GAME-EVOLVER: DUAL-LOOP EVOLUTION ELICITS BETTER END-TO-END GAME ORCHESTRATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent coding agents have made significant strides in tackling complex tasks across various domains. However, even frontier agents can solve only a minority of game generation tasks, owing to the extreme long-horizon requirements from building games and interacting in it. Moreover, most agent frameworks rarely provide comprehensive, holistic workflows specifically designed for game making tasks, preventing their further applications. These factors jointly pose a unique challenge in the context of agent game development.

We address this challenge with Game-Evolver, a dual-loop framework that couples two agents. A game orchestration agent (GOA) builds the game; a harness proposal agent (HPA) proposes bounded changes to the GOA's harness. In the inner loop, the GOA A/B-tests each proposed harness against the best candidate so far. A harness is admitted only when the new harness passes dual-rubric verification, where hard rubrics enforce the game legality and soft rubrics measure its quality. In the outer loop, the HPA distills the GOA's completed runs into recurring failure modes, and uses them to update its own harness library. Game-Evolver thus improves both the game-making and the agent that generates it.

Extensive experiments on four game benchmarks and six backbone models show that Game-Evolver improves every baseline, delivering several-fold gains on weaker models and up to 30% on the strongest. Our approach also transfers to general agent tasks, while showing impressive applicability to real game projects.

1 INTRODUCTION

Recent language agents have moved from producing answers to interacting in dynamic environments (Zhang et al., 2026a; Ye et al., 2026; Jiang et al., 2026a; Wang et al., 2026). In games, this capability has been applied along two interleaved lines: ❶ game-playing: agents act inside an environment whose rules, state, and reward are already defined (Mnih et al., 2015; Silver et al., 2016; Vinyals et al., 2019); ❷ game-generation: agents operate workspaces and engines to construct the environment itself (Agashe et al., 2025; Lai et al., 2026; Jiang et al., 2026b; Yin et al., 2026; Huang et al., 2026). Consequently, Game tasks impose extremely stringent demands on an agent's ability to handle long-horizon tasks. Benchmark results confirm that this difficulty is real: even the strongest agent reaches only 41.46% on GameCraft-Bench (Luo et al., 2026) across 15 game families, and most agents score below 20%. GameDev-Bench (Chi et al., 2026) and OpenGame-Bench (Jiang et al., 2026b) report a similar pattern. Therefore, given the same level of model capability, the bottleneck lies precisely in how the agent organizes its work under a heavy interactive runtime.

This bottleneck produces two requirements for the agent harness (i.e., all the equipments surrounding the LLM that makes it an agent). ① *The ability of, tackling even more long-horizon, complex tasks.* The agent must produce and interact with a runnable game, and any missing checks or uncontrolled operations can result in a broken scene. ② *The ability of, evolution.* Game development itself is an incremental process (Jiang et al., 2026b), and the same holds true for game making agents. This is because game quality has no golden reference, and the agent itself determines the upper bound of the generated games. Under the same time budget, primary harness produces only basic games on average, while evolving the harness pushes that upper bound higher and higher. We then conclude our pursuits as the following:

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



Figure 1: Game-Evolver couples a game orchestration agent (GOA) and a harness proposal agent (HPA). A GOA creates runnable games while receiving iterative updates from the harness proposed by the HPA, and the HPA generates new harness while using the GOA's trajectory to refine itself.

❶ *A stable, lightweight interface.* Existing RSI mostly relies a light and predefined interface, such as a compiler or test runner. However, game generation instead runs against a heavy runtime built by itself. The harness to evolve must therefore orchestrate and run that runtime precisely.

❷ *A delicate criteria.* Coding and tool-use RSI is judged by hidden tests or an explicit checker. Games keep such oracles only for hard legality, and quality above that gate is subjective and has no single oracle. Game generation therefore needs fixed alongside task-adaptive rubrics.

❸ *Cheap attribution.* Knowing whether a gain came from the harness or from randomness needs a controlled comparison (Lin et al., 2026b). Games make this harder: every evaluation is a full interactive run whose outcome can vary, so repeating it many times to average out the noise is too expensive. RSI must therefore run parent and child under identical conditions a few times.

To deal with these three issues, we introduce Game-Evolver, which couples artifact execution to harness evolution. It employs two evolution loops between a game orchestration agent (GOA) and a harness proposal agent (HPA). As shown in Figure 1, in the *inner loop*, the GOA improves the game artifact while the harness stays frozen; when the HPA proposes a bounded harness change, the loop re-runs the same cases under the child harness and admits it only if the A/B test passes dual-rubric verification. In the *outer loop*, the HPA distills the GOA's completed runs into recurring failure families and uses them to dynamically update the reusable element library. The element libraray is also dynamically maintained to ensure that inefficient or redundant harness elements can be pruned. Our contributions can be summarized as the following:

- We treat game generation as a budgeted orchestration problem, and separate the evolution of the game artifact from the evolution of the harness that builds it.
- We introduce the first self-evolution method for game-gen tasks named Game-Evolver: the inner loop improves the harness by A/B-testing and dual-rubric verification; the outer loop turns recurring failures into updates of the element library.
- Extensive experiments show that Game-Evolver improves over each baseline, with several-fold gains on weaker models up to 30% on the strongest and measurable transfer to general tasks.

2 Related Work

Game AI and Game Agents. The classic game-AI treats a game as a fixed environment. That means, its rules, action space, and success signal are given, and the agent only learns behavior

within them. Deep Q-learning made this concrete on Atari by mapping observations to actions under delayed reward (Mnih et al., 2015). AlphaGo (Silver et al., 2016; 2017) and AlphaStar (Vinyals et al., 2019) scaled the same formulation with search and self-play. Later work (Baker et al., 2022; Fan et al., 2022) like Voyager (Wang et al., 2024a) relaxed the assumption that behavior must be a single reactive policy; they learn reusable behavior and skills from large-scale gameplay, Generative Agents persist in a simulated social world (Park et al., 2023), and Cradle (Tan et al., 2024) and MobileAgent-v2 (Wang et al., 2024b) run the same plan–act–observe loop in graphical and device-control settings. In all of these, the environment is assumed to exist before evaluation starts.

Self-Evolution of Agent Harness. A second line changes the agent harness instead of weights, including the roles, tool interfaces, and validation rules that surround the model. Multi-agent systems such as CAMEL (Li et al., 2023), AutoGen (Wu et al., 2023), MetaGPT (Hong et al., 2024), and ChatDev (Qian et al., 2024) organize agents through specialized roles and handoffs, while Multi-Agent Debate makes the resulting execution graph explicit (Du et al., 2024). Recent harness-centric work (Wang et al., 2025; Zhang et al., 2025), including (Ning et al., 2026), Meta-harness (Lee et al., 2026), and Observability-Driven Automatic Evolution (Lin et al., 2026a) treats the wrapper as a measurable and adjustable search object rather than a fixed implementation detail; AgentX (Lao et al., 2026) pushes the same direction further. Code and computer-use benchmarks supply the external execution signals needed to judge such changes (Jimenez et al., 2024; Yang et al., 2024; Zhou et al., 2024; Koh et al., 2024). Game-Evolver belongs to this second line. It co-evolves the harness that inspects an executable game artifact and propose new harness changes.

3 Problem Setting

3.1 Task Definition

We formalize the general task as constructing a runnable game project from a task specification and an initial project state. Let $x \sim \mathcal{D}$ denote a task with natural-language requirements r_x, an initial project $p_{0,x}$, an engine/runtime configuration γ_x, and a task-specific evaluation protocol $\mathcal{E}_x$. The project comprises code, content, interaction bindings, and configuration, and the finished project must be valid under the engine. We thus define the target as optimizing the game as the following:

$$p_x^\star \in \arg\max_{p \in \mathcal{P}(p_{0,x}, \gamma_x)} S_x(p) \quad \text{subject to} \quad F_x(p) = 1, \qquad Q_{x,k}(p) \geq \ell_{x,k} \quad (k = 1, \ldots, m_x), \tag{1}$$

where $\mathcal{P}(p_{0,x}, \gamma_x)$ is the set of projects reachable from the initial state while respecting the engine configuration, and m_x is the number of explicit auxiliary criteria. The primary score S_x, feasibility indicator F_x, criterion values $Q_{x,k}$, and thresholds $\ell_{x,k}$ are all supplied by the task evaluator and are benchmark-specific; when a benchmark defines no auxiliary criterion, this set is empty.

3.2 Executable Interaction and Verification

Building a game project is inherently multi-step as it proceeds through many stateful interactions with the engine. At step t, let p_t be the current project state, z_t the observable game runtime, and a_t an operation to the project or the runtime. The resulting development trajectory is

$$(p_0, z_0) \xrightarrow{a_0} (p_1, z_1) \xrightarrow{a_1} \cdots \xrightarrow{a_{T-1}} (p_T, z_T), \qquad a_t \in \{\text{inspect, edit, run, interact, observe}\}. \tag{2}$$

Each action can modify several files or resources, and cross-file call chains mean that a locally plausible edit may break an import, a scene, or an interaction. Correctness therefore cannot be judged by inspection alone; it must be checked against the tool outputs and runtime observations returned at each step. After generating the final executable project p_T, its runtime evidence ξ_T is passed to the task evaluator:

$$p_T \xrightarrow{\text{import + launch + interaction}} \xi_T \xrightarrow{\mathcal{E}_x} o_x = \big(S_x(p_T), \mathbf{Q}_x(p_T), F_x(p_T), \mathbf{D}_x\big). \tag{3}$$

The output combines the primary score, task-specific objectives or constraints, feasibility, and diagnostics, where $\mathbf{D}_x$ holds evidence references and $\mathbf{Q}_x$ records the task-specific quantities.

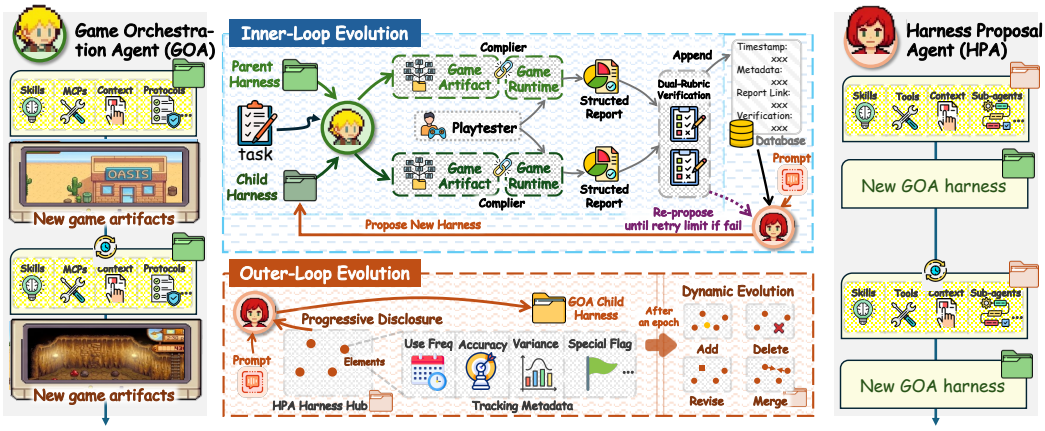


Figure 2: Game-Evolver is a dual-loop evolution framework. It uses a game orchestration agent (GOA) to create executable games and a harness proposal agent (HPA) to improve its harness. The inner loop conducts an A/B test of the parent (current) and child (new) harnesses by comparing the artifacts that each GOA run generates, with dual-rubric verification for comprehensive judgment.

4 Inner-Loop Evolution for the Game Orchestration Agent

Loop Components. This section introduces the inner loop of Game-Evolver, which holds the game-making harness fixed while recording evidence for later harness comparison. Figure 2 illustrates the framework that couples the inner loop to the outer one.

At the start of a run, the harness is a profile that the loop loads in full into the GOA's context. The profile is $H = \langle \mathcal{M}, \mathcal{U}, \pi_{\text{ctx}}, \pi_{\text{wf}}, \pi_{\text{val}}, \mathcal{Z} \rangle$, where $\mathcal{M}$ groups the instruction modules that steer the agent's planning and runtime behavior; $\mathcal{U}$ provides the agent to the engine and workspace interfaces; π_{ctx} compiles the task anchors and the evidence gathered during a run into the context; π_{wf} orders the workflow of the agent; π_{val} fires the replay and rubric gates and decides when a targeted repair branch is entered; and $\mathcal{Z}$ stores the reusable assets.

The loop therefore starts with a benchmark adapter, which stages the public task in a candidate workspace. The staged task then runs in the OpenGame runtime, which loads the selected backbone and the frozen harness into context. It then exposes bounded workspace / engine tools, and returns observations after each action. The agent plans from those observations and carries failure and probe summaries forward.

Loop Dynamics. The inner loop alternates between two activities. First, the GOA improves the game artifact while the harness stays frozen; this is the game-making run. Second, when the HPA proposes a child harness, the GOA reruns under the child harness and the two artifact sets are compared; the loop evolves only through this matched comparison.

Game-making run under a fixed harness. At generation step g, the GOA looks at the current artifact y_g, its latest evaluation o_g, and the history of steps so far $\tau_{\leq g}$, and decides what to change next: a change intent m_g, such as improving a mechanic, repairing a broken import, or trying a different strategy. This selection is the harness's change policy, realized by the planning modules of $\mathcal{M}$. The adapter stages the task in a workspace, and the context compiler π_{ctx} assembles the task, the current game, and the accumulated feedback into the agent's context. Together they turn that intent into an executable session:

$$(y_g, o_g, \tau_{\leq g}) \xrightarrow{\text{change policy}} m_g \xrightarrow{\text{adapter + context compiler}} \zeta_g = (x, y_g, m_g, \text{feedback}). \tag{4}$$

The GOA runs this session under the fixed backbone and the frozen harness, producing an updated game y'_g and a new history τ'_g. Gates and probes check the result, the evaluator scores it as o'_g. The selection policy keeps either the updated game or the previous champion (the best version so far):

$$\zeta_g \xrightarrow{\text{frozen harness}} (y'_g, \tau'_g) \xrightarrow{\text{gate + probes + task evaluator}} o'_g \xrightarrow{\text{selection policy}} (y_{g+1}, o_{g+1}) \text{ or } (y_g, o_g). \tag{5}$$

The intent is benchmark-independent: the adapter translates it into task-specific context while the change policy itself stays general. The harness therefore shapes planning, editing, and reactions to runtime feedback at every generation.

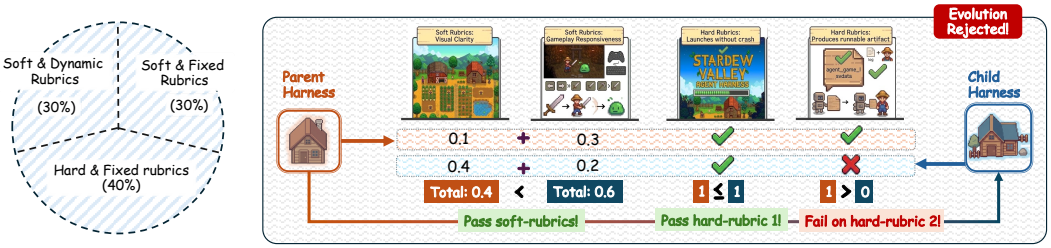


Figure 3: Dual-rubric verification for inner loop. **Left:** The rubric settings. Hard rubrics measure the runnability and legality of game artifacts; soft rubrics measure the completeness and other attributes of the game. Fixed rubrics are dataset-agnostic and fixed alongside the evolution; dynamic rubrics are generated from the parent harness and task descriptions. **Right:** A new harness is accepted only if it matches or exceeds the parent harness on every hard rubric and in the sum of all soft rubrics.

Harness A/B test. When the HPA proposes a child harness $\widetilde{H}^{\text{in}}_{t+1}$, the inner loop re-runs the game-making process under the child harness. The two runs yield paired artifacts, whose outcomes o_c^- and o_c^+ differ only in the harness. The loop then feeds the two artifact sets to the dual-rubric verification of Section 4, which decides whether the child harness is admitted or the parent remains the champion. Once the champion is decided, the GOA continues to update the game artifact under the winning harness, so artifact improvement and harness improvement alternate rather than overlap. Our design keeps the two sources of gain separable: a better game can come from a better harness or from a better artifact lineage, and only the matched comparison isolates the harness contribution.

Verification. Given the parent and child artifact sets from the matched replay of Section 4, we then enter the dual-rubric A/B test of Figure 3. Our rubrics differ along two axes: ❶ *hard/soft:* hard rubrