequently reflects the distribution of scores across the contract, rather than the presence of a root failure alone. The gaps are small relative to judge variability, so we interpret the group difference as a tendency in the observed scores. This tendency is

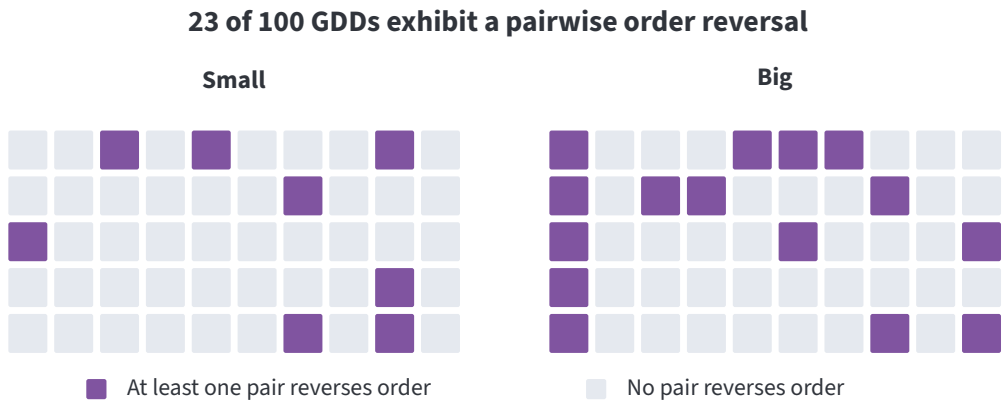


Figure S15: **Dependency weighting changes game-level comparisons.** Each cell represents one GDD. Purple cells indicate a strict reversal in the source-score ordering of at least one pair of the six proprietary agents between uniform weighting ($\alpha = 1$) and downstream-reach weighting ($\alpha = 0$), with rule judgments and invariant scores fixed. Such reversals occur on 23 of the 100 GDDs.

consistent with the intended role of dependency weighting: giving greater influence to rules with broader downstream effects.

Furthermore, we examine how dependency weighting affects comparisons on individual GDDs. For the six proprietary agents in Appendix E, we hold the rule judgments and invariant scores fixed and compare source fidelity under uniform weighting ($\alpha = 1$) and downstream-reach weighting ($\alpha = 0$). As shown in Figure S15, the ordering of at least one agent pair reverses on 23 of the 100 GDDs. The same rule-level judgments can therefore favor different agents when greater influence is assigned to rules on which other requirements depend.

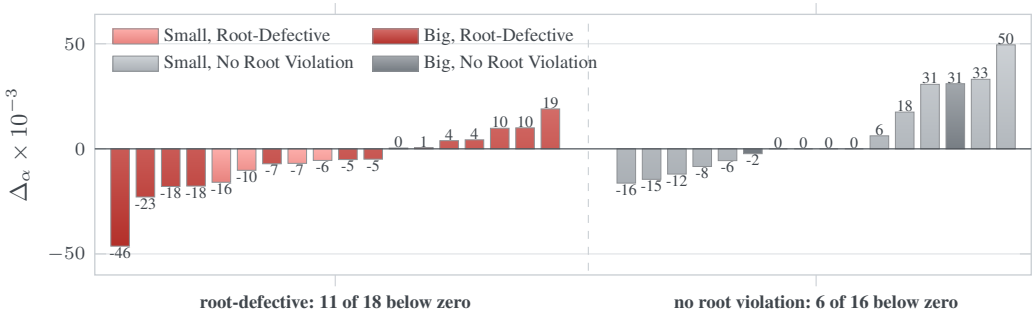


Figure S16: **Effect of dependency weighting on rule scores.** Each bar is $\Delta_\alpha = F_i^{\text{rule}}(\alpha) - F_i^{\text{rule}}(1)$ at $\alpha = 0.5$. Left: root-defective games meeting the joint source-code and playtest criteria. Right: comparison games without an observed playtest violation among their assessed root rules. Negative values indicate lower weighted scores; the sign is unchanged at $\alpha = 0$.

F.2 Scenario-Based Replay Assessment

Table S11 evaluates GPT-5.6-Sol and GPT-5.6-Luna at four reasoning efforts on the same six builds, scenario replays, and rubrics, against the original sol extra-high judgments; item agreement is the Pearson correlation between the per-item scores of ρ_i^{vis} and those judgments, with a re-run agreement of about 0.86. At extra-high effort, substituting luna for sol changes the mean F_i^{replay} by -0.005 at an agreement of 0.85. Lowering the effort increases mean scores relative to the reference, though not equally: sol rises by $+0.03$ to $+0.04$ at each lower level with unchanged agreement, whereas luna stays at $+0.006$ at high and rises to $+0.035$ at medium and $+0.081$ at low, with agreement falling to 0.79, mostly on the visual and art items. We therefore use luna at high effort: its bias is smaller than sol's at any effort below extra-high, its agreement is close to the re-run agreement, and a judgment

takes 41 seconds at about $0.007 against 78 seconds and $0.048 for the reference, a 6.7× reduction that brings the replay score of a game (three judgments over 13–24 scenario replays) from about $2–3.5 to $0.3–0.5.

Table S11: **Agent-as-a-judge on six builds with identical scenario replays and rubric.** Item r is the Pearson correlation of rubric-item scores with the original sol extra-high judgments of the same builds (re-run agreement ≈ 0.86); Δ is the mean reward difference from them. Cost is per judge call at list prices. sol = GPT-5.6-Sol, luna = GPT-5.6-Luna.

Judge, effort	Item r	Mean reward	Δ vs. ref.	s / call	$ / call
sol, xhigh (reference)	–	0.639	–	78	0.048
sol, high	0.88	0.667	+0.027	71	0.034
sol, medium	0.88	0.669	+0.030	63	0.034
sol, low	0.89	0.680	+0.041	52	0.045
luna, xhigh	0.85	0.634	−0.005	63	0.0074
luna, high	0.85	0.645	+0.006	41	0.0072
luna, medium	0.83	0.674	+0.035	27	0.0070
luna, low	0.79	0.720	+0.081	21	0.0070

F.3 Adaptive Playtest

Table 3(c) pairs each game's normal and final judgments to show what adversarial testing adds. Following Section 3.3.3, normal satisfaction and violation verdicts are kept and adversarial evidence is used only for unverified rules, so with $n_i = |\mathcal{R}_i^{\text{rule}}|$ and $n_i^{\text{sat}}, n_i^{\text{vio}}$ rules finally satisfied and violated, the score $F_i^{\text{adapt}} = n_i^{\text{sat}}/n_i$ and the coverage $(n_i^{\text{sat}} + n_i^{\text{vio}})/n_i$ keep every rule in the denominator, and the increase reflects added evidence on unchanged builds.

To separate prerequisite initialization from merely running another test, we compare the modes on 86 dependency-linked rules from 25 *Small* and 25 *Big* games, with the build, contract, Test Policy, interface API, and budgets held fixed and two seeds per target; initialization supplies $z_{i,r} \models \phi_r$, never target outcomes.

Table S12 reports a first normal run and a second pass that re-tests only its unverified targets, once normal and once adversarial, with the adversarial mode alone as reference. Each episode takes the majority of three GPT-5.6-Luna judgments at high effort, and coverage is averaged over targets and seeds, then games and splits. A second normal test increases coverage from 40.5% to 41.0%, whereas an adversarial second test increases it to 48.5%. The combined setting also exceeds the 45.0% coverage of adversarial testing alone, supporting their complementary use: normal play provides evidence along ordinary gameplay paths, while initialization enables additional downstream checks.

Table S12: **Controlled playtest comparison.** Mean judgment coverage (%) on 86 target rules from 25 *Small* and 25 *Big* games. A second pass keeps the first pass's conclusive verdicts and re-tests only the targets it left unverified.

First pass	Second pass	Coverage	Δ
Normal	–	40.5	–
Normal	Normal	41.0	+0.5
Normal	Adversarial	48.5	+8.0
Adversarial	–	45.0	–

F.4 Complementarity of Evaluation

Code Implementation and Runtime Gap. Source-code inspection and playtesting provide different evidence for the same rule. Figure S17 shows three rules that receive a source-code item score

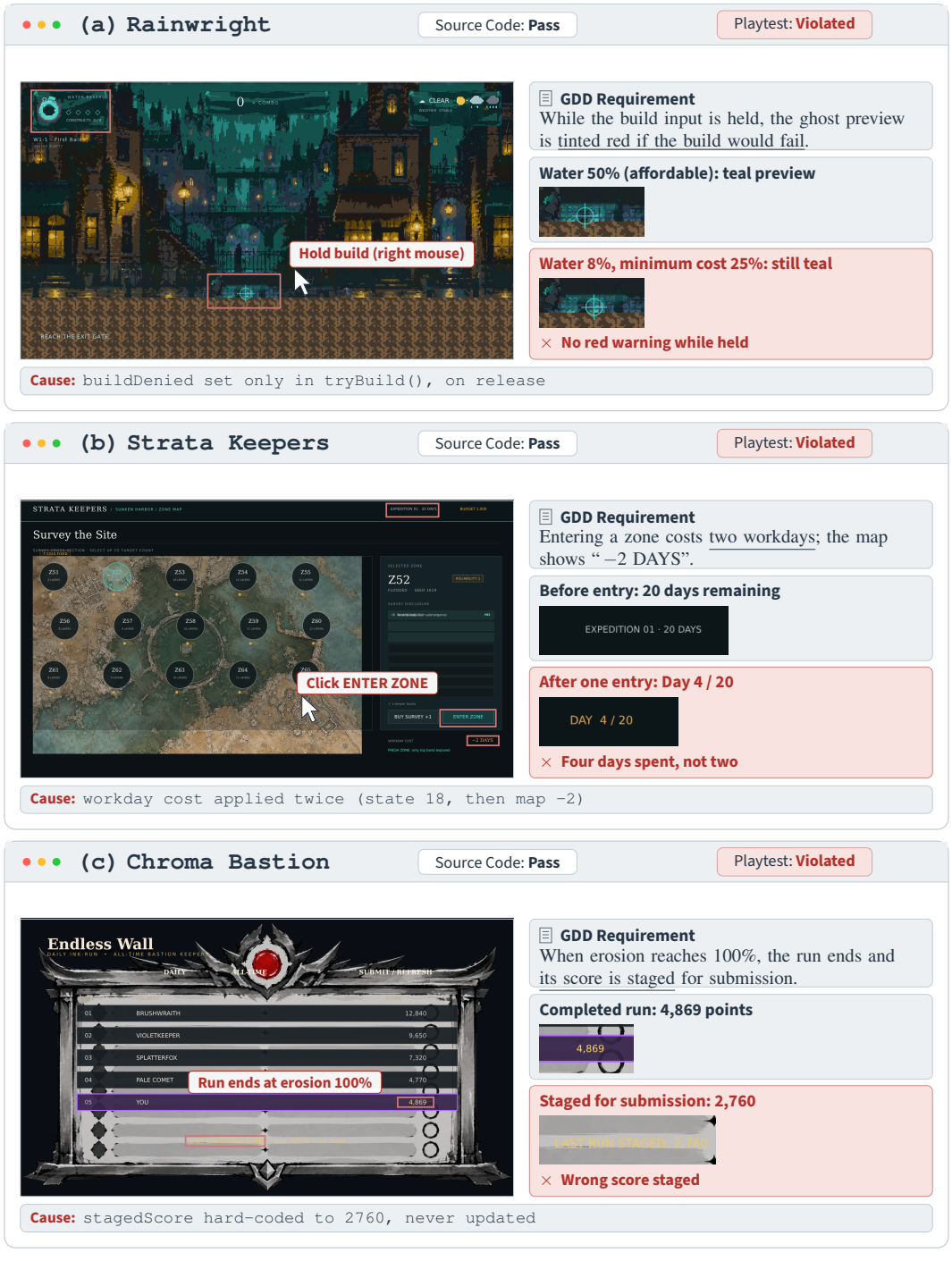


Figure S17: **Full source-code credit with a runtime violation.** Each rule passes the source-code judge (item score 1.00) and is judged *violated* in the playtest. (a) `Rainwright`: with 8% water, below the 25% minimum cost, the held build preview stays teal instead of turning red, because the denial flag is set only on release. (b) `Strata Keepers`: a two-day zone entry consumes four days because the cost is applied twice. (c) `Chroma Bastion`: the finished run scores 4,869 but a constant 2,760 is staged for submission. Screenshots are from asset-integrated builds whose relevant logic is unchanged from the evaluated source; boxes, callouts, and crops are annotations.

of 1.00 but are judged *violated* during playtesting. In these cases, the recorded source-code judgments credit relevant implementation logic, while execution reveals problems in its integration with the rest of the game. In *Rainwright*, the preview turns red when `buildDenied` is set, but the flag is updated only in `tryBuild()` on input release, so the preview remains teal while the input is held. In *Strata Keepers*, the map handler deducts two workdays after excavation setup has already reduced the remaining days from 20 to 18, producing a four-day cost. In *Chroma Bastion*, the run-end path stages a score initialized to 2,760 without replacing it with the completed run's score. These examples expose integration failures missed by the recorded source-code judgments: a condition checked at the wrong time, a duplicated effect, and an outdated value. Linking both forms of evidence to the same requirements makes these discrepancies identifiable and provides concrete targets for revision.

Axis Omission. We compare the full three-axis scores with scores recomputed after omitting each axis, using all 100 initial GPT-5.6-Sol builds, comprising 50 *Small* and 50 *Big* games. The full score is the arithmetic mean of the three axes, and each omitted-axis score is the arithmetic mean of the remaining two axes. Reversed orderings are counted over all unordered pairs of games within the same split, giving $2\binom{50}{2} = 2{,}450$ comparisons. A pair counts as reversed only when its score ordering strictly changes direction; ties in either ordering remain in the denominator but are not counted as reversals.

G Additional Analysis on Evaluation

G.1 Dependency-Aware Contract

G.1.1 Rule Coverage and Run-to-Run Consistency

We compare the requirement sets produced by naive judgment and dependency-aware contract construction on all 100 GDDs. The naive judge identifies requirements while evaluating a generated game, whereas the contract is constructed from the GDD alone and represents state constraints as invariants and conditional behaviors as rules, following Appendix B. Both methods use GPT-5.6-Sol and require a supporting GDD quotation for every extracted item.

For each GDD, we construct a shared pooled reference set from the rules referenced across five runs of each method, consolidating repeated references to the same rule. Rule coverage is the fraction of rules in this shared set that are referenced by a run's extracted evaluation items, with each reference rule counted at most once. We average coverage over the five runs for each GDD. Table S13 reports the split-level means: the contract increases mean coverage from 0.84 to 0.88 on *Small* and from 0.62 to 0.80 on *Big*. These values measure coverage relative to the rules recovered by repeated extraction.

Citation overlap measures the mean pairwise Jaccard similarity between the sets of referenced requirements across runs of the same GDD. On *Big*, it is 57% for the naive judge and 93% for repeated contract extractions. Figure S18 summarizes rule coverage and citation overlap across all 100 GDDs. Among naive items that can be matched across runs, judgment status agrees in 91% of pairs on *Small* and 87% on *Big*, indicating that much of the instability concerns which requirements are selected for evaluation.

Table S13: **Coverage against a pooled reference set.** Fraction of a shared per-GDD reference set cited by each method's extracted items, reported as mean (median) across each split. The reference set pools rules referenced across five runs of both methods; per-GDD coverage is averaged over five runs. The contract is constructed from the GDD alone, whereas the naive judge sees both the GDD and the generated game.

GDD Split	Naive Judge	Dependency-Aware Contract (Ours)
Small	0.84 (0.92)	**0.88** (0.90)
Big	0.62 (0.63)	**0.80** (0.92)

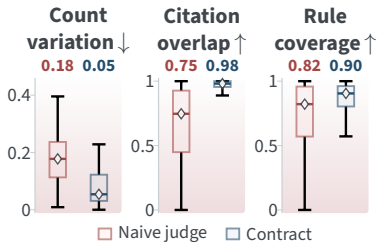


Figure S18: **Requirement set stability.** Across 100 GDDs, the contract yields more stable requirement sets than naive GDD-based judgment and higher citation overlap. Boxes show interquartile ranges; diamonds and labels mark medians across GDDs.

G.1.2 Comparison With Enumerated Evaluation

We compare contract-based source evaluation with an *enumerated* baseline, which scores individual GDD-derived requirements without explicitly representing their dependencies. This baseline evaluates a list of GDD-derived requirements, each recording a condition, an action, and an expected outcome, without explicit dependency edges between requirements. Supporting GDD quotations provide the correspondence between these requirements and contract rules. Using one initial GPT-5.6-Sol build for each of the 100 GDDs (50 *Small*, 50 *Big*), we analyze additional inspection targets, their coverage of recorded playtest violations, and implementation omissions identified by comparing against contract-based source evaluation.

Enumerated Source Judgments. We hold the builds, enumerated source judgments, and rule-to-requirements mapping fixed. Averaging the mapped requirements scores provides source scores for 2,589 of the 2,729 frozen contract rules in Section B.5. We use the same attribute-level links from state updates to conditions as the source scorer (Section 3.3.1). Full credit here means 1.0, whereas

the path analysis in Section B.7 uses a confirmed-pass threshold of 0.8. A full-credit rule is eligible if it has at least one direct predecessor and all predecessor scores are available. We select it for further inspection if any predecessor scores are below 1.0.

This source-only analysis identifies 408 of 908 eligible targets with full enumerated credit (44.9%) in 53 of the 100 games (Table S14). The share is 25.7% in *Small* and 58.6% in *Big*, with targets in 17 and 36 games, respectively. Of these, 365 targets across 47 games have a predecessor score of 0.5 or lower.

