GAMECRAFT-BENCH: CAN AGENTS BUILD PLAYABLE GAMES END-TO-END IN A REAL GAME ENGINE?

Anonymous authors
Paper under double-blind review

ABSTRACT

Game generation is an emerging application of coding agents, requiring models to transform natural-language specifications into playable interactive systems. Unlike traditional coding tasks, game generation takes place within a game engine, where scripts, scenes, assets, rendering, and runtime interactions must jointly produce coherent gameplay. We formalize end-to-end game generation as the problem of producing a complete game artifact that realizes a specification through observable player-game interaction in a target environment. We argue that evaluating this setting requires three desiderata: **Engine Grounding**, **Artifact Completeness**, and **Interactive Verification**. We propose an interaction-grounded evaluation framework that assesses executable gameplay through replayed demonstrations and rubric-guided multimodal judging. We instantiate this framework as **GameCraft-Bench**, a benchmark comprising 140 Godot tasks across 15 game families. Evaluations of frontier coding agents show that end-to-end game generation remains highly challenging: the strongest agent achieves only 68.44%, and most agents score below 40%. Further analysis reveals that while agents often implement recognizable mechanics, they struggle to deliver complete games with sufficient content, functional visual feedback, and coherent presentation. All code, data, demos are provided in the supplementary material.

1 INTRODUCTION

Game generation Yang et al. (2026); Jiang et al. (2026); Ma et al. (2026); Yin et al. (2026) turns creative intent into interactive systems: players do not only view the result, but act inside it and expect the world to respond. Recent coding agents Yang et al. (2024); Team et al. (2026b); Bolin (2026); Anthropic (2026) make automated game generation increasingly plausible, because they can inspect files, edit projects, run tools, and iterate from execution feedback. However, evaluating generated games is not the same as evaluating ordinary code: success depends on whether an artifact can be built, launched, played, and observed as an interactive system.

Desiderata for Game Generation. End-to-end game generation must answer three successive questions: *where* the agent develops, *what* artifacts it delivers, and *how* the resulting game is evaluated. These questions motivate three desiderata: **Desideratum I: Engine Grounding**, which requires development in a real game engine with its native documentation and assets; **Desideratum II: Artifact Completeness**, which requires agents to produce a launchable game together with all necessary artifacts; and **Desideratum III: Interactive Verification**, which requires evaluation through direct interaction using player inputs and gameplay replays. Together, these desiderata shift evaluation from **code correctness** to **interactive system correctness**, establishing a benchmark contract for engine-grounded, fully constructed, and behaviorally evaluated game generation.

The Existing Benchmarks Fail to Meet the Desiderata. Existing game generation benchmarks each satisfy part of this contract, but not all three desiderata. OpenGame-Bench Jiang et al. (2026) satisfies Artifact Completeness by asking agents to generate complete games from open-ended prompts; however, it targets web games rather than real-engine projects and does not judge games through gameplay interactions. GameDevBench Chi et al. (2026) satisfies Engine Grounding by moving evaluation into real engines; however, it studies localized tutorial-derived edits and deterministic tests rather than complete game construction and interaction-based playability. Closest to our setting, WebGameBench (Zhang et al., 2026) evaluates Web games through browser interac-

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.

tion, but it remains outside real game-engine development and relies on evaluation-side exploration. Thus, existing benchmarks leave open the full setting captured by Table 1: whether agents can carry user intent all the way to a complete engine-native game artifact whose behavior is judged through interaction.

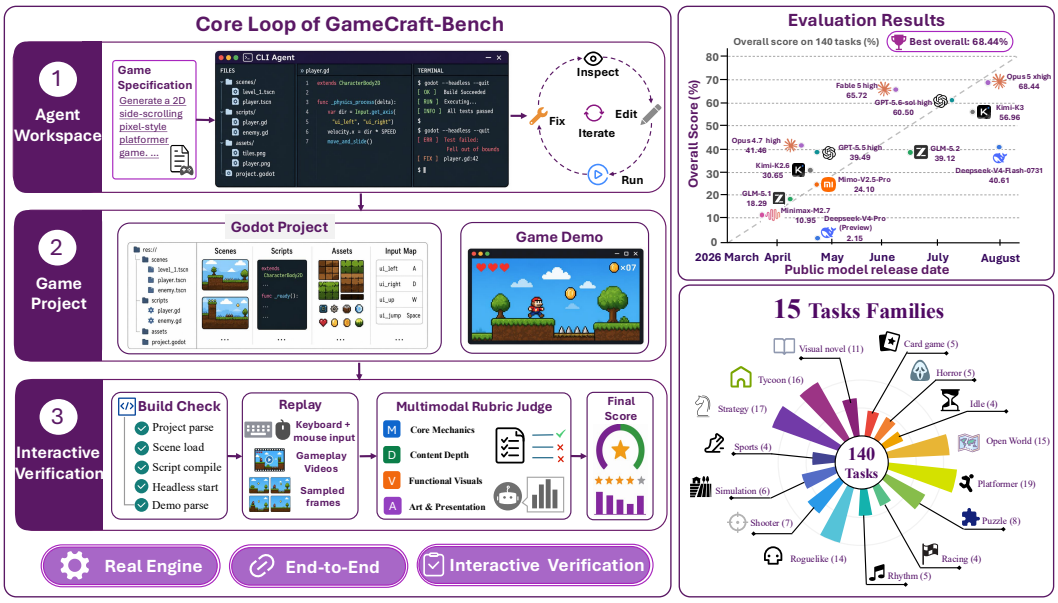


Figure 1: Overview of **GameCraft-Bench**. Agents turn natural-language game specifications into complete Godot projects, the verifier replays submitted interaction traces, and rubric-guided judging scores gameplay evidence.

How GameCraft-Bench Fills the Gap. We introduce **GameCraft-Bench**, an end-to-end benchmark designed to jointly satisfy the three desiderata for game generation. It grounds development in a real game engine (Godot), evaluates complete launchable game projects rather than partial artifacts, and verifies generated games through replayed interactions instead of static inspection alone. The benchmark comprises 140 tasks across 15 game families, covering diverse gameplay requirements ranging from mechanics and progression systems to visual presentation. By unifying **Engine Grounding**, **Artifact Completeness**, and **Interactive Verification**, **GameCraft-Bench** evaluates whether coding agents can transform natural-language game specifications into playable game experiences.

Table 1: Comparison with existing game generation benchmarks along the three desiderata. *Notably, WebGameBench (Zhang et al., 2026) is concurrent with our work: it evaluates delivered Web games, while **GameCraft-Bench** targets complete projects in a dedicated game engine.

Benchmark	Desideratum I Engine Grounding	Desideratum II Artifact Completeness	Desideratum III Interactive Verification
GameDevBench Chi et al. (2026)	Godot	× (Single Operation)	×
OpenGame-Bench Jiang et al. (2026)	Web	✓	×
WebGameBench* Zhang et al. (2026)	Web	✓	✓
GameCraft-Bench	Godot	✓	✓

Observations on GameCraft-Bench. We evaluate frontier coding agents on **GameCraft-Bench** and find that end-to-end game generation remains far from solved. Even the strongest agent achieves only 68.44% overall, and most evaluated agents score substantially lower. The low scores are not caused by a single missing capability: agents fail across core mechanics, content depth, functional visuals, and art and presentation, with performance varying sharply across game families. These results suggest that current coding agents can often produce fragments of game-like software, but

still struggle to reliably follow a game specification all the way to a playable, visually coherent artifact in a real engine.

Contributions. We summarize our contributions as follows:

- We formalize end-to-end game generation as the problem of transforming game specifications into playable interactive systems, and identify three desiderata for its evaluation: **Engine Grounding**, **Artifact Completeness**, and **Interactive Verification**.
- We propose an interaction-grounded evaluation framework that verifies generated games through executable gameplay evidence, combining replay-based interaction protocols with rubric-guided assessment of mechanics, content, visual functionality, and presentation quality.
- We instantiate this framework as **GameCraft-Bench**, a benchmark comprising 140 tasks across 15 game families in Godot, where agents must generate complete game projects from natural-language game specifications.
- We benchmark frontier coding agents and provide diagnostic analyses showing that current systems remain far from reliable end-to-end game generation, with limitations spanning mechanics, content depth, visual feedback, presentation quality, and task-completion behavior.

2 What Should Be a Good Game Generation Benchmark?

2.1 Problem Definition

AI game generation studies *how an agent turns a high-level design intent into a playable interactive system*. As illustrated in Figure 2, an agent receives a game specification, operates within a development and runtime environment, and produces a playable game artifact. We abstract this process as

$$x = (s, \mathcal{E}) \quad \longmapsto \quad y = G, \qquad (1)$$

where s is a game specification describing the intended player experience, including rules, mechanics, goals, content, and presentation; $\mathcal{E}$ is the target development and runtime environment; and G is the generated game artifact. A successful output G should be launchable in $\mathcal{E}$ and should realize s through observable responses to player actions.

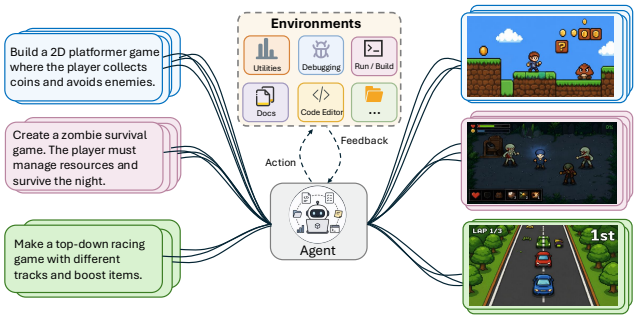


Figure 2: Problem definition for AI game generation. An agent transforms natural-language game specifications s into playable game artifacts G by acting within a development and runtime environment $\mathcal{E}$.

2.2 Three Desiderata.

A good benchmark must preserve the full meaning of the mapping $(s, \mathcal{E}) \mapsto G$ by constraining three spaces: the **development space**, or where the game is constructed; the **output space**, or what artifact is delivered; and the **evaluation space**, or how that artifact is judged. These spaces induce **Engine Grounding**, **Artifact Completeness**, and **Interactive Verification**, which jointly cover development in an operational environment, delivery of a complete game, and verification through interaction (Figure 3). Leaving any space unconstrained reduces end-to-end evaluation to toy environments, partial artifacts, or static assessments that miss gameplay failures.

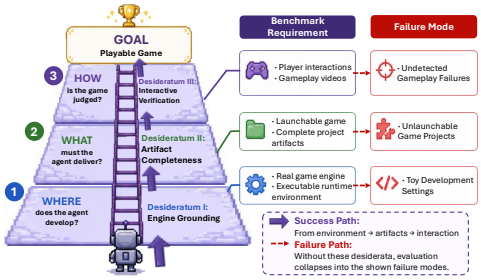


Figure 3: Three desiderata for evaluating end-to-end game generation.

Desideratum I ***Engine Grounding***. *Games should be evaluated within a* ***concrete*** *game generation and runtime* ***environment***.

Game behavior depends on engine-level semantics—including scene hierarchy, scripting lifecycle, assets, input dispatch, physics, rendering, configuration, and launch procedure—rather than source code alone. Engine Grounding preserves these runtime constraints, preventing locally plausible code that never functions in the operational game environment from receiving credit.

Desideratum II ***Artifact Completeness***. *Games should be evaluated as a* ***complete*** *launchable* ***artifact***.

Playability requires project metadata, scenes, scripts, assets, UI, input mappings, configuration, and runtime resources to form a functioning whole. Artifact Completeness therefore makes a self-contained, launchable project the unit of evaluation, preventing isolated scripts, prototypes, or scenes that require additional human assembly from receiving full credit; commercial-level polish is not required.

Desideratum III ***Interactive Verification***. *Games should ultimately be judged by what happens when they are* ***played interactively***.

A game is defined by its action–response loop, and failures such as unresponsive controls, incorrect collisions, unreachable objectives, broken state transitions, or unreadable feedback may appear only during play. Interactive Verification therefore judges observed behavior after the artifact is run and acted upon through a standardized interaction procedure, rather than certifying an artifact from static inspection alone.

3 THE GAMECRAFT-BENCH

3.1 TASK DEFINITION

GameCraft-Bench instantiates the general game-generation mapping $(s, \mathcal{E}) \mapsto G$ in a real game-engine setting. Each task is defined as

$$\tau = (s, \mathcal{E}, \rho), \tag{2}$$

where s is the game specification given to the agent, $\mathcal{E}$ is the target development and runtime environment, and ρ is the hidden rubric used by the verifier. In **GameCraft-Bench**, the environment $\mathcal{E}$ is operationalized as

$$\mathcal{E} = (\mathcal{R}, \mathcal{W}, \mathcal{A}, \mathcal{C}), \tag{3}$$

where $\mathcal{R}$ is the Godot engine runtime and toolchain, $\mathcal{W}$ is the editable workspace, $\mathcal{A}$ is the shared resource interface, and $\mathcal{C}$ is the submission contract. The agent observes $(s, \mathcal{E})$, but not ρ.

Given $(s, \mathcal{E})$, the agent must produce

$$y = (G, \Pi), \tag{4}$$

where G is a complete Godot project and $\Pi = \{\pi_i\}_{i=1}^{n}$ is a set of replayable interaction traces. Notably, the traces are not part of the game artifact itself; they provide standardized interaction evidence for evaluation. The verifier launches G in $\mathcal{R}$ and replays Π to obtain gameplay observations:

$$O = \mathrm{Replay}_{\mathcal{R}}(G, \Pi). \tag{5}$$

A successful submission is one whose observed behavior O realizes the specification s under the constraints of $\mathcal{E}$.

3.2 IMPLEMENTATION

Figure 4 summarizes the five-stage pipeline used for every task, which standardizes evaluation without forcing open-ended games to share controls, levels, objectives, or termination conditions.

Stage 1: Task Packaging. Each task packages a natural-language game specification, a Godot development environment, and a hidden rubric. The specification describes the intended experience and demonstration scenarios; the environment provides the runtime, workspace, shared assets, documentation, helper tools, and submission contract; and the rubric translates the same intent into observable requirements for Mechanics, Depth, Visuals, and Art. The agent receives the specification and environment, but not the rubric.

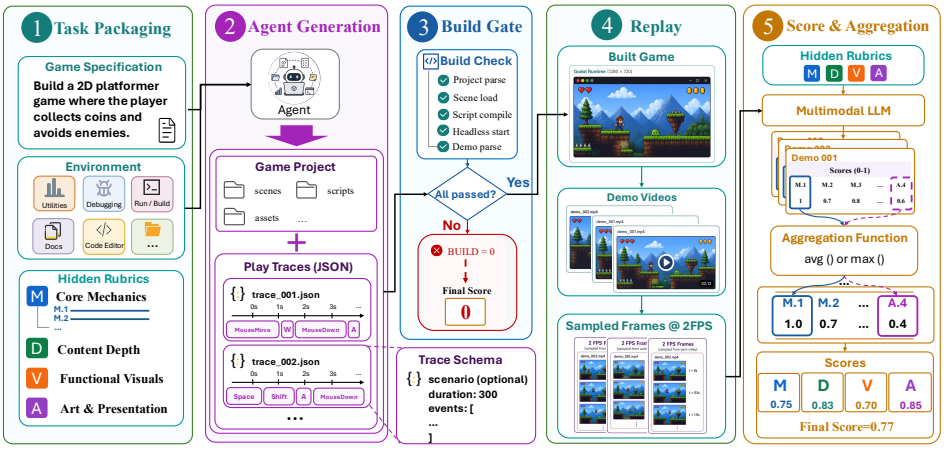


Figure 4: End-to-end evaluation pipeline of **GameCraft-Bench**. A task packages a game specification, Godot environment, and hidden rubric; the agent produces a complete game project and replay traces; the verifier checks launchability, records each replay as a gameplay video, samples frames from the video, and scores only those frames with rubric-guided multimodal judging.

Stage 2: Agent Generation. The agent constructs and iteratively tests a Godot project in the packaged environment, then submits the complete project G and replayable traces Π. Each trace contains timed mouse and keyboard events, optionally with a scenario identifier, that specify how the verifier should expose relevant gameplay behavior.

Stage 3: Build Gate. The verifier parses the project and traces, launches the game under the benchmark runtime, and checks for a runnable state. If the project fails to launch or no trace can be parsed, it sets $\mathrm{BUILD} = 0$, making the final score zero.

Stage 4: Replay. For each valid trace, the verifier starts a fresh Godot instance, replays the recorded events in a fixed 1280×720 viewport, records a gameplay video, and samples frames at two frames per second. Only the sampled frames, not the full videos, are passed to the judge.

Stage 5: Scoring and Aggregation. Each demo is scored independently against the hidden rubric by a multimodal judge using the sampled frames. Following MDA and playability heuristics (Hunicke et al., 2004; Desurvire et al., 2004), the four categories capture rules and state transitions (Mechanics), runtime scope and progression (Depth), readability and gameplay feedback (Visuals), and the authored sensory layer (Art). Scenario-specific requirements use maximum aggregation across demos, while persistent requirements use mean aggregation. The resulting category scores M, D, V, and A are combined as

$$\mathrm{Score} = \mathrm{BUILD} \times (w_M \mathrm{M} + w_D \mathrm{D} + w_V \mathrm{V} + w_A \mathrm{A}), \tag{6}$$

with default weights $(w_M, w_D, w_V, w_A) = (0.15, 0.35, 0.15, 0.35)$. The higher weights on Depth and Art emphasize complete games over minimal mechanics and functional prototypes.

3.3 How It Fulfills the Three Desiderata

Grounded on a Real Engine: Godot (Desideratum I) GameCraft-Bench uses Godot 4 as the concrete engine environment. The goal is not to claim that Godot is the only automatable game engine, but to choose an engine that makes large-scale, reproducible, execution-based evaluation practical. Godot is open source, lightweight, provides a native 2D engine stack, stores scenes in text-based project files, and supports command-line/headless execution for running and exporting projects. These properties let the verifier launch generated games, replay traces, record gameplay evidence, and inspect failures without relying on a GUI editor workflow. By contrast, Unity, Unreal, and GameMaker also support automation in different forms, but they introduce heavier installations, proprietary licensing constraints, editor-centric workflows, binary project formats, or weaker fit for lightweight 2D benchmark execution. For this reason, **GameCraft-Bench** uses Godot as the benchmark environment while leaving multi-engine evaluation to future work.

Table 2: Engine fit for automated 2D game-generation benchmarking. △ indicates support with heavier setup, editor/toolchain constraints, or weaker fit for lightweight reproducible evaluation.

Benchmark Requirement	Godot	Unity	Unreal	GameMaker
Open-source	✓	×	×	×
Lightweight setup	✓	△	×	✓
Command-line/headless-friendly	✓	△	△	△
Native 2D workflow	✓	△	△	✓
Text-based scene/files	✓	△	△	×

Full Game Delivery (Desideratum II) GameCraft-Bench makes the generated game artifact G the unit of submission. The submission contract $\mathcal{C}$ requires agents to place a complete Godot project under the target workspace, including project metadata, entry scenes, scripts, copied assets, input configuration, UI resources, and the runtime entry point needed to launch the game. This prevents agents from receiving credit for isolated mechanics, loose assets, partial scenes, or code fragments that require additional human assembly. The verifier enforces this requirement with a launchability gate: if the project cannot be launched in the Godot runtime, $\textsc{Build} = 0$ and the final score is zero. Thus, **GameCraft-Bench** evaluates whether an agent can deliver a self-contained playable artifact, not merely produce game-like components.

Interactive Evaluation (Desideratum III) GameCraft-Bench grounds evaluation in observed player–game interaction rather than static inspection. Obtaining such evidence poses the trade-off shown in Figure 5. A black-box judge must infer controls and explore within a limited budget, potentially missing important content and conflating game quality with exploration ability. A white-box verifier could instead synthesize traces from code, but this requires costly program understanding and makes evaluation depend on verifier capability. We therefore ask the builder agent, which knows its controls and progression, to submit demonstrations targeting relevant gameplay phases, while the verifier performs standardized replay.

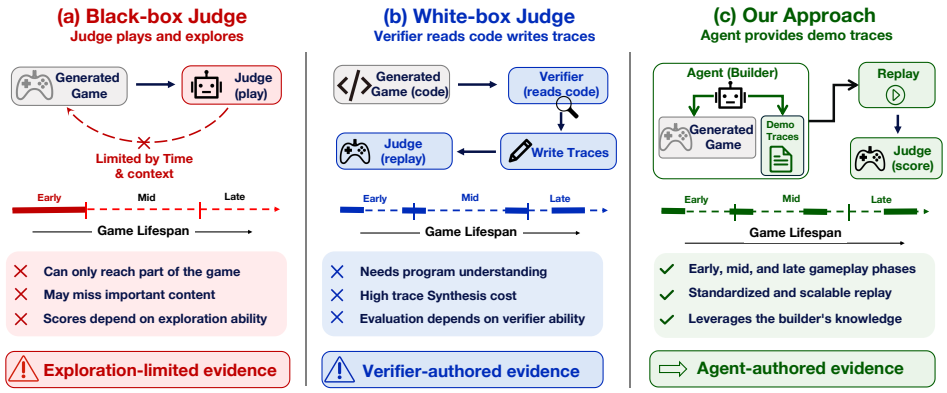


Figure 5: Comparison of interaction-evidence authoring strategies. Black-box exploration is coverage-limited, while white-box trace synthesis requires program understanding. We instead combine agent-authored demonstrations with verifier-side replay and generator-blind judging.

Each submission includes replayable traces Π; the verifier independently executes the project G, replays the traces, and records $O = \text{Replay}_{\mathcal{R}}(G, \Pi)$. The hidden rubric ρ is then applied to this gameplay evidence to score the requested mechanics, content, visual feedback, and presentation. Thus, the agent authors the evidence rather than the verdict: execution and scoring remain external, and static artifacts do not substitute for observed behavior under player input.

3.4 Task Suite and Annotation Quality

GameCraft-Bench contains 140 tasks across 15 game families. The families are chosen to cover distinct game-construction demands rather than surface genres alone: continuous control and colli-

sion in platformers and racing games, rule and state management in card and strategy games, progression and economy in tycoon and idle games, exploration and spatial layout in open-world and roguelike games, and narrative or presentation-heavy interaction in visual novels, horror, rhythm, and sports games. Figure 1 summarizes the family-level coverage.

4 BENCHMARKING RESULTS

Experimental Setup. We evaluate 13 frontier coding-agent configurations across Claude Code (Anthropic, 2026), Codex (Bolin, 2026), Kimi Code (Team et al., 2026b), and Code Buddy, spanning recent models from Anthropic (Anthropic, 2026;a;b), OpenAI (OpenAI, 2026a;b), MoonshotAI Team et al. (2026b;a), DeepSeek (DeepSeek-AI, 2026), Xiaomi (Xiaomi MiMo Team, 2026), GLM (Z.ai, 2026a;b), and MiniMax (MiniMax, 2026). Table 3 lists the exact configurations and settings. Each configuration is run on all 140 tasks under the same benchmark interface and environment; Appendix C provides the detailed evaluation setup.

Table 3: Leaderboard results (%) on the full 140-task benchmark. Configurations are ordered by Overall score. Modality indicates whether the agent model can consume rendered screenshots. Mechanics, Depth, Visuals, and Art correspond to the four rubric categories. Best scores are **bolded**, and second-best scores are underlined.

Harness	Model / Effort	Modality	Overall	Mechanics	Depth	Visuals	Art
Claude Code	Opus-5 xhigh	Vision	**68.44**	**76.75**	**63.31**	**68.04**	**70.11**
Claude Code	Fable-5 high	Vision	65.72	76.52	58.61	66.93	67.68
Codex	GPT-5.6-sol high	Vision	60.50	74.50	56.10	64.80	57.00
Kimi Code	Kimi-K3	Vision	56.96	73.03	53.38	56.90	53.59
Claude Code	Opus-4.7 high	Vision	41.46	55.34	39.48	42.78	36.86
Claude Code	DeepSeek-V4-Flash-0731	Text	40.61	52.11	37.14	43.70	37.72
Codex	GPT-5.5 high	Vision	39.49	54.36	38.61	41.84	32.94
Claude Code	GLM-5.2	Text	39.12	50.09	36.40	41.06	36.29
Kimi Code	Kimi-K2.6	Vision	30.65	39.76	28.07	33.66	27.99
Claude Code	MiMo-V2.5-Pro	Text	24.10	32.33	22.59	27.45	20.65
Code Buddy	GLM-5.1	Text	18.29	25.23	17.80	21.14	14.59
Code Buddy	MiniMax-M2.7	Text	10.95	14.27	9.92	14.92	8.85
Codex	DeepSeek-V4-Pro	Text	2.15	2.25	1.69	1.97	2.63

Main Results. Table 3 reports benchmark-level scores in percentage points; detailed family-level results for configurations with available breakdowns are provided in Appendix D. The strongest configuration, Opus-5 xhigh under Claude Code, reaches 68.44% overall, followed by Fable-5 high at 65.72%, GPT-5.6-sol high at 60.50%, and Kimi-K3 at 56.96%. Opus-4.7 high and DeepSeek-V4-Flash-0731 form the next tier at 41.46% and 40.61%, while the remaining configurations score below 40%. Thus, even the best current coding agents remain far below reliable end-to-end game generation: they can often produce runnable game-like artifacts, but they do not consistently realize the requested mechanics, content, visual state, and presentation. For the seven configurations with available execution logs, Appendix Figure 9 further separates launchability from replayable evidence. Most strong agents satisfy the basic execution contract, but low final scores show that producing a runnable project and replayable traces is still far from satisfying the rubric. DeepSeek-V4-Pro often ignores or violates the demonstration-trace requirement, producing a large gap between build pass rate and valid-trace rate.

Category-level Results. Opus-5 xhigh leads every aggregate category, reaching 76.75% on Core Mechanics, 63.31% on Content Depth, 68.04% on Functional Visuals, and 70.11% on Art and Presentation. The family-level results in Appendix D, however, show that this progress is structured rather than uniform: Opus-5 exceeds 76% overall on Strategy and Tycoon, but reaches only 54.59% on Open-world, where its Mechanics score of 69.70% contrasts with 43.70% on Depth and 41.98% on Visuals. DeepSeek-V4-Flash-0731 and GLM-5.2 are close in aggregate score but exhibit different specializations: the former is stronger on Horror, Idle, and Tycoon, whereas the latter performs better on Visual novel, Roguelike, and Puzzle. Both score poorly on Rhythm and Racing, whereas Opus-5 reaches 70.05% and 68.06% on these families, showing that newer configurations substantially narrow earlier gaps in timing-sensitive interaction.

Findings 1.

Open-ended worlds remain difficult even for the strongest configurations, while newer configurations substantially narrow earlier gaps on timing-sensitive games.

5 In-depth Analysis

Appendix E reports complementary analyses of execution and replay, MiMo-V2.5-Pro's task-completion failure pattern, human calibration, and category decomposability.

5.1 On the Diagnostic Patterns of Agents

Which Edit Targets Follow Visual Review? Rendered feedback exposes player-facing failures that are invisible in code, but inspection counts alone do not reveal which artifact layer an agent repairs.

We define a *post-review edit candidate* as the next implementation edit within four semantic turns of visual review, yielding 392 candidates across 137 GPT-5.6-sol runs and 267 across 35 Fable-5 runs. Two independent coding passes agree strongly on an 80-candidate stratified audit ($\kappa = .870$). Figure 6 shows that Presentation accounts for 54.3% of Fable-5-associated candidates versus 23.8% for GPT-5.6-sol ($+30.5$ points; 95% interval $[+8.9, +51.0]$), while state transitions (3.3% versus 35.1%; -31.8 points, $[-45.2, -18.4]$) and mixed edits (5.2% versus 27.9%; -22.7 points, $[-39.3, -5.5]$) reverse this pattern. Fable-5 therefore concentrates nearby edits on visible presentation, whereas GPT-5.6-sol more often revises state transitions or multiple layers. Because the trajectory populations are not task-matched and configuration factors vary together, we treat these as workflow signatures rather than causal model effects.

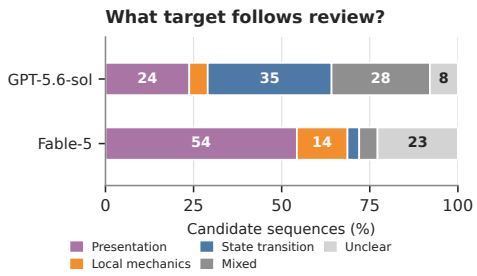


Figure 6: Targets of post-review edits; bars show inverse-probability-weighted estimates from a stratified audit.

Findings 2.

Frontier agents use visual feedback differently: some focus on how the game looks, while others focus on how it works.

Kimi-K2.6: Perception-Guided Debugging. Across 140 tasks, Kimi-K2.6 performs 2,998 rendered-screen inspections (mean 21.41; median 19; only four tasks with none), versus 1,952 for Opus-4.7 (13.94 per task) and 268 for GPT-5.5 (1.91 per task). Figure 7 exemplifies perception-guided iteration: in a *Strategy-Skirmish* trajectory, repeated inspection reveals misaligned units and missing selection highlights, prompting grid and turn-indicator layout fixes.

5.2 On the Reliability of Playability Judge

Stability: Does Fixed Evidence Receive Consistent Scores? GameCraft-Bench relies on a multimodal judge, so we first test whether identical playable evidence receives stable scores across repeated evaluations. We fix the sampled replay frames and rubric for tasks from two families, Simulation and Card Game, and rerun only the GPT-5.5 judge $K = 10$ times. As shown in Figure 8, the mean scores remain consistent across repeated evaluations, and the error bars are small across the evaluated agent configurations and task families. Quantitatively, Kimi-K2.6 exhibits a standard deviation of 0.0037 on Card Game and 0.0038 on Simulation, while Opus-4.7 high exhibits standard deviations of 0.0050 and 0.0036, respectively. These variations are far smaller than the performance gaps across agents and families, so the observed rankings are robust to repeated judging noise. Residual noise mainly comes from borderline visual interpretations, such as whether a card interaction or state transition is visually clear enough.

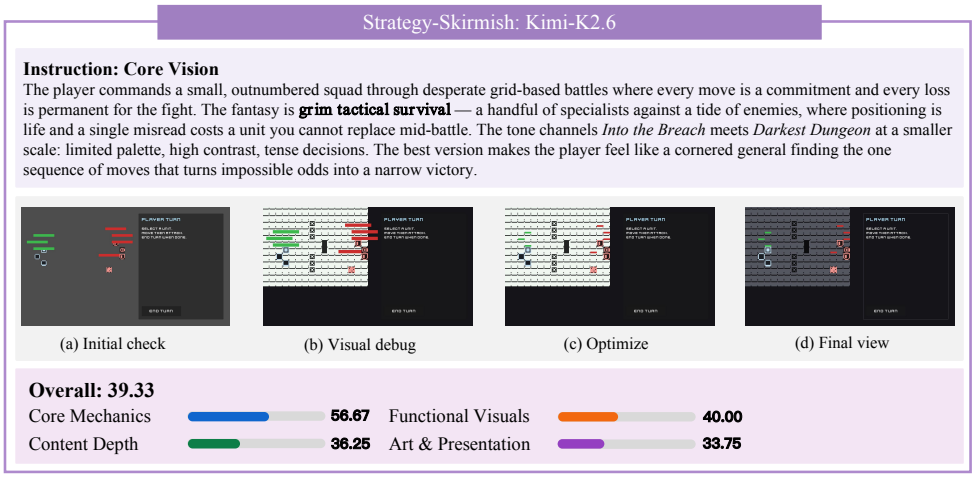


Figure 7: Example of perception-guided debugging in Kimi-K2.6. Repeated inspection of rendered gameplay enables the agent to identify and correct player-facing failures.

Calibration: How Does the Judge Compare with Humans? We also run a preliminary human calibration on Kimi-K2.6 submissions from three families: five annotators score each rubric item after reviewing its demos and, when useful, playing the game directly. The multimodal judge is close to the human average overall but is 3.32 percentage points more permissive; Appendix Table 5 provides the full breakdown. The main discrepancy is category-specific: relative to humans, the judge is more permissive on Content Depth (+5.38 points) and Art and Presentation (+5.80), but stricter on Functional Visuals (−5.87). At the family level, Card Game is nearly matched, whereas Idle shows the largest permissive gap, suggesting that future calibration should focus on content variety and presentation judgments. Because this subset covers only three families, we treat it as a preliminary calibration check rather than a definitive human-agreement study.

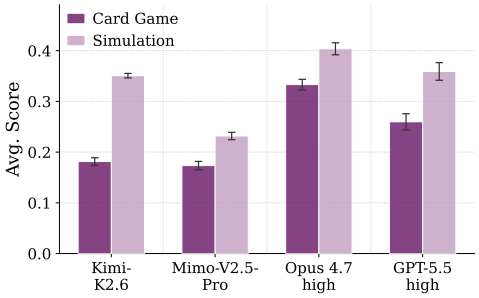


Figure 8: Judge stability on fixed gameplay evidence. Bars show mean scores over 10 repeated GPT-5.5 judge runs, and error bars show ±1 standard deviation.

Findings 3.

The rubric judge is stable on fixed evidence and shows preliminary agreement with human ratings.

6 Conclusion

We presented an interaction-grounded evaluation framework for end-to-end game generation and instantiate it as **GameCraft-Bench**, a Godot-based benchmark for coding agents. The central claim is that game generation cannot be evaluated as ordinary code synthesis: a benchmark must preserve the engine environment, require a complete artifact, and judge the artifact through interaction. Our results show that current frontier agents remain far from reliable under this contract. They can often produce recognizable mechanics or runnable prototypes, but still struggle to assemble coherent games with sufficient content, functional feedback, presentation quality, and completed demonstration traces. More broadly, our findings suggest that evaluating executable creative software systems requires interaction-grounded assessment beyond static code inspection, build success, or visual plausibility alone.

AI USE STATEMENT

Generative AI systems were used in two roles in this work. First, as part of the experimental apparatus, the coding-agent configurations reported in Table 3 generated the evaluated Godot projects and demonstration traces, and GPT-5.5 was used as the multimodal rubric judge. Second, generative AI tools were used to assist with benchmark implementation and data analysis; support paper structuring, and language editing. All AI-assisted outputs were reviewed and validated by the authors. The authors verified the reported results and take responsibility for the final text, claims, code, figures, and artifacts.

REPRODUCIBILITY STATEMENT

Appendix C reports the runtime environment, software versions, timeouts, trace schema, frame-sampling procedure, judge model, and judge prompt. Appendix F describes task annotation and quality control. The supplementary material contains the benchmark tasks, rubrics, verifier and evaluation configurations needed to reproduce the reported experiments.

REFERENCES

Anthropic. Claude Opus 4.7. Anthropic Blog Post, 2026. URL https://www.anthropic.com/news/claude-opus-4-7.

Anthropic. Claude code overview. Official Documentation, 2026. URL https://code.claude.com/docs.

Anthropic. Claude fable 5 and claude mythos 5, June 2026a. URL https://www.anthropic.com/news/claude-fable-5-mythos-5.

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

Michael Bolin. Unrolling the codex agent loop. OpenAI Engineering Blog, January 2026. URL https://openai.com/index/unrolling-the-codex-agent-loop/.

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.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Heather Desurvire, Martin Caplan, and Jozsef A Toth. Using heuristics to evaluate the playability of games. In *CHI'04 extended abstracts on Human factors in computing systems*, pp. 1509–1512, 2004.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, January 2026. URL https://github.com/harbor-framework/harbor.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024a.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14281–14290, 2024b.

Robin Hunicke, Marc LeBlanc, Robert Zubek, et al. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, pp. 1722. San Jose, CA, 2004.

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.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Hongnan Ma, Han Wang, Shenglin Wang, Tieyue Yin, Yiwei Shi, Yucong Huang, Yingtian Zou, Muning Wen, and Mengyue Yang. CreativeGame: Toward Mechanic-Aware Creative Game Generation, 2026. URL https://arxiv.org/abs/2604.19926.

MiniMax. MiniMax-M2.7. Hugging Face Model Repository, 2026. URL https://huggingface.co/MiniMaxAI/MiniMax-M2.7.

OpenAI. GPT-5.5 system card. OpenAI, 2026a. URL https://openai.com/index/gpt-5-5-system-card/.

OpenAI. Gpt-5.6 system card, July 2026b. URL https://deploymentsafety.openai.com/gpt-5-6. System Card.

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

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026a.

Kimi Team, Tongtong Bai, Yifan Bai, et al. Kimi K2.5: Visual agentic intelligence, 2026b. URL https://arxiv.org/abs/2602.02276.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: Foundation action model for generalist gui agents. In *International Conference on Learning Representations*, volume 2025, pp. 5090–5108, 2025.

Xiaomi MiMo Team. Mimo-v2.5-pro. https://huggingface.co/collections/XiaomiMiMo/mimo-v25, 2026.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

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

Runxin Yang, Yuxuan Wan, Shuqing Li, and Michael R. Lyu. 90% faster, 100% code-free: Mllm-driven zero-code 3d game development. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, FSE Companion '26, pp. 1381–1385, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400726361. doi: 10.1145/3803437.3805592. URL https://doi.org/10.1145/3803437.3805592.

Lei Yin, Wentao Cheng, Zhida Qin, Tianyu Huang, Yidong Li, and Gangyi Ding. AutoUE: Automated Generation of 3D Games in Unreal Engine via Multi-Agent Systems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2026), Findings*, 2026. URL https://arxiv.org/abs/2603.07106.

Z.ai. GLM-5.1: Towards long-horizon tasks, April 2026a. URL https://z.ai/blog/glm-5.1. Technical Blog.

Z.ai. GLM-5.2: Built for long-horizon tasks, June 2026b. URL https://z.ai/blog/glm-5.2. Technical Blog.

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

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

A Related Work

Coding Agents and Software Engineering Evaluation. Coding agents have evolved from code completion models such as CodeBERT (Feng et al., 2020) and StarCoder (Li et al., 2023) to autonomous software engineering systems capable of repository navigation, multi-file editing, and iterative debugging. Recent frameworks, including SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), ChatDev (Qian et al., 2024), and MetaGPT (Hong et al., 2024a), increasingly evaluate agents on realistic software development tasks. Existing software engineering benchmarks, however, primarily assess correctness through code patches, unit tests, or issue resolution outcomes, assuming that executable behavior can be sufficiently captured by static artifacts. End-to-end game generation challenges this assumption because success depends on the quality of interactive runtime behavior rather than source code alone.

GUI Agents and Interactive Evaluation. Recent advances in GUI agents enable models to interact directly with desktop and web environments through visual perception and native interface actions. Benchmarks such as Mind2Web (Deng et al., 2023), WebArena (Zhou et al., 2024), and OSWorld (Xie et al., 2024) emphasize end-to-end task completion under interactive settings, while systems including CogAgent (Hong et al., 2024b) and OSAtlas (Wu et al., 2025) demonstrate increasingly capable computer-use behaviors. These developments highlight the importance of evaluating agents through interaction rather than static outputs alone. However, existing GUI benchmarks focus on accomplishing predefined interface tasks rather than synthesizing executable software artifacts such as games.

Game Generation Benchmarks. Recent work has begun to evaluate coding agents on game generation, but existing benchmarks cover different parts of the evaluation contract. OpenGame-Bench (Jiang et al., 2026) asks agents to generate complete games from open-ended prompts, but it targets web games and relies primarily on static or page-level judgment rather than gameplay interaction. GameDevBench (Chi et al., 2026) brings evaluation into Godot, but studies localized tutorial-derived edits within existing projects rather than end-to-end construction of complete

games. Closest to our setting, WebGameBench (Zhang et al., 2026) evaluates delivered browser-native games through real browser interaction; however, it remains outside engine-native game development and depends on evaluation-side exploration to discover gameplay behavior. Concurrent work on GameGen-Verifier (Jia et al., 2026) improves runtime verification through state injection, but does not evaluate open-ended game construction from specifications. In contrast, **GameCraft-Bench** jointly requires agents to build complete Godot projects and to provide replayable demonstrations, enabling interaction-grounded evaluation without turning verification itself into an autonomous game-exploration problem.

B LIMITATIONS

GameCraft-Bench has several scope limitations. It focuses on 2D game generation in Godot, which makes the benchmark lightweight and reproducible for headless evaluation, but does not cover other major engines such as Unity and Unreal. It also does not evaluate 3D games, multiplayer systems, large-scale physics, or long-form production workflows. In addition, the current verifier scores visual gameplay evidence only; audio-dependent aspects of rhythm, horror, or sports games are represented through visual behavior rather than direct audio evaluation.

GameCraft-Bench also has evaluation limitations. It relies on a multimodal judge to score replayed gameplay evidence, so results may still be affected by judge-model bias, API drift, or limitations in visual understanding. Finally, **GameCraft-Bench** does not measure whether a generated game is subjectively fun. Instead, it measures whether the agent follows the game specification and realizes the requested mechanics, content, visual state, and presentation in an executable artifact.

C EVALUATION DETAILS

Runtime Environment. GameCraft-Bench is implemented on Harbor Harbor Framework Team (2026) and runs each trial in a local subprocess sandbox. The reference setup uses Ubuntu 22.04, Python 3.12, and Godot 4.6.2 stable. The verifier depends on Xvfb for headless display, xdotool for synthetic input, and ffmpeg for screen recording and frame extraction. Each task uses an agent timeout of 7200 seconds, a verifier timeout of 1800 seconds. We do not impose an explicit tool-call limit beyond the task timeout. The agent is given a fixed workspace rooted at `/workspace/` and must place the generated Godot project at `/workspace/game`.

Submission Format. Each submission consists of a Godot project and one to ten demonstration trace files named `*.json` under `/workspace/game/demo_outputs/`. Only the first ten traces by filename are evaluated. A trace contains an optional scenario identifier, a total replay duration in frames, and a time-ordered list of mouse or keyboard events in a fixed 1280×720 viewport. Demos can deterministically initialize a requested state such as a battle, upgrade screen, or late-game setup. The trace schema is:

Demo Trace Schema

```
{
  "scenario": "title_flow",
  "duration_frames": 360,
  "events": [
    {"frame": 30,  "type": "mouse_click",
     "button": "left", "x": 300, "y": 360},
    {"frame": 90,  "type": "key_press", "keycode": "1"},
    {"frame": 180, "type": "key_press", "keycode": "SPACE"},
    {"frame": 300, "type": "wait"}
  ]
}
```

Replay and Judge. For each valid trace, the verifier starts a fresh Godot instance, replays input at 30 frames per second, records the 1280×720 display through ffmpeg, and stores a compressed 854×480 version. The verifier samples frames every 0.5 seconds, corresponding to two frames per second. Each rubric may cap the number of demos and the sampled time window; the default benchmark configuration uses at most ten demos and samples at most a deterministic 20-second window per demo. Only the sampled frames, not the full gameplay videos, are supplied to the judge. We use `GPT-5.5` as the multimodal judge. The judge prompt template is shown below:

Judge Prompt Template

```
You are a strict but fair video-game evaluator.  Given sampled frames from a short
playthrough recording of a Godot 2D game, decide how clearly each listed requirement
is demonstrated.  Score every requirement on a 0.0 to 1.0 scale where 0.0 = not
demonstrated at all (or contradicted), 0.5 = partially demonstrated / ambiguous, and
1.0 = clearly and unambiguously demonstrated by what is visible in the sampled frames.
Reply with strict JSON only, no prose, no markdown, no code fences.

Evaluate the sampled frames against each of the following requirements.

Requirements:
{requirements}

Return JSON in exactly this shape (no extra keys, no markdown):

{
  "scores": {"<requirement_id>": <0..1>},
  "rationales": {"<requirement_id>": "<one short sentence>"}
}
```

D Full Family Results

Table 4: Family-level and benchmark-level results (%). For each model, rows report core mechanics (M), content depth (D), functional visuals (V), art and presentation (A), and overall score.

Model	Metric (↑)	Overall	Platformer	Strategy	Tycoon	Open-world	Roguelike	Visual novel	Puzzle	Shooter	Simulation	Card game	Horror	Rhythm	Idle	Racing	Sports
Claude Code																	
Opus-5 xhigh	M	**76.75**	**77.12**	**77.56**	76.46	69.70	**85.85**	67.30	80.21	71.00	77.75	82.08	84.58	**79.17**	96.67	**77.50**	46.67
	D	**63.31**	**57.10**	**79.83**	77.15	43.70	**66.12**	56.30	**61.88**	52.33	**70.25**	54.06	53.75	**63.12**	**84.38**	55.62	48.75
	V	**68.04**	**77.15**	**86.05**	80.64	41.98	**77.67**	67.70	79.17	68.96	74.06	42.25	49.38	**52.42**	**43.75**	**63.50**	39.17
	A	**70.11**	**65.10**	**71.37**	74.72	64.39	66.78	**71.43**	75.56	**75.46**	**60.94**	62.19	73.94	**80.69**	**79.88**	**78.50**	54.08
	Overall	**68.44**	**65.92**	**77.44**	76.72	54.59	**71.01**	**65.33**	**72.01**	65.70	68.70	59.35	64.78	**70.05**	**78.56**	**68.06**	48.84
Fable-5 high	M	76.52	69.69	71.00	**86.20**	64.43	80.60	**78.50**	**93.80**	**78.00**	**98.00**	**96.70**	**89.25**	68.35	**100.00**	75.00	61.70
	D	58.61	37.44	67.60	**87.70**	45.07	62.30	**63.50**	38.80	**67.00**	70.00	**80.00**	**55.35**	48.80	67.50	51.90	56.30
	V	66.93	69.53	85.00	**86.46**	30.50	68.95	**70.10**	**84.20**	**77.10**	**77.80**	**68.10**	**66.40**	46.80	40.00	32.30	50.00
	A	67.68	59.24	60.74	**77.72**	**70.43**	**71.60**	57.65	**77.10**	71.40	55.90	**71.80**	**78.30**	75.45	78.60	69.80	**79.80**
	Overall	65.72	54.74	68.30	**83.78**	54.67	69.30	64.70	67.20	**71.70**	**70.50**	**77.80**	**70.13**	60.70	72.10	58.70	**64.40**
Opus-4.7 high	M	55.34	45.77	54.91	48.31	47.88	64.87	56.33	51.43	55.00	61.17	54.67	72.67	62.00	76.67	55.83	**66.25**
	D	39.48	35.50	46.78	39.42	34.43	38.79	42.22	26.43	40.57	37.33	26.25	49.00	40.00	61.25	45.31	40.62
	V	42.78	46.28	62.60	45.11	29.57	46.50	50.99	43.36	38.20	44.13	18.48	50.50	24.70	23.12	23.15	39.38
	A	36.86	29.55	30.65	24.69	48.94	28.05	35.21	35.44	34.58	26.93	38.92	61.92	55.90	58.82	50.15	51.95
	Overall	41.46	36.57	44.73	36.45	40.80	40.10	43.60	35.87	40.28	38.29	33.78	57.30	46.57	56.99	45.26	48.24
DeepSeek-V4-Flash-0731	M	52.11	39.51	46.47	66.02	49.72	50.54	54.00	53.59	60.33	57.80	48.33	76.67	42.08	83.33	34.17	13.33
	D	37.14	28.53	38.47	53.43	31.19	31.58	37.52	21.41	45.83	49.40	30.31	53.00	23.12	67.50	31.87	18.75
	V	43.70	39.61	54.24	60.02	28.57	50.92	47.62	40.12	47.46	52.15	16.67	53.00	25.75	13.33	23.92	12.50
	A	37.72	34.60	35.69	41.24	39.08	32.42	29.15	24.54	49.00	38.70	41.44	61.95	32.94	64.67	36.25	58.00
	Overall	40.61	33.97	41.06	52.02	36.33	37.60	39.06	30.13	49.39	47.33	34.89	59.68	29.75	60.74	32.55	30.74
GLM-5.2	M	50.09	49.17	42.72	57.30	39.53	58.94	56.00	54.64	52.86	52.00	52.33	45.00	40.00	62.50	32.50	51.25
	D	36.40	30.19	37.34	47.62	31.46	35.92	43.44	26.61	35.43	43.17	33.75	31.75	22.75	46.56	22.81	51.88
	V	41.06	44.38	48.48	55.60	26.26	52.36	48.05	41.62	47.04	43.96	19.07	29.00	18.53	28.12	16.67	28.12
	A	36.29	33.88	24.02	38.43	43.67	32.97	32.25	35.00	46.04	32.42	39.30	53.90	31.70	42.38	33.69	52.44
	Overall	39.12	36.44	35.15	47.03	36.15	40.80	42.31	35.99	43.47	40.81	36.32	41.07	27.82	44.72	27.15	48.42
MiMo-V2.5-Pro	M	32.33	32.58	28.81	41.34	17.22	34.13	27.64	33.12	40.29	37.17	26.67	45.67	31.00	50.83	23.75	32.92
	D	22.59	17.50	20.11	36.31	17.08	21.27	21.02	21.09	19.00	28.17	12.25	26.00	13.75	41.25	16.56	38.75
	V	27.45	34.41	29.41	39.42	11.81	35.85	29.63	32.02	30.10	30.59	4.95	28.00	16.34	11.88	9.51	19.38
	A	20.65	18.46	14.12	19.53	22.96	18.19	15.37	21.21	25.02	20.53	14.59	41.14	16.69	42.94	24.57	29.29
	Overall	24.10	22.64	20.72	31.66	18.37	24.31	21.36	24.58	25.96	27.21	14.14	34.55	17.76	38.87	19.39	31.66
Codex																	
GPT-5.6-sol high	M	74.50	69.00	73.00	76.00	**72.00**	76.00	74.00	72.00	77.00	77.00	88.00	82.00	72.00	93.00	69.00	65.00
	D	56.10	49.00	63.00	68.00	**49.00**	53.00	57.00	49.00	56.00	56.00	55.00	53.00	46.00	67.00	**59.00**	**65.00**
	V	64.80	74.00	80.00	78.00	**49.00**	70.00	67.00	66.00	66.00	65.00	39.00	62.00	49.00	23.00	41.00	**54.00**
	A	57.00	57.00	60.00	55.00	58.00	49.00	43.00	55.00	62.00	51.00	61.00	72.00	69.00	72.00	58.00	67.00
	Overall	60.50	58.00	66.00	66.00	**56.00**	57.00	56.00	57.00	63.00	59.00	60.00	65.00	58.00	66.00	57.00	64.00
GPT-5.5 high	M	54.36	50.51	50.96	42.64	54.48	50.99	56.00	67.97	64.00	53.83	50.67	59.67	57.33	90.83	57.50	52.08
	D	38.61	34.66	40.69	36.85	39.23	32.60	45.61	37.97	33.71	37.50	23.25	36.50	40.75	72.19	35.00	55.62
	V	41.84	45.49	56.61	41.01	27.29	43.42	50.46	53.34	44.89	40.10	17.73	44.00	32.84	28.12	25.23	33.75
	A	32.94	30.65	26.94	17.45	40.51	25.65	28.50	43.75	37.62	33.11	20.29	59.20	42.45	62.29	35.69	49.61
	Overall	39.49	37.26	39.81	31.55	40.17	34.55	42.15	46.80	41.30	38.81	25.50	49.05	42.65	64.91	37.15	49.71
DeepSeek-V4-Pro	M	2.25	0.39	1.35	0.00	0.92	1.71	0.45	0.00	0.00	6.50	0.00	7.33	0.00	33.33	0.00	8.33
	D	1.69	1.00	0.35	0.00	0.00	1.00	0.82	0.00	0.00	6.33	0.00	5.00	0.00	13.44	0.00	18.12
	V	1.97	2.67	0.88	0.00	1.11	3.48	2.56	0.00	0.00	5.62	0.00	2.00	0.00	6.88	0.00	11.25
	A	2.63	2.73	0.59	0.00	0.28	2.75	0.91	0.00	0.00	4.93	7.00	10.40	4.83	17.92	0.00	10.38
	Overall	2.15	1.77	0.66	0.00	0.40	2.09	1.06	0.00	0.00	5.76	2.45	6.79	1.69	17.01	0.00	12.91
Kimi Code																	
Kimi-K2.6	M	39.76	34.32	47.65	38.50	26.99	52.31	45.73	30.47	43.00	47.50	29.00	34.67	41.00	64.58	20.00	39.17
	D	28.07	22.71	32.22	32.04	23.10	33.22	40.09	18.12	30.43	40.83	11.50	17.50	14.50	38.75	8.12	44.38
	V	33.66	39.27	44.17	39.30	19.05	43.30	46.49	35.51	29.65	38.42	10.57	19.00	22.91	15.62	8.33	25.62
	A	27.99	28.43	21.23	22.60	24.62	26.27	27.75	26.23	34.61	25.60	24.33	42.19	31.84	45.87	36.32	46.74
	Overall	30.65	28.94	32.48	30.79	23.61	35.17	37.85	25.42	33.66	36.14	18.48	28.94	25.80	41.65	19.81	41.61
Code Buddy																	
GLM-5.1	M	25.23	22.37	19.12	27.25	22.53	30.09	21.45	35.62	24.29	42.67	18.67	25.00	15.33	40.00	33.33	12.92
	D	17.80	11.95	18.97	22.21	16.90	19.06	16.27	20.94	16.29	30.00	14.25	10.50	14.25	24.06	8.12	25.62
	V	21.14	22.35	25.85	26.40	11.75	25.12	24.33	29.49	20.37	34.58	7.91	13.00	12.13	11.25	7.86	12.50
	A	14.59	11.99	9.19	9.01	17.26	13.35	12.26	18.91	15.26	20.17	14.01	36.68	11.76	26.57	14.05	19.97
	Overall	18.29	15.09	16.60	18.98	17.10	19.62	16.88	23.71	17.74	29.15	13.88	22.21	13.22	25.41	13.94	19.77
MiniMax-M2.7	M	14.27	9.96	13.24	20.69	2.98	21.00	12.91	4.53	21.14	30.50	16.33	17.67	9.00	22.92	14.58	10.00
	D	9.92	5.00	10.51	14.56	4.98	11.72	12.18	5.78	11.86	20.17	3.00	12.25	11.00	11.88	9.69	10.31
	V	14.92	15.87	17.84	25.63	6.67	20.48	16.89	16.51	17.15	29.12	1.20	2.00	4.67	3.75	2.60	2.50
	A	8.85	6.37	4.13	7.48	9.34	6.84	5.77	3.04	13.29	15.25	10.31	25.19	14.24	21.79	12.00	8.94
	Overall	10.95	7.85	9.79	14.66	6.46	12.72	10.82	6.24	14.55	21.34	7.29	16.05	10.88	15.78	10.17	8.61

E Additional Analyses

E.1 Execution and Replay Diagnostics

Figure 9 separates project launchability from the availability and amount of replayable gameplay evidence for the seven configurations with execution logs.

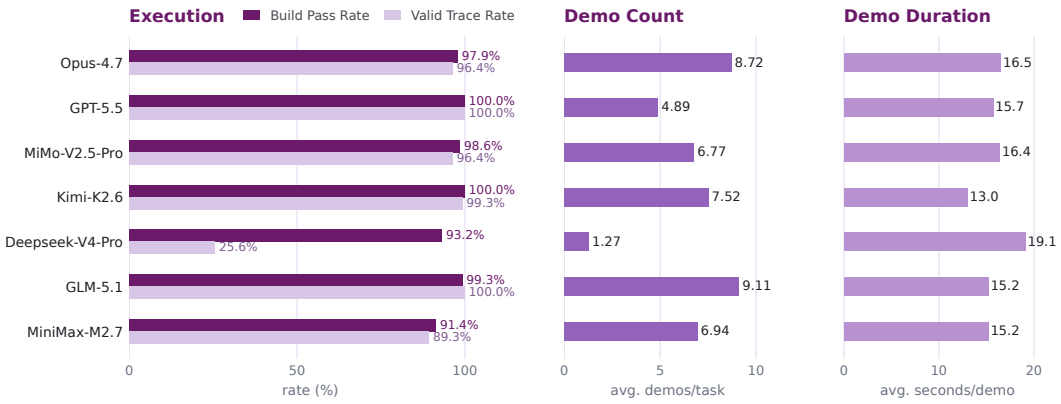


Figure 9: Execution and replay statistics for the seven configurations with available execution logs. Build pass measures launchability, valid trace measures whether at least one submitted demonstration can be replayed, and demo count/duration summarize the amount of gameplay evidence available for judging.

E.2 MiMo-V2.5-Pro Task-Completion Failure Pattern

Writing more code or running more commands is not sufficient for producing a playable game. MiMo-V2.5-Pro exhibits a characteristic *write-first, bash-heavy* pattern: the agent rapidly scaffolds a complete file set, including `project.godot`, GDScript sources, and scene files, before any validation. It then enters an extended execution-driven debug phase dominated by shell commands, which account for 56.3% of all tool calls across 140 tasks, while code-reading and editing operations (`Read` + `Edit`) account for only 16.5%. Crucially, total tool usage is decoupled from task quality, as shown in Figure 10: a typical task involves a median of 128 tool calls, yet additional effort in the bash debug loop does not translate to a higher score. The bottleneck is structural rather than raw effort: of the five zero-score tasks, all produced a valid build but submitted no demo traces, leaving every rubric dimension unscored. The write-first strategy front-loads code generation but frequently misses the demo-trace delivery that closes the evaluation loop, revealing a task-completion gap orthogonal to raw coding ability.

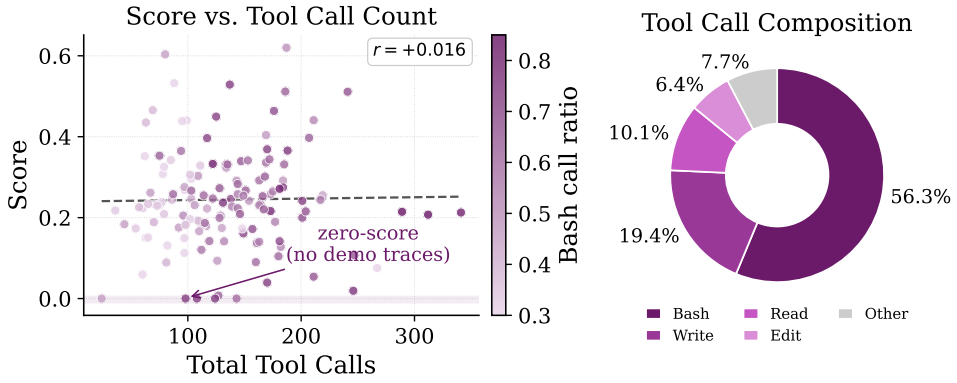


Figure 10: Tool usage of MiMo-V2.5-Pro across 140 tasks. ***Left*:** Score versus total tool calls, colored by the per-task `Bash` fraction; the near-zero correlation ($r = +0.016$) indicates that tool usage volume is decoupled from task quality. ***Right*:** Aggregate tool-call composition; shell execution dominates at 56.3%, while code reading and editing account for only 16.5%.

Findings 4.

Execution effort and tool-usage volume are weak predictors of final playability; success depends on closing the full build–replay–evaluation loop.

E.3 Human Calibration

Five annotators received each task rubric and the submitted demos and could additionally launch and play the generated game. Rather than scoring demos separately, each annotator assigned one score to every rubric item after reviewing the available evidence; human category scores average these item-level ratings within each category and across annotators. This comparison therefore tests whether frame-based judge scores approximate a richer human artifact assessment, rather than agreement under identical evidence. Table 5 reports the resulting calibration breakdown for Kimi-K2.6 submissions from Card Game, Idle, and Racing.

Table 5: Preliminary human calibration on Kimi-K2.6 submissions from three families. Human scores aggregate five annotators' item-level ratings; Δ columns report Judge–Human differences in percentage points.

Family	Human	Judge	ΔM	ΔD	ΔV	ΔA	ΔOverall
Card Game	18.75	18.48	+2.33	+1.50	-6.10	-0.67	-0.27
Idle	32.89	41.65	+6.25	+12.19	-3.12	+11.50	+8.76
Racing	17.42	19.81	-2.92	+3.44	-8.33	+8.20	+2.38
Overall	22.69	26.02	+1.92	+5.38	-5.87	+5.80	+3.32

E.4 Decomposability of Game Generation Ability

We examine whether the four rubric categories rise and fall together across Kimi-K2.6 and MiMo-V2.5-Pro tasks. As shown in Figure 11, Core Mechanics correlates moderately with Content Depth ($r = 0.61$) and Functional Visuals ($r = 0.53$) for Kimi-K2.6, suggesting that stronger interaction loops often expose more game state and feedback. However, Art and Presentation is much less coupled with the other categories, especially Functional Visuals ($r = 0.11$). MiMo-V2.5-Pro shows a similar but more globally coupled pattern, with Art and Presentation correlating with Mechanics, Content Depth, and Functional Visuals at $r = 0.56$, $r = 0.39$, and $r = 0.26$, respectively. This supports the main-result observation that a mechanically recognizable game does not automatically become a complete, visually coherent, or polished game artifact.

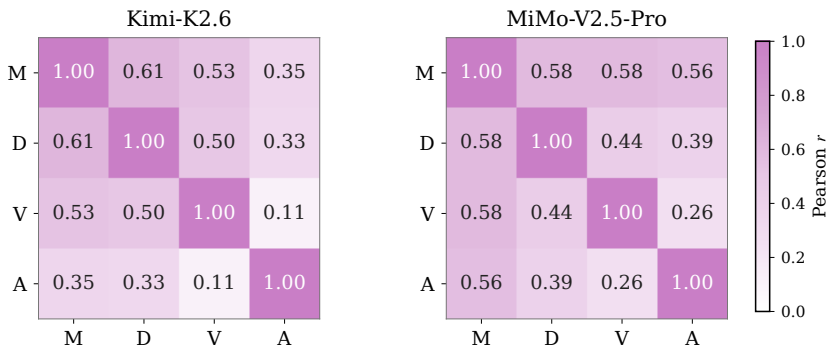


Figure 11: Correlation among rubric categories for Kimi-K2.6 and MiMo-V2.5-Pro. Mechanics, Depth, and Visuals are moderately coupled, while Art and Presentation is more weakly coupled with the other categories.

Findings 5.

Game generation capability is partially factorized: mechanics, content, visual feedback, and presentation are not fully coupled.

F Annotation Process and Quality Control

Annotation Process. We use Harbor Harbor Framework Team (2026) as the task-authoring framework. The task suite is annotated by 12 annotators, each selected for extensive gameplay experience across diverse genres. For each task, an annotator authors two aligned artifacts: a public product-style game specification that describes the player fantasy, core loop, mechanics, and art direction; and a hidden evaluation rubric that decomposes the same game into observable, requirement-level criteria. The annotator ensures that the specification is natural and open-ended—specifying the intended experience without prescribing implementation details—while the rubric is precise enough to be scored from replayed gameplay evidence. The instruction given to annotators is summarized below.

Instruction for Annotators

Annotate each GameCraft-Bench task into three files: task.toml, instruction.md, and tests/rubric.json. The final goal is to evaluate whether an agent can build a complete, shippable micro-game, not a prototype, static UI mockup, or label-only demo.
1. **Metadata.** Use task.toml only for metadata and runtime configuration, such as the task name, short description, keywords, verifier settings, agent budget, and environment settings. Do not put gameplay requirements, demo rules, or rubric items in task.toml.
2. **Game specification.** Use instruction.md as the source of truth for what the agent must build. The opening should name the game, specify the output path /workspace/game, and set the quality target as a complete, shippable micro-game. Write a *Core Vision* section describing the player fantasy and core loop, and a *What the Player Experiences* section describing the player-facing arc. The specification should describe observable player experience, meaningful decisions, feedback, variation, and outcomes, while avoiding rubric-style checklists or implementation-level commands. Include standard sections for assets, project layout, launch command, demos, scenarios, and trace format. Do not ask for audio requirements, because the verifier scores visual evidence.
3. **Evaluation rubric.** Use tests/rubric.json to translate the game specification into concrete rubric items for the verifier. Each rubric should include score_formula, max_demos, max_demo_seconds, build_check, categories, and requirements. Use four fixed categories: Core Mechanics, Content Depth, Functional Visuals, and Art & Presentation. Use stable item prefixes: M*, D*, V*, and A*. Keep the total number of rubric items at 24 or fewer.
4. **Category calibration.** Core Mechanics should evaluate whether the irreducible gameplay loop works through player action, state change, rules, consequences, goals, and failure. Content Depth should evaluate varied material, progression, late-game states, or multiple scenarios. Functional Visuals should evaluate readability and feedback during play, including readable state, warnings, objectives, and stable UI at 1280×720. Presentation & Art should evaluate whether the game feels authored and publishable rather than debug-like. Visual style, asset quality, and polish should not affect Mechanics, Depth, or Functional Visuals unless they prevent the relevant behavior, content, or feedback from being observed; these qualities are evaluated only under Art & Presentation.
5. **Scoring calibration and aggregation.** Score each item only against its category. For Mechanics and Depth, simple visual encodings do not reduce credit when functional behavior and content differences are clearly observable. Functional Visuals evaluates clarity and feedback rather than asset or style quality. Reserve asset quality, stylistic coherence, and polish for Art & Presentation, where debug-like default widgets or raw programmatic shapes may cap credit at 0.5. For Mechanics, Depth, and Functional Visuals, cap claimed functional evidence at 0.5 when it is merely label-only, decorative, static, or disconnected from gameplay. Use mean for properties that should hold across demos, such as readability, visual feedback, art consistency, UI overlap, and presentation quality. Use max only when observing a feature once is enough to prove it exists, and use it conservatively to avoid over-crediting static demonstrations.

Quality Control. After the specification and rubric are drafted, annotators enter a validation phase by writing a simple oracle solution—a minimal playable sketch in Godot—to cross-check the task's internal consistency. The oracle serves three validation functions: it verifies whether the specification is implementable in the engine, whether the requested behavior can be demonstrated through replayed gameplay, and whether every rubric item corresponds to an observable state rather than a subjective preference. If the oracle cannot reach the intended gameplay state, or if a rubric item lacks supporting evidence in the replay, the annotator revises the specification and rubric and re-validates until the task becomes internally consistent. This specification–rubric–oracle alignment ensures that the agent faces a well-defined target, the judge evaluates only executable behavior, and the final score reflects whether the generated artifact realizes the intended player experience.

G CASE STUDY

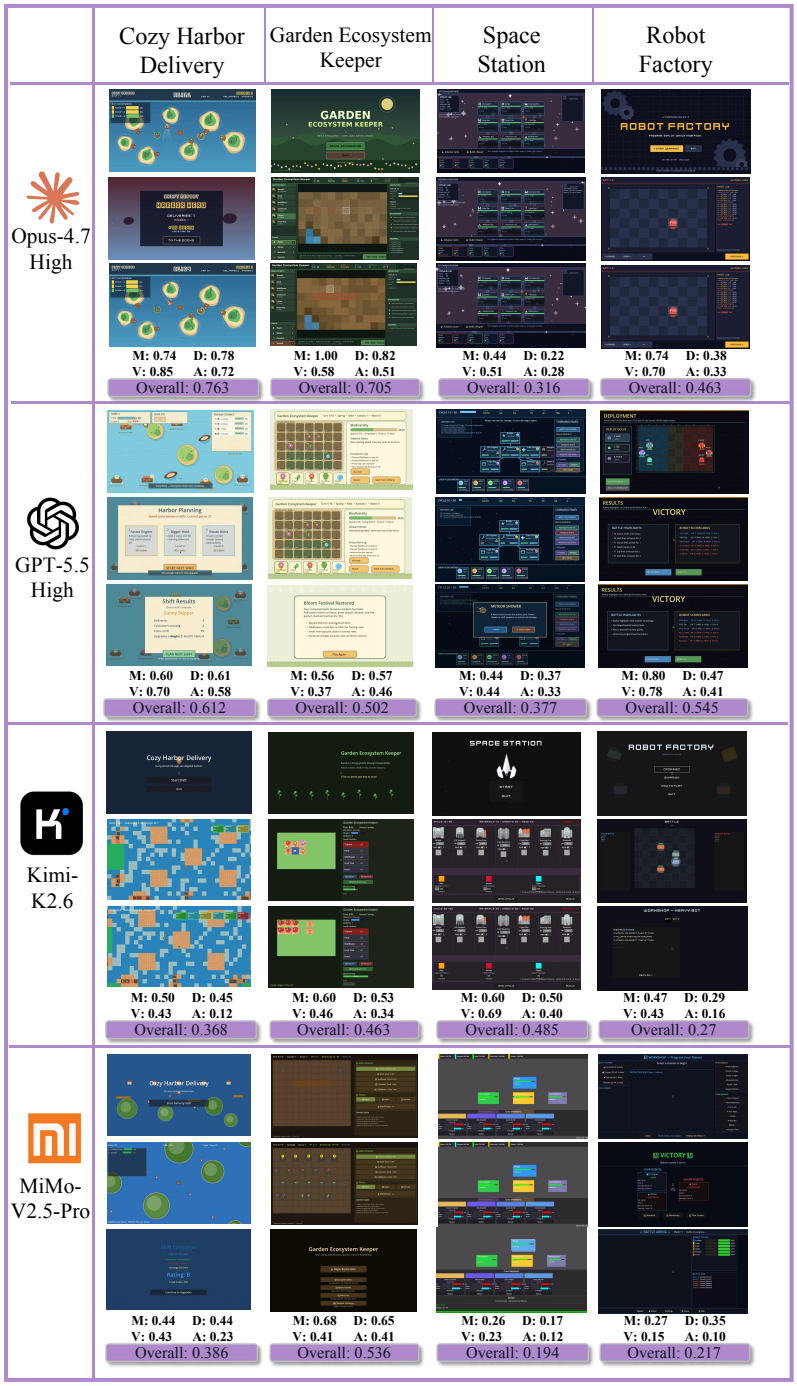


Figure 12: Case Study: Four Models on Four Representative Tasks. Each cell shows three sampled gameplay frames from the agent's submitted demo traces, together with per-category scores (*M*: Core Mechanics, *D*: Content Depth, *V*: Functional Visuals, *A*: Art & Presentation), and the overall score shown below.

Example `rubric.json` of Strategy-Skirmish

```
  "score_formula": "BUILD * (0.15*((M1+M2+M3+M4)/4) + 0.35*((D1+D2+D3+D4)/4)
            + 0.15*((V1+V2+V3+V4)/4) + 0.35*((A1+A2+A3+A4)/4))",
  "max_demos": 10, "max_demo_seconds": 20,
  "build_check": {
  "id": "BUILD", "cmd": "godot --headless --path /workspace/game --quit-after 5" },
  "categories": [
    {"name": "Core Mechanics", "items": ["M1", "M2", "M3", "M4"]},
    {"name": "Content Depth", "items": ["D1", "D2", "D3", "D4"]},
    {"name": "Functional Visuals", "items": ["V1", "V2", "V3", "V4"]},
    {"name": "Art & Presentation", "items": ["A1", "A2", "A3", "A4"]} ],
  "requirements": [
    {
      "id": "M1", "agg": "max",
      "description": "The irreducible tactical loop is playable: the player can select a
      friendly unit, see a limited legal movement range highlighted on the grid, choose a
      legal destination, and watch the unit move to the chosen cell. Illegal clicks
      outside the range are rejected without corrupting state. Selecting a different unit
      or clicking the same unit again cancels the current selection. Score 0 if selecting
      and moving a unit is missing, scripted-only, or not controlled by player input.
      Score 1 requires the full loop and edge cases above to work under player input.
      Score 0.5 at most if selection or movement works only partially or rule enforcement
      is ambiguous. Visual style and asset quality are not scored under this item."
    },
    ...
    {
      "id": "D1", "agg": "max",
      "description": "The player's squad has meaningful depth: at least three controllable unit
      types or named characters are present, visually identifiable, and functionally different
      through HP, movement range, attack range, damage, or special ability. Score 1 only if the
      differences change tactical choices during battle. Score 0 if there is only one player
      unit type or if all units share identical stats and behavior. Score 0.5 at most if
      only labels or decorative differences distinguish units with no observable gameplay impact.
      Score 1 requires at least three variants whose functional differences visibly change
      tactical choices. Visual style and asset quality are not scored under this item."
    },
    ...
    {
      "id": "V1", "agg": "mean",
      "description": "Battle readability is strong throughout gameplay: grid cells, walkable
      terrain, blocked terrain, friendly units, enemy units, selected unit, legal move
      cells, and attackable targets are all visually distinct at 1280x720. Score 1 only
      if a player can understand legal actions without guessing. Score 0 if key states
      blend together, overlays are missing, or placeholder shapes make sides and zones
      ambiguous. Score 0.5 at most if feedback is incomplete, inconsistent, or requires
      guessing. Visual encoding may be simple if it is unambiguous and consistent; asset
      quality and artistic style are not scored under this item."
    },
    ...
    {
      "id": "A1", "agg": "mean",
      "description": "The game has a coherent dark-fantasy art direction: terrain, units, UI,
      effects, and backgrounds share one consistent visual language with a deliberate
      limited palette and high contrast. Score 1 only if the presentation looks like an
      authored vertical slice rather than a prototype. Score 0 if visible gameplay relies
      on raw ColorRect shapes, default Godot widgets, mismatched asset packs, or bright
      off-brief placeholder colors."
    },
    ...
  ]
```

Example `instruction.md` of Strategy-Skirmish

```
# Strategy: Skirmish

Build a **dark-fantasy tactical skirmish** in Godot 4 at `/workspace/game/`.
This is not a prototype. It is a **complete, shippable micro-game** that could
sit on an itch.io page or Steam as a polished vertical slice.

## Core Vision

The player commands a small, outnumbered squad through desperate grid-based
battles where every move is a commitment and every loss is permanent for the
fight. The fantasy is **grim tactical survival** | a handful of specialists
against a tide of enemies, where positioning is life and a single misread costs
a unit you cannot replace mid-battle. The tone channels *Into the Breach* meets
*Darkest Dungeon* at a smaller scale: limited palette, high contrast, tense
decisions. The best version makes the player feel like a cornered general
finding the one sequence of moves that turns impossible odds into a narrow
victory.

## What the Player Experiences

A moody title screen sets the dark-fantasy tone immediately. The player begins
and receives a brief tactical briefing | the squad's objective, the threat
ahead, the stakes | before the grid appears.
```

```
The battle is turn-based and deliberate. The player selects a unit, sees its
limited movement range light up on the grid, and commits it to a position.
Enemies are visible, aggressive, and numerous | the squad is always
outnumbered. After the player spends their actions, an End Turn command hands
control to the enemy, which advances with purpose: flanking, closing distance,
attacking when in range. Then control returns and the cycle repeats.

Combat is lethal and readable. Attacks require proximity or a clear range
indicator, reduce persistent HP, and kill. Dead units vanish from the board and
stop blocking or threatening. The player's squad members are specialists |
different movement ranges, attack patterns, HP pools, or abilities | so
choosing who moves where and who attacks what is the core decision space.

The battlefield itself adds tactical texture: terrain obstacles funnel movement,
hazards punish careless positioning, or objectives create time pressure beyond
simple elimination. Multiple battle layouts keep the experience from feeling
solved after one fight.

Victory comes from eliminating all enemies; defeat from losing the squad. Either
outcome lands on a styled result screen showing what happened, and the player
can retry or return to the title without relaunching. The entire arc | title,
briefing, battle, result | flows as one continuous authored experience.

{Resource Instruction}

{Foramt Instruction}
```

Resource Instruction in `instruction.md`

```
## Assets

2D assets are mounted read-only at:

- `/workspace/assets/library/` | Kenney CC0 packs (sprites, tiles, UI, fonts).
- `/workspace/assets/library-oga/` | OpenGameArt entries; respect each
  subdir's `LICENSE.txt`.

Browse the library and choose packs.
Copy what you need into your project's `assets/` folder.

## Project layout

```
/workspace/game/
  project.godot
  Main.tscn
  demo_outputs/    ← your input traces (1{10 files)
  scripts/  scenes/  assets/
```

The build must launch cleanly with:

```
godot --headless --path /workspace/game --quit-after 5
```

A reference for Godot CLI flags is at `/workspace/tools/godot_command_line.md`.
Engine flags like `--headless` and `--quit-after N` must come BEFORE `--` |
anything after `--` is forwarded to the project as user args and silently
ignored by the engine. Correct shape:
`godot --headless --quit-after 5 --path . -- --scenario near_victory`.

A screenshot helper is available at `/workspace/tools/screenshot.sh`.
Use it to actually see what your UI / battlefield /
result screens look like.

```
/workspace/tools/screenshot.sh --path /workspace/game \
      -- --out /workspace/frame.png --frames 60
```

To screenshot a specific scenario, append `--scenario <id>` after `--`. The
helper consumes only `--out` / `--frames` / `--scene`; remaining args stay in
`OS.get_cmdline_user_args()` for your game code to read. Example:

```
/workspace/tools/screenshot.sh --path /workspace/game \
      -- --out /workspace/battle_debug.png --frames 120 --scenario battle
```
```

Format Instruction in `instruction.md`

```
## Demos

Ship **1{10 input-trace files** under `/workspace/game/demo_outputs/`, one per
demo, each named `*.json`. The evaluator launches a fresh game per trace,
replays your trace as synthetic mouse and keyboard input at 1280×720, and
```

````
records the screen. Only the first 10 traces by filename are evaluated;
recordings longer than 20 s are sampled from a deterministic 20 s window.

### Scenarios

Normal play should start from the title screen and demonstrate the task's
core gameplay loop.
Demo playback must be deterministic. For demos that need a specific state
(a specific level, combat state, upgrade screen, result state, or late-game
setup), define named scenarios your game loads when launched with:

```
godot --path /workspace/game -- --scenario <id>
```

When `--scenario <id>` is present the game must skip menus, set up the named
state deterministically (seed any RNG), and begin accepting input immediately.

### Trace file format

```json
{
  "scenario": "title_flow",
  "duration_frames": 360,
  "events": [
    {"frame": 30,  "type": "mouse_click", "button": "left", "x": 300, "y": 360},
    {"frame": 90,  "type": "key_press",   "keycode": "1"},
    {"frame": 180, "type": "key_press",   "keycode": "SPACE"},
    {"frame": 300, "type": "wait"}
  ]
}
```

- `scenario` | optional; omit for a normal game launch from the title screen.
- `duration_frames` | total frames to record at 30 fps; cap at **600 (20 s)**.
- `events` | time-ordered inputs. Coordinates are pixels in the 1280×720
  viewport. Supported types:
  - `mouse_click`: `{frame, type, button: "left"|"right", x, y}`
  - `mouse_down` / `mouse_up`: `{frame, type, button: "left"|"right", x, y}` |
    use these for drag interactions: emit `mouse_down` at the start point,
    one or more `mouse_move` events along the way, and `mouse_up` at the end.
    A `mouse_click` is a `mouse_down` + `mouse_up` at the same point in tight
    succession.
  - `mouse_move`: `{frame, type, x, y}`
  - `key_press` / `key_down` / `key_up`: `{frame, type, keycode}` | keycodes:
    `A`–`Z`, `0`–`9`, `ESCAPE`, `ENTER`, `SPACE`, `TAB`, `BACKSPACE`,
    `DELETE`, `SHIFT`, `CTRL`, `ALT`, `UP`, `DOWN`, `LEFT`, `RIGHT`.
  - `wait`: `{frame, type}` | anchor frame, no input.

Replay must be deterministic: same trace, fresh launch, same outcome every time.
````