Table S14: **Dependency context identifies additional source-code inspection targets.** A target has full enumerated source credit and a lower-scored direct predecessor. Percentages are pooled over eligible targets; the enumerated judgments remain fixed.

Measure	*Small*	*Big*	**All**
Games	50	50	100
Full-credit rules	738	896	1,634
Eligible full-credit targets	377	531	908
Additional inspection targets	97	311	408
Games with additional targets	17	36	53
Additional targets / eligible targets	25.7%	58.6%	44.9%
Targets with a predecessor score ≤ 0.5	60	305	365

Deduplicating targets with identical mapped requirements sets within each game leaves 361 distinct sets. Excluding targets that share any requirements with a predecessor still leaves 338 targets across 49 games: 80 in 16 *Small* games and 258 in 33 *Big* games.

Runtime Coverage. We match enumerated source scores with conclusive adaptive-playtest verdicts for 2,185 targets, of which 560 are violated. A target is violated if any mapped requirement is violated, and satisfied if all are satisfied. The enumerated baseline inspects rules with mapped source scores below 1.0; adding dependency context includes the selected full-credit targets. Coverage is the fraction of recorded violated targets included in each inspection set.

Table S15: **Coverage of recorded playtest violations through source-guided inspection.** Enumerated selects targets with mapped source scores below 1.0; adding dependencies includes the selected full-credit targets. Both keep enumerated judgments fixed. Coverage uses recorded playtest violations; inspection-set sizes count all selected targets among the 2,185 matched targets.

	Violation Coverage (%)		**Added Violations**	**Inspection-Set Size**	
Split	**Enumerated**	**+ Dependencies**	**Rules (Games)**	**Enumerated**	**+ Dependencies**
Small	16.7	24.2	5 (3)	49	140
Big	78.3	87.7	46 (23)	722	974
All	71.1	80.2	51 (26)	771	1,114

Dependency context increases coverage from 398/560 (71.1%) to 449/560 (80.2%), adding 51 violated targets across 26 games to the inspection set (Table S15). These targets comprise 45 distinct game–requirement sets and account for 31.5% of the 162 recorded violations outside the enumerated inspection set. The gain is 9.1 percentage points, with a 95% paired game bootstrap interval of 6.1–13.0 points (20,000 resamples stratified by split). Dependency context thus directs inspection to execution failures that received full enumerated source credit.

Source-Judgment Comparison. We compare archived enumerated (GPT-5.6-Luna) and contract-based (GPT-5.6-Sol) source judgments on identical source revisions, matching requirements through their cited GDD requirements. Among 2,446 rules with paired scores, 244 rules across 68 games receive full enumerated credit but less than full contract-based credit. The reverse occurs for 375 rules across 62 games, yielding 619 disagreements in full-credit status.

A retrospective assistant review checks all 619 disagreements against the original requirements, both judgment rationales, and the source code. It identifies implementation omissions in 17 rules across 13 *Big* games that received full enumerated credit, corresponding to 15 distinct mapped requirement sets. Sixteen rules across 12 games concern conditions, state propagation, transitions, persistence, or ordering. The following examples illustrate how these connections reveal missing behavior.

In *Grand Atelier 2D*, rules `R-CL2` and `R-CL3` require a submitted collection with insufficient grades or inconsistent composition to fail with a reputation penalty of 12. The enumerated judgments credit the grade and coherence checks. However, `toggleGarment` enables submission only for passing collections, and `submitCollection` returns otherwise. These checks block the required failure transition, leaving both the failure branch and its penalty unimplemented (`scenes/AtelierScene.ts`, lines 412–415).

In *Last Announcement*, rule `RT-CAPTURE-1` requires chase success to take priority over capture, remove the threat, and update the retry checkpoint. The enumerated judgment credits the success-before-capture ordering. The `chaseSuccess` handler removes the threat and records completion but never updates `retryCheckpoint`, omitting the required link between encounter completion and retry state (`scenes/GameScene.ts`, lines 222–227 and 442–451).

In *Whispers of the Wild*, rule `L1` requires a qualifying landscape photo to receive a bonus and set the species' `landscapeBadge`. The code calculates the condition and bonus but updates the badge only when the photo also improves the species' best base score. A qualifying photo below that record therefore misses the required badge (`scenes/FieldScene.ts`, lines 451–455).

In *Rainwright*, rule `R5#4` requires an overlapping pillar to move to the nearest free adjacent tile. The enumerated judgment credits a one-tile offset. The code shifts the pillar without checking whether the destination is free, so an occupied adjacent tile produces another overlap (`scenes/PlayScene.ts`, lines 844–853).

Tracing each condition through its required effects identifies the missing branch, state update, or constraint check and its code location. This produces concrete guidance for the contract-based revision setting in Section 4.4.

G.2 Cost Analysis

G.2.1 Source Code Evaluation

Table S16: **Agreement and cost across source-code judge configurations.** Rule-level correlation and invariant agreement are measured against a fixed set of reference judgments from GPT-5.6-Sol at extra-high reasoning effort. The first row reports an independent re-run of that configuration. Mean Rule Score reports the average rule implementation score in each run. Relative cost is estimated from token usage per call and the prices used in the experiment, normalized to the first row.

Judge	Effort	Rule-Level Correlation r	Invariant Agreement	Mean Rule Score	Tokens/Call (10^3)	Relative Cost
GPT-5.6-Sol	Extra high	0.74	0.83	0.76	213	1.00
	High	0.70	0.83	0.76	214	1.00
	Medium	0.75	0.88	0.78	176	0.83
	Low	0.80	0.86	0.79	160	0.75
GPT-5.6-Luna	Extra high	0.72	0.77	0.79	198	0.15
	High	0.72	0.76	0.79	187	0.15
	Medium	0.69	0.75	0.80	169	0.13
	Low	0.71	0.73	0.77	151	0.12

We examine how judge model and reasoning effort affect source-code evaluation. We hold the contracts and builds fixed and compare seven configurations against fixed reference judgments produced by GPT-5.6-Sol at extra-high reasoning effort. An independent re-run of GPT-5.6-Sol at extra-high effort yields a rule-level Pearson correlation of 0.74 with those reference judgments. This re-run provides a baseline for interpreting agreement across configurations.

As shown in Table S16, the seven alternative configurations have rule-level correlations from 0.69 to 0.80 with the reference judgments, comparable in magnitude to the 0.74 of the independent re-run. Mean rule scores $f^{\mathrm{src}}_{i,r}$ range from 0.76 to 0.80 across all eight runs.

The benchmark's default configuration is GPT-5.6-Luna at high effort, as described in Section 4.1. It has a rule-level correlation of 0.72 at an estimated cost of $0.15\times$ the independent sol re-run; at low effort, luna has a correlation of 0.71 at $0.12\times$. Across luna settings, invariant agreement ranges from 0.73 to 0.77, against 0.83 for the independent sol re-run. Luna's rule-level agreement is therefore comparable in magnitude to the re-run baseline at substantially lower estimated cost.

G.2.2 Replay-Based Evaluation

The number of image tokens increases with frame width, as shown on the left of Figure S19. A native-width frame uses about 4.7K image tokens, compared with 2.0K at 1280 pixels and 0.5K at 640 pixels. These values exclude the fixed prompt of roughly 39K tokens per call, one call per scenario replay $\xi_{i,q}$. Across six *Small* and *Big* games, evaluation at 640 pixels agrees closely with evaluation at native resolution, with a Pearson correlation of 0.967 between the per-item scores of ρ^{vis}_i. Each call at 640 pixels uses approximately 45K tokens. Reducing the frame width from 640 to 480 pixels lowers image tokens per frame from 0.5K to 0.36K, a 28% reduction. When both image dimensions scale proportionally, this is smaller than the 44% reduction in pixel count. Total tokens per call decrease little because the fixed prompt accounts for most of the usage. At 480 pixels, the per-item correlation with the 640-pixel setting drops to 0.921. As shown in Figure S20, the word "space" becomes increasingly fragmented and difficult to read at lower resolutions.

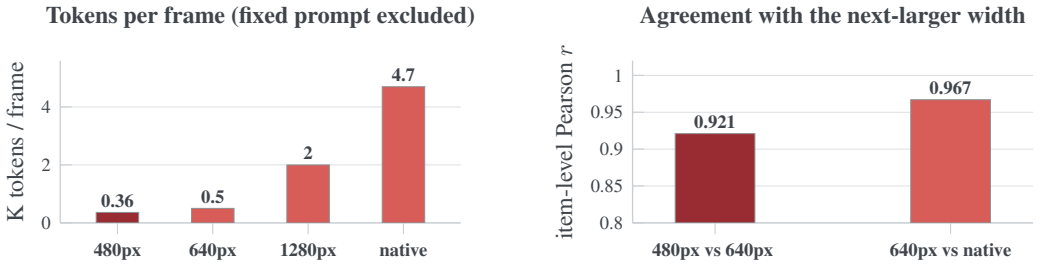


Figure S19: **Tokens vs.frame width**. Left: tokens per frame, fixed prompt excluded. Right: item-level agreement with the next-larger width.

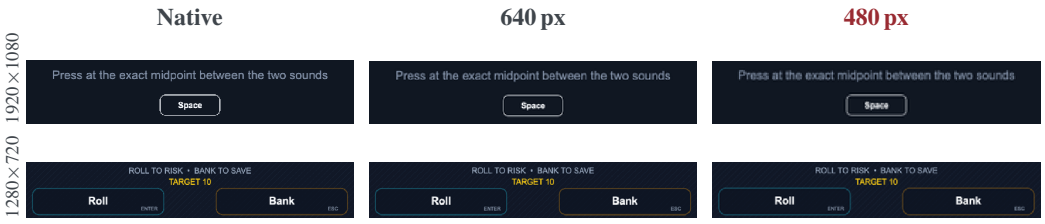


Figure S20: **Example of sampled frame across different resolutions.** The same frame region as the judge receives it at native resolution, 640 px and 480 px (Lanczos downscaling as in the pipeline). Top: 1920×1080 game; bottom: 1280×720 game.

G.2.3 Adaptive Playtest

Both the bot author A^{play} and the trace judge were developed with GPT-5.6-Sol; our default evaluation uses GPT-5.6-Luna at high reasoning effort. On fixed traces (Table S17), Luna achieves rule-level agreement of 0.910 on Small and 0.888 on Big against the Sol high reference, compared with 0.959 on both splits for an independent GPT-5-6-Sol at high reasoning effort's re-run. Flips that change $f^{\mathrm{adapt}}_{i,r}$ occur on 6 of 222 Small rules and 3 of 242 Big rules, compared with 3 and 1, respectively, for the Sol re-run. Token usage per vote decreases from 516K–688K to 51K–71K.

For the bot author, judgment coverage (Section D.3) differs by 0.15–0.17 between Sol and Luna runs, compared with 0.11–0.16 between repeated runs of the same model. These ranges partially overlap, but do not establish equivalence between model substitution and re-running. With three judgments per rule retained, the reported per-game playtest cost decreases from $0.65–0.73 to $0.08–0.11.

Table S17: **Playtest judge comparison on fixed traces.** Entries report rule-level agreement with a GPT-5.6-Sol high reference run and the number of rules (*Small*: 222 rules; *Big*: 242 rules), with three judgments per rule. An independent Sol high re-run provides a reference for run-to-run variation.

Judge, effort	Small	Big	Tokens / vote
Sol, xhigh	0.959 / 2	0.930 / 4	485–753K
Sol, high (re-run)	0.959 / 3	0.959 / 1	516–688K
Sol, low	0.892 / 7	0.897 / 3	217–315K
Luna, xhigh	0.919 / 3	0.893 / 1	57–86K
Luna, high	0.910 / 6	0.888 / 3	51–71K
Luna, low	0.833 / 9	0.810 / 10	46–64K

G.3 Fixed-Rate Sampling vs. Adaptive Frame Selection

Experimental Setup. The comparison in Table 3(b) uses games built by GPT-5.6-Sol on all 50 *Small* GDDs, comprising 244 scenario replays and 4,102 paired rubric-item assessments. Both conditions use the same recording and visual rubric, and the number of original-resolution frames matches for each replay (2,268 per condition in total). Two games, `Bin Bit` and `Same Or Shift`, are also used in preliminary diagnostics.

Selection and Judging. Adaptive selection starts from the first and last frames and uses the input trace, scenario entry condition, and visual rubric to request additional timestamps. A deterministic tool retrieves the first captured frame at or after each requested time, allowing subsequent requests to use the observed frames. All inspected frames count toward the matched frame budget and are passed to the final judge. The selector and final judge use GPT-5.6-Sol with medium reasoning effort in separate sessions; the final judge receives only the selected frames and rubric, without the selector's reasoning or scenario-entry prompt.

Audit and Metrics. A separate GPT-5.6-Sol session with high reasoning effort compares how well the cited frames support the two rationales for each rubric item. It selects the better-supported rationale or assigns an *equivalent* or *insufficient-evidence* outcome. Method names and frame identifiers are masked, with each method appearing first in 122 scenarios, although image paths retain source directory names. This audit measures rationale support rather than accuracy against reference labels. *Item preference* is each method's share of the 1,159 comparisons favoring either method, pooled across all rubric items and scenarios. Equivalent and insufficient-evidence outcomes are excluded from this denominator. *Game wins* counts games in which a method receives more item-level preferences across its scenarios; equal totals are ties. Adaptive selection obtains 60.5% item preference and 35 game wins, compared with 39.5% and 13 for fixed-rate sampling, with two tied games. Table S18 includes the complete outcomes.

Table S18: **Rationale-Support Outcomes.** Counts and percentages over all 4,102 paired rubric-item assessments.

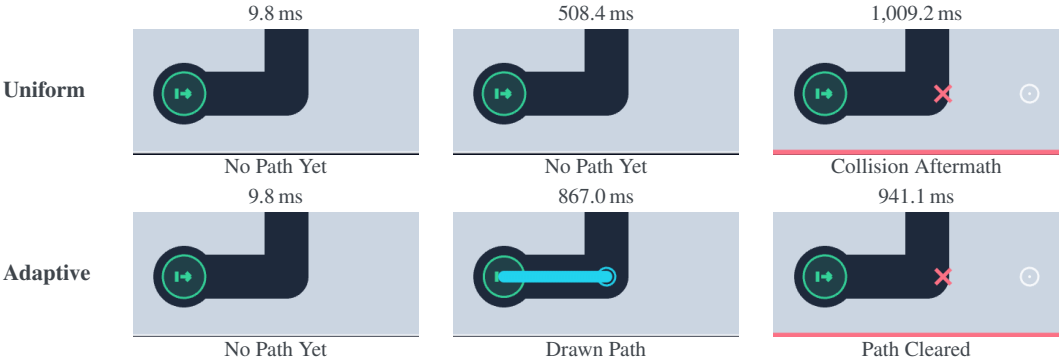
Audit Outcome	Assessments	Share (%)
Fixed-rate preferred	458	11.2
Adaptive preferred	701	17.1
Equivalent	2,848	69.4
Insufficient evidence	95	2.3
Total	4,102	100.0

Qualitative Examples. Figure S21 demonstrates that adaptive selection captures intermediate states that uniform sampling omits. In `Blackout Dispatch` (Figure S21(a)), the Confirm button is disabled during a drag, enabled after both tokens are assigned, and disabled following confirmation; uniform sampling fails to capture the enabled state. In `Chalk Escape` (Figure S21(b)), adaptive frames display a drawn path and its subsequent removal upon wall contact, while uniform frames only depict the pre-drawing state and the aftermath of the collision, omitting the path itself. These findings provide direct visual evidence for the specified transitions, rather than requiring inference from the final state alone.

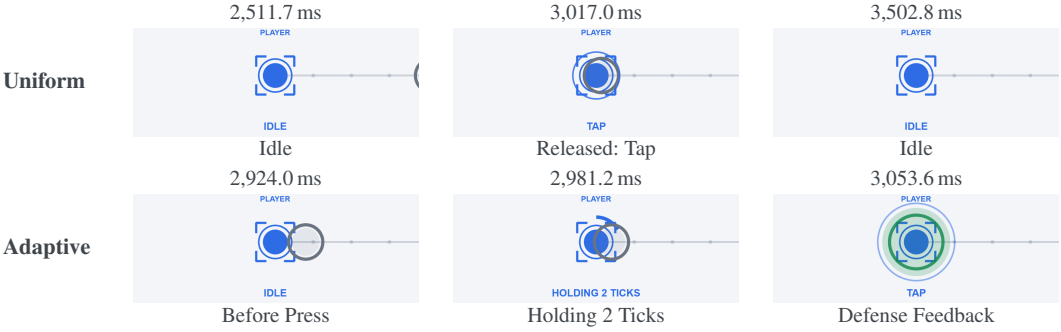
(a) `Blackout Dispatch`: Confirm Gating



(b) `Chalk Escape`: Path Removal



(c) `Rooftop Rumble`: Hold and Defense Feedback



(d) `Tile Merge`: Merge Animation

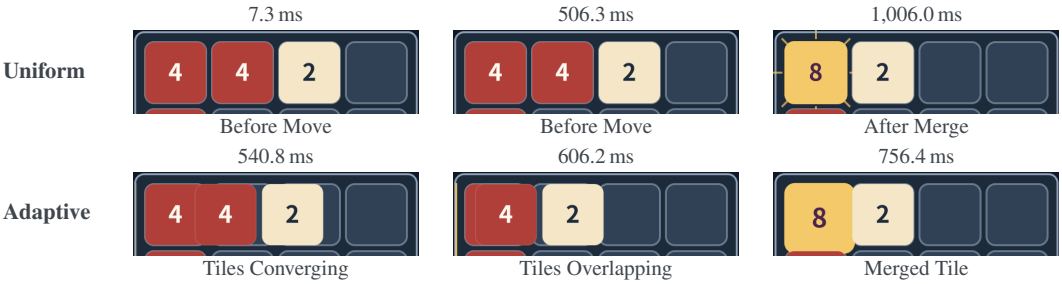


Figure S21: **Visual evidence from adaptive frame selection.** Each panel compares the same replay of a *Small* game. Adaptive selection captures (a) Confirm activation, (b) the path before removal, (c) holding and defense feedback, and (d) tile convergence and overlap. These cues are absent from the displayed uniform samples. Each row shows three cropped frames with timestamps.

G.4 GAMEPLAY AGENT VS. ADAPTIVE PLAYTEST

We compare the evaluation information provided by a gameplay agent from Orak (Park et al., 2026) and our adaptive playtesting with CODE-AS-POLICIES on ten *Big* games generated by GPT-5.6-Sol. The online agent uses GPT-5.6-Luna with high reasoning effort, three seeds, and at most 200 action steps per trial. It receives textual state observations and clickable controls, and is scored by game-specific milestone attainment. Our playtests instead return judgments linked to individual GDD requirements and their execution evidence.

`Toybox Rider` illustrates the distinction. All three online trials receive credit for reaching `Art Desk`; the implemented milestone checks whether the area or its speed stage has been reached. In a separate normal-play trace of the same build, our evaluator reports that the game enters Art Desk at 180.23 s but returns to Block Street at 182.98 s, violating the prescribed area sequence and duration. It also reports a moving obstacle at 7.37 s, despite the GDD excluding moving obstacles during the first 120 s of the tutorial. Neither condition is checked by the implemented milestone predicates.

Adversarial playtesting extends this process to requirements left unverified during normal play (Table S19). Across the same ten games, recorded judgment coverage—the fraction of requirements judged satisfied or violated, $(S + V)/N$—increases from 9.95% with normal play to 73.35% after adding adversarial results.

Table S19: **Recorded results for all ten games.** Gameplay agent progress is milestone attainment (mean $\pm$ sample SD over three seeds); Judgment coverage is $(S + V)/N$ over the archived GDD-derived requirements. The percentages use different target sets. Coverage is calculated from the saved judgments, with unverified requirements retained in the denominator.

	Gameplay Agent	**Adaptive Playtest**	**Judgment Coverage (%)**	
Game	**Progress (%)**	**Requirements N**	**Normal**	**Normal + Adv.**
`Abyssal Chain`	6.67 ± 0.00	110	6.36	89.1
`Alias Alchemy Shop`	13.33 ± 0.00	133	12.0	91.0
`Beat Reroute`	26.67 ± 0.00	84	19.05	86.9
`Chameleon Slide`	13.33 ± 0.00	87	0.0	67.8
`Fogfall Delivery`	15.56 ± 10.18	119	5.0	88.2
`Night Shift Rx`	23.08 ± 0.00	96	8.3	87.5
`Today's Best Spot`	35.71 ± 0.00	103	5.8	62.1
`Toybox Rider`	40.00 ± 0.00	112	20.5	88.4
`ToyFit`	0.00 ± 0.00	100	1.00	40.0
`Whispers of the Wild`	42.22 ± 10.18	117	21.4	32.5

Compared with the implemented gameplay-agent milestone scores, our benchmark provides a more specific account of implementation errors: it names the violated requirement and links it to execution evidence. That information gives a revision agent a concrete behavior to correct.

H HUMAN CALIBRATION

We compare scenario-based replay score, adaptive playtest score, and overall GDD Fidelity with the same mean overall expert rating using Spearman correlation. While A2Z GameSpec-Bench evaluates source-code compliance directly, manual source-code inspection by human experts is excluded from this calibration. Evaluating raw generated code by hand is highly labor-intensive, sensitive to individual reviewer bias, and often fails to reflect true runtime behavior. Instead, we focus human evaluation strictly on observable and playable behavioral axes, replay and playtest scores, which are judged through a custom browser-based workspace called playground (Figure S22).

Evaluation Protocol. For each game, experts reference the corresponding Game Design Document (GDD) specification and examine axis-specific artifacts within the playground:

- **Replay-based Evaluation (R1–R3):** Examination of recorded video traces and frame sequences to verify rendered behavior (R1), visual feedback and interface indicators (R2), and graphical presentation (R3).
- **Playtest (P1–P3):** Interactive playtesting within an embedded 1280×720 canvas viewport to check normal progression (P1), goal/defeat termination conditions (P2), and downstream mechanics under verified preconditions (P3).

Each panel records scores in $\{0, 0.25, 0.5, 0.75, 1\}$ with supporting observations and optional comments. Experts select `Cannot Observe` when the available evidence does not support a judgment, and these responses are assigned zero. For each expert, we take the equal-weight mean of their *Replay* and *Playtest* ratings and set it to zero when that expert records an execution failure. Then, we uniformly average their overall ratings.

Table S20: **Agreement between benchmark scores and expert assessment.** Spearman correlations with the mean overall expert rating on the same *Small* and *Big* games built by GPT-5.6-Sol. The final row gives inter-expert correlation on the same games. Bold marks the highest benchmark correlation.

Evaluation	Spearman Correlation (ρ)
Scenario-based Replay	0.549
Adaptive Playtest	0.566
Overall *GDD Fidelity*	**0.632**
Human–human reference	0.659

Agreement with A2Z GameSpec-Bench. We compare scenario-based replay score, adaptive playtest score, and overall GDD Fidelity with mean expert ratings using Spearman correlation. As shown in Table S20, the correlations are 0.549 for replay, 0.566 for playtest, and 0.632 for overall GDD Fidelity. Inter-expert correlation between human judgments is 0.659 on the same evaluation set. These results indicate positive corpus-level correspondence between benchmark scores and expert assessments.

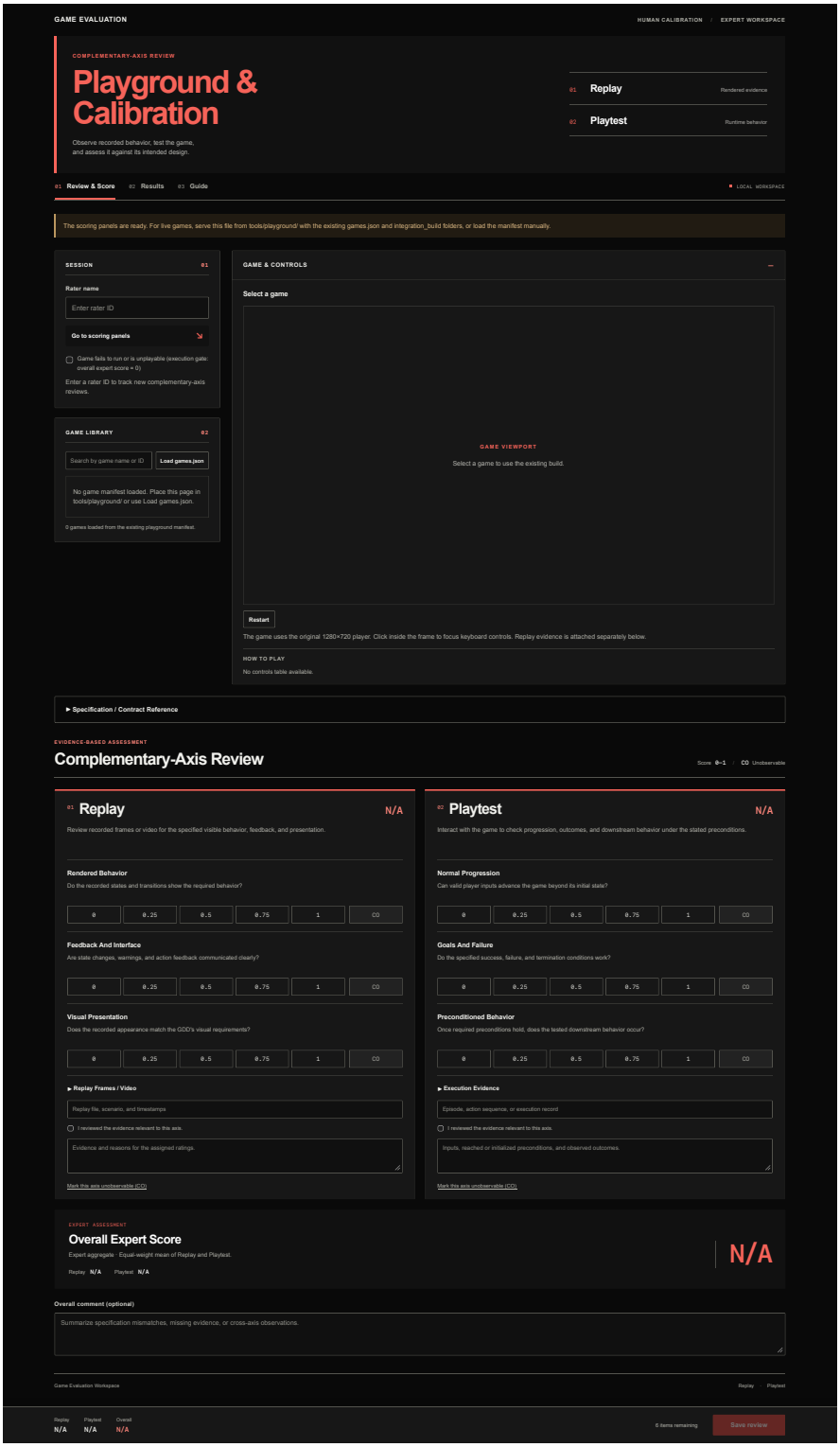


Figure S22: **Playground interface for expert evaluation.** The page interface combines a playable game view with *Replay* and *Playtest* scoring panels, evidence references, and comments.

I SCALABILITY OF A2Z GAMESPEC-BENCH

Towards Engine-Agnostic Evaluation. Our main benchmark uses 2D single-player games in Phaser engine to provide a controlled environment for comparing coding agents. The framework separates the specification of what to verify from the engine-specific procedures used to collect evidence. A dependency-aware contract represents requirements from a GDD, while test policies exercise their conditions, triggers, and effects in the generated game. Extending this process to another environment requires adapting game execution, player input, state observation, scenario initialization, and frame capture. Thus, the contract representation and the three evaluation axes can be retained. Likewise, the brief-to-GDD workflow can specify new game designs and target environments, allowing the framework to accommodate tasks beyond the original 100-game corpus.

Evaluation of 3D games. We demonstrate this extension with three additional 3D games built by GPT-6-Astra using Three.js: *Sonic: Cascade Coast*, *Diablo Cathedral*, and *Rocket League*. These tasks cover momentum-based platforming, dungeon exploration and combat, and vehicle-based ball play. For each game, source-code inspection, scenario-based replay, and adaptive playtesting assess the reported build against its fixed GDD-derived contract. The 3D runtime exposes the observation and interaction interfaces needed by the test policies, including precondition initialization for adversarial playtests. Figure S23 presents the GDD summaries and generated gameplay of `Sonic` and `Rocket League`.

Table S21: **Evaluation of three additional 3D games.** Scores are reported on a 0–100 scale. Playtest combines normal and adversarial judgments, using the full fixed rule set as the denominator.

Game	Source	Replay	Playtest
`Sonic: Cascade Coast`	99.71	53.50	80.00
`Diablo Cathedral`	86.74	55.00	66.55
`Rocket League`	83.64	57.40	77.54

Table S21 shows that the framework produces complementary fidelity assessments in this 3D environment. Sonic receives a source-code score of 99.71, but its Replay and Playtest scores are 53.50 and 80.00, respectively. Thus, high implementation credit alone does not establish fidelity in rendered output or observed gameplay. These cases demonstrate the applicability of the contract-based evaluation structure beyond the main Phaser setting.

Requirement-level feedback. The evaluation also identifies concrete revision targets in these games. For example, Sonic's GDD requires landing velocity to be projected onto the contacted tangent without creating tangential speed. The source evaluator identifies a rail-landing implementation that imposes a minimum speed of 12 m/s, which can increase speed when the projected component is smaller. This diagnosis points to a targeted revision of the velocity update and a subsequent test of low-speed rail landings. Linking such findings to the fixed contract allows a coding agent to revise the game and be reassessed against the same requirements, extending the framework's use to specification-driven development and refinement in additional environments.

(a) Rocket League

GDD (Summarized)

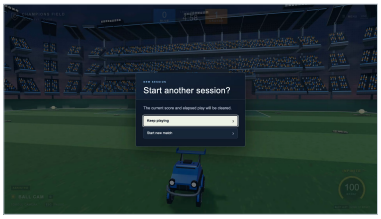
Overview. A local car-soccer game: one player drives a blue Octane around Champions Field at dusk, steering into touches, jumping through shots, and collecting boost. The browser opens on a five-minute match against an orange bot; goals trigger a celebration and a new kickoff, and a tie enters sudden-death overtime. Solo practice removes the opponent and adds ground-shot, wall-aerial, and flip-reset setups on the same rules.

Profile.

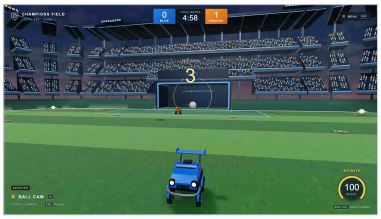
Controls	keyboard driving; keyboard or mouse overlays
Play mode	one human vs. a baseline bot, or solo practice
Session	5-min regulation, 6–10 min with stoppages
Scope	one stadium, one Octane in two finishes, one ball, 34 boost pads

Systems, in reading order.

- Sessions and match flow
- Arena and boost routes
- Vehicle and ball (traction, jumps, aerials, demolitions)
- Opponent and practice (bot decisions, three setups)
- Interface (HUD, pause, remapping, results)
- Camera, art, and sound



New-session dialog over a paused match (0–1, 4:58 left)



Keep playing: 3-second resume countdown, same state

(b) Sonic: Cascade Coast

GDD (Summarized)

Overview. A single-player 3D action-platforming course around a tropical bay. Sonic accelerates down grassy cliffs, carries momentum through a giant loop and corkscrew, grinds ocean rails, and connects airborne attacks into the ruins; a collapsing waterfall ramp and a beach sprint lead to the finish. Lower routes let the player continue after missed links, four checkpoints support retries, and shortcut clusters and sector practice reward returning players.

Profile.

Controls	gamepad primary; keyboard and mouse remappable
Play mode	single player, local records
Session	15–30 min for learning, completion, and retries
Scope	one continuous course, nine landmarks, three shortcut clusters

Systems, in reading order.

- Coastal course (landmarks, routes, recovery)
- Movement (steering, surface and airborne states)
- Traversal links (launchers, rails, robot attacks)
- Attempts and continuation (rings, checkpoints, finish)
- Interface (enter, play, pause, replay)
- Presentation (camera, animation, art, audio)



Cliffside Overlook: opening “Find your stride” prompt, 16 m/s



The Long Descent: 11 rings, 1:54.52, 46 m/s, loop ahead

Figure S23: **Two scalability test cases and their generated games.** Each card summarizes the specifications overview, and system index, together with screens of the generated game. (a) `Rocket League`: a new-session confirmation over a paused 1v1 match, then the resume countdown. (b) `Sonic: Cascade Coast`: the opening Cliffside Overlook prompt, then the Long Descent sector mid-run.

J MORE VISUAL RESULTS

Additional Qualitative Repairs in Iterative Revisions. Figure S24 extends the examples in Figure 5(b) with `Abyssal Chain` and `Whispers of the Wild`. The paired screenshots show projectile visibility and duck placement before revision and after the second round of three-axis feedback.

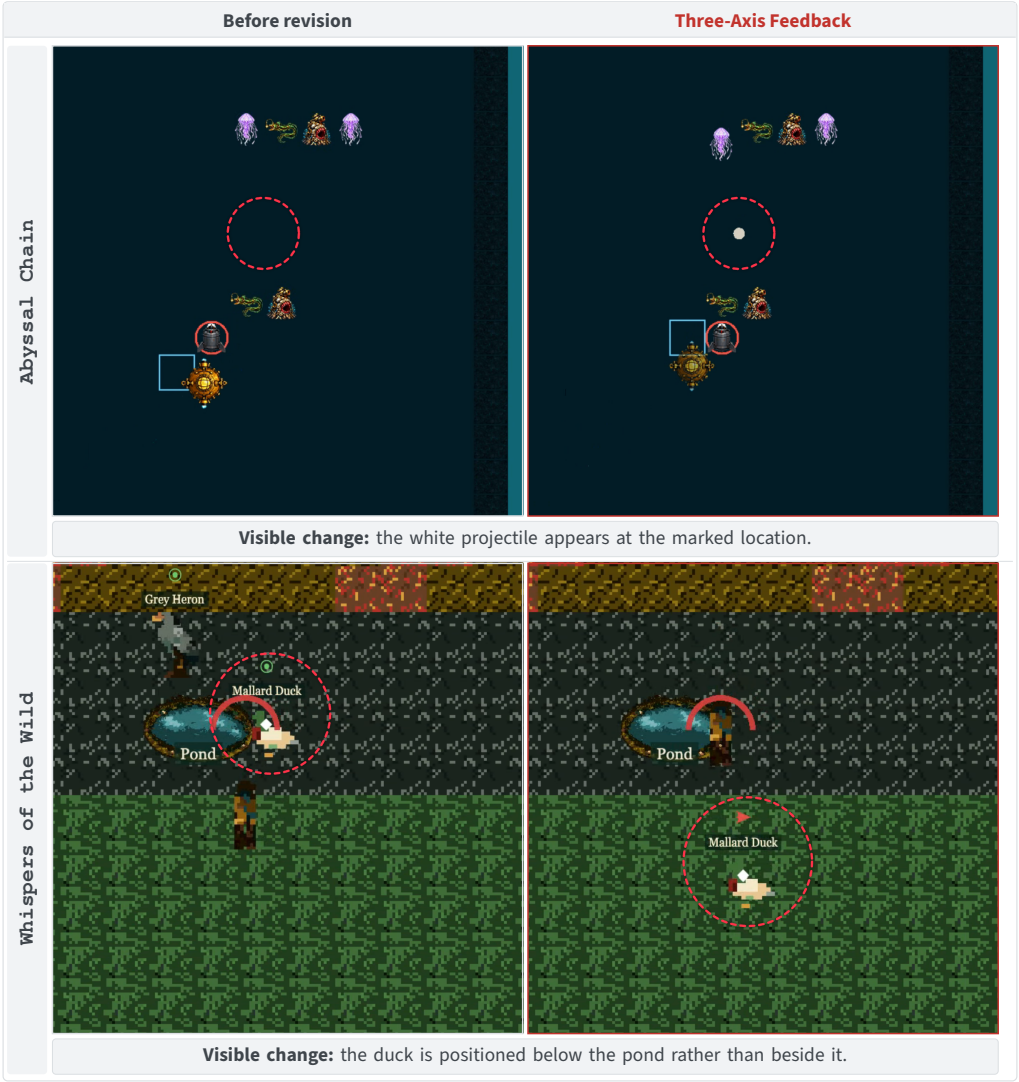


Figure S24: **Additional qualitative repairs following three-axis feedback.** Before-revision and second-round three-axis-feedback screenshots for `Abyssal Chain` (top) and `Whispers of the Wild` (bottom), complementing Figure 5(b). The original image annotations highlight the projectile and duck placement, respectively.

Qualitative Gameplay Examples. Figures S25 and S26 show selected gameplays from eight generated *Small* games. `Bloom or Weed` presents flowers and weeds that require clicking and withholding input, respectively, while `Odd Patch` asks the player to identify differences in color or shape. `Orbit` shows the player on two concentric rings followed by visible feedback at hazard contact, and `Echo Three` alternates between pattern presentation and player input. `Key Under Cups` depicts the reveal, shuffle, and selection feedback of a visual tracking task, while `Bin Bit` communicates shape categories through persistent labels and color-coded bins. `Greedy Die` distinguishes temporary and banked points as the player rolls and secures the target total. `Safe Dial` combines a rotating dial with a fixed pointer and indicators for confirmed and remaining combination entries. These examples illustrate how gameplay prompts and state changes appear in the generated interfaces.

Qualitative Examples From *Big*-GDD Games. Figures S27 and S28 present selected gameplays from eight generated *Big* games. `Cloud Cast` combines casting and reeling controls with catch rewards, while `Afterglow Network` presents traversal tools, regulator installation, and base defense. `Siege Deck 2D` exposes unit deployment, tactical order cards, and deck rewards, and `Wham Bam Logistics` provides route, fleet, and hub management within a shared city map. `ChromaShade` combines shadow controls with platform traversal, while `Ssitgim` presents soul encounters, a cleansing interface, and a boss encounter. `Alias Alchemy Shop` exposes production, shared inventory, and the distinction between resources that reset and progress that persists at closure. `Strata Keepers` provides separate interfaces for recording excavation context, rejoining artifact fragments, and assembling evidence for research hypotheses. These examples illustrate how distinct mechanics are expressed through gameplay controls, state information, and feedback in the generated interfaces.

(a) `Bloom or Weed`: Selective Clicking



Click the Bloom
Rose prompt

Ignore the Weed
Withhold input

Click the Bloom
Daisy prompt

Bloom and weed prompts require different responses: click the flowers and withhold input for the weeds.

(b) `Odd Patch`: Color and Shape Discrimination



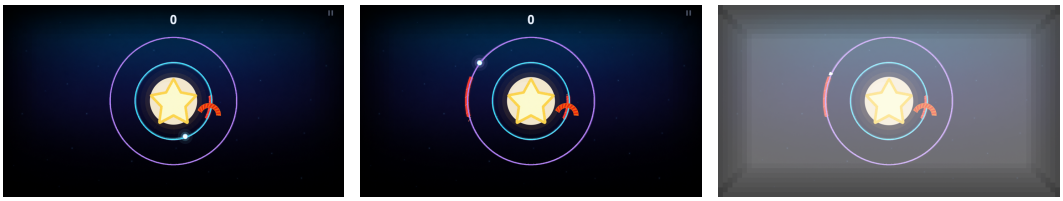
Different Color
Round 1: center patch

Different Shape
Round 2: left patch

Different Color
Round 5: center patch

Choose the patch that differs from the other two. The distinguishing attribute changes across rounds, from color to shape.

(c) `Orbit`: Ring Switching and Hazard Contact



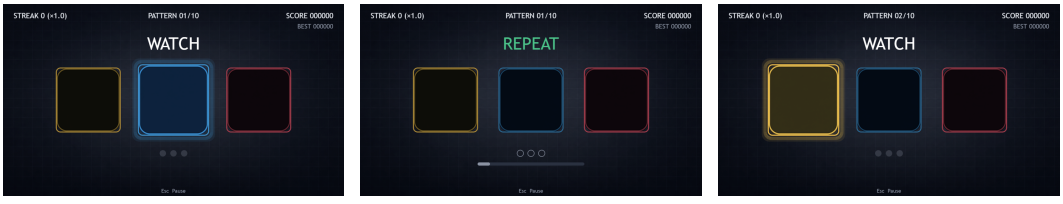
Inner Ring
Player on cyan orbit

Outer Ring
Player on purple orbit

Hazard Contact
Visible collision flash

The player moves between two concentric rings. The final frame shows a screen flash when the player reaches the red hazard.

(d) `Echo Three`: Observe and Repeat



Watch the Pattern
Pattern 1: blue pad

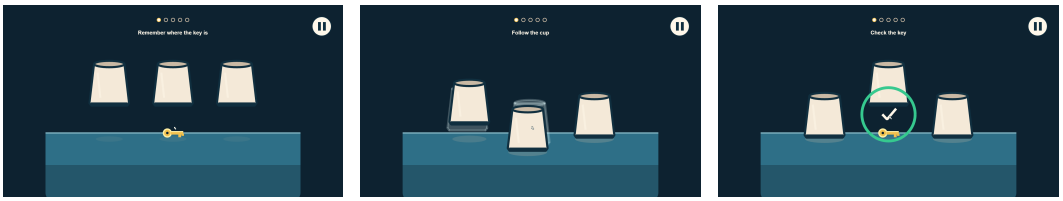
Repeat the Pattern
Pattern 1: input phase

Watch the Next Pattern
Pattern 2: yellow pad

Illuminated pads present a pattern during WATCH, followed by a timed REPEAT phase for player input. A new pattern then begins.

Figure S25: **Sampled gameplay in generated *Small* games (01).** Four games from the *Small* split illustrate (a) selective clicking, (b) visual discrimination, (c) ring switching and collision feedback, and (d) the observation and input phases of a memory task. Each row shows three states in the order of the supplied capture sequence. The full game screens are retained, including their prompts and feedback.

(a) `Key Under Cups`: Visual Tracking and Reveal



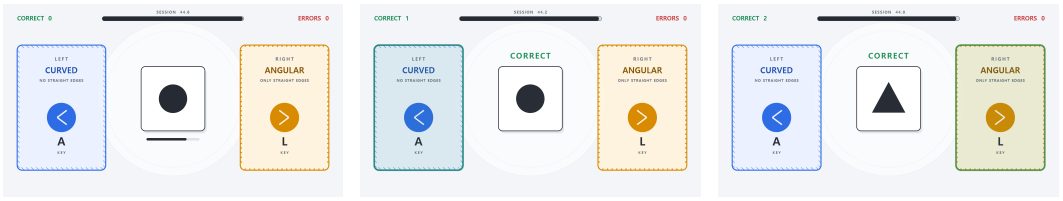
Remember the Key
Key exposed before shuffling

Follow the Shuffle
Cups exchange positions

Check the Choice
Key and correct-answer mark

The key is revealed before the cups move. After the shuffle, the selected cup lifts to expose the key, accompanied by a green check mark.

(b) `Bin Bit`: Shape Sorting and Immediate Feedback



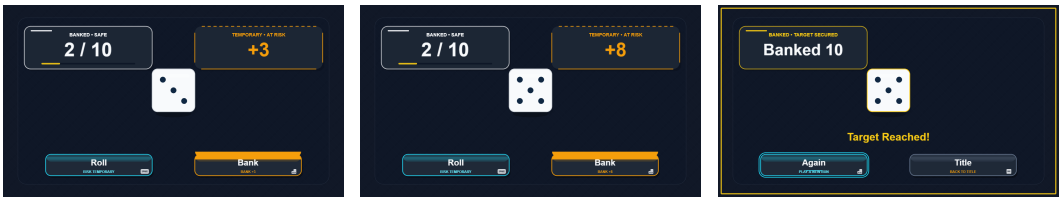
Read the Shape
Circle prompt; correct count 0

Choose Curved
Left bin highlighted; count 1

Choose Angular
Right bin highlighted; count 2

Blue and gold bins retain their curved and angular labels. Correct choices highlight the corresponding bin and increment the visible counter.

(c) `Greedy Die`: Rolling, Accumulation, and Banking



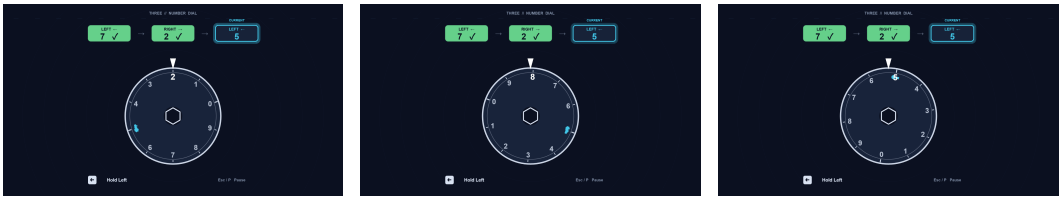
Choose Roll or Bank
Banked: 2; temporary: +3

Accumulate Points
Roll of 5; temporary: +8

Bank the Total
Banked: 10; target reached

Separate panels distinguish banked points from the temporary total. Rolling adds five temporary points; banking then secures the target of ten.

(d) `Safe Dial`: Dial Rotation and Combination Cues



Read the Next Target
Entries 7 and 2 confirmed

Rotate the Dial
Final target: left to 5

Approach the Pointer
Highlighted 5 near the pointer

With two entries already confirmed, the dial rotates toward the highlighted final target. A fixed pointer and progress cards expose the current state.

Figure S26: **Sampled gameplays in generated *Small* games (02)** Additional four frames illustrate (a) tracking a hidden key through a cup shuffle, (b) sorting curved and angular shapes, (c) accumulating and banking dice points, and (d) rotating a combination dial with persistent progress cues. Each row contains three full screenshots from one replay scenario, ordered from left to right.

(a) `Cloud Cast`: Gearing, Reeling, and Catch Rewards



Geared Max
Hold and release Space

Reel In
Catch progress and timing

Catch Reward
Grade, size, and coins

Casting prompts, reel controls, and timing cues support fishing. The catch screen reports the fish, its grade and size, and the coin reward.

(b) `Afterglow Network`: Traversal and Base Defense



Traversal Tools
Cutter pickup prompt

Install a Regulator
Timed interaction prompt

Night Defense
Barrier and turret status

The selected states show tool pickup, a timed regulator installation prompt, and a defense site with separate core, barrier, and turret status.

(c) `Siege Deck 2D`: Deployment and Tactical Cards



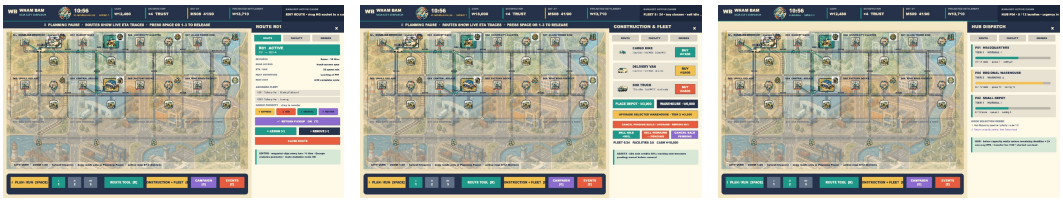
Deploy Units
Roster and placement cells

Plan Orders
Three order slots

Choose a Reward
Cards and deck changes

Deployment cells guide unit placement, and an order hand supports tactical planning. The victory screen offers new cards and deck modification options.

(d) `Wham Bam Logistics`: Route and Fleet Management



Edit Routes
Waypoints and cargo priorities

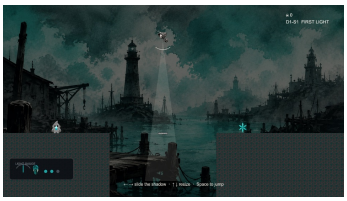
Manage the Fleet
Vehicles and facilities

Monitor Hub Loads
Queues and capacity status

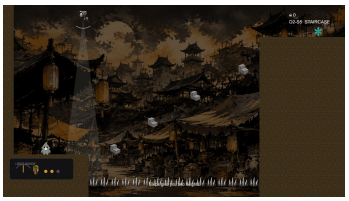
A shared city map connects route editing, vehicle and facility purchases, and hub dispatch. Dedicated panels expose cargo priorities and facility loads.

Figure S27: **Sampled gameplays in generated *Big* games (1).** Four games illustrate (a) fishing controls and catch rewards, (b) traversal tools and base defense, (c) unit deployment and tactical cards, and (d) delivery route and fleet management. Each row groups three scenario-specific screenshots from one game. Full screenshots retain the gameplay context and interface feedback.

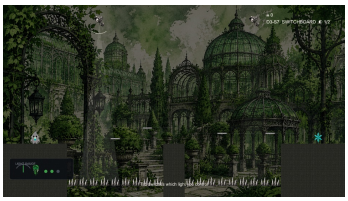
(a) `ChromaShade`: Light and Shadow Traversal



Shape the Shadow
Slide and resize controls



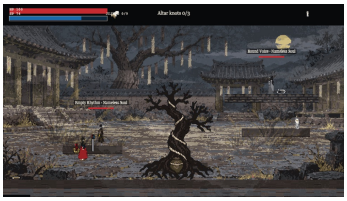
Climb the Steps
Platforms above floor spikes



Switch Lights
Two controllable light sources

Movement puzzles combine shadow controls with platform traversal. Selected stages show stepped obstacles and the option to switch between light sources.

(b) `Ssitgim`: Soul Encounters and Cleansing



Encounter Souls
Two named soul types



Perform a Cleansing Rite
Ring prompt and progress



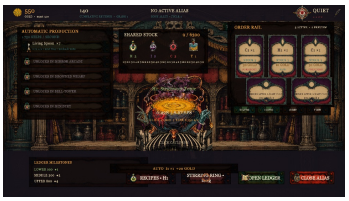
Confront a Boss
Boss identity and health bar

Soul encounters, a ring-based cleansing interface, and a boss encounter present distinct interactions within the same shrine setting.

(c) `Alias Alchemy Shop`: Production and Persistent Progress



Choose an Alias
Shop identity and upgrades



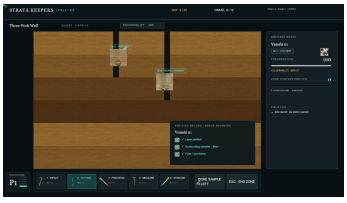
Manage Production
Shared stock and order cards



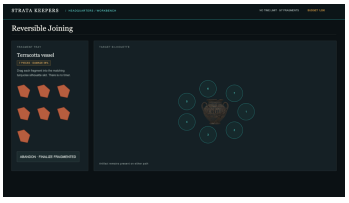
Review Shop Closure
Reset and persistent resources

The shop combines production, shared inventory, and customer orders. Its closure screen lists resources that reset and progress that carries forward.

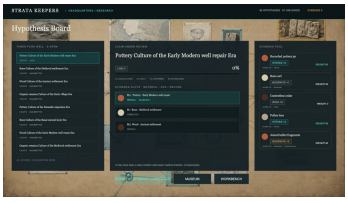
(d) `Strata Keepers`: Excavation, Restoration, and Research



Record a Find
Layer, samples, and orientation



Rejoin Fragments
Drag pieces into target slots



Build a Hypothesis
Evidence slots and confidence

Artifact records preserve excavation context. A joining puzzle and an evidence board provide separate interfaces for restoration and research.

Figure S28: **Sampled gameplays in generated *Big* games (2).** Four games illustrate (a) light and shadow traversal, (b) soul encounters and cleansing, (c) production and persistent shop progress, and (d) excavation, restoration, and research. Each row groups three scenario-specific screenshots from one game. Full screenshots retain the gameplay context, controls, and state information.

K LIMITATIONS

A2Z GameSpec-Bench primarily evaluates 100 synthetic game design documents (GDDs) for two-dimensional single-player games developed in Phaser. Whether these results generalize to human-authored specifications, alternative game engines, and broader software tasks remains to be established. The three additional Three.js games in Appendix I demonstrate applicability to 3D environments but do not establish performance across a broad range of engines or genres. Contract construction and evidence interpretation are based on generative models, so fixed contracts and repeated judgments cannot fully eliminate omissions or evaluator bias. Runtime assessments depend on the behaviors observed and the evidence collected within the allocated test budget; unverified requirements do not necessarily indicate implementation failures. Human calibration involves five experts and games from a single coding-agent configuration, covers both scenario-based replay assessment and adaptive playtest, and does not provide comprehensive requirement-level ground truth or validate Source judgments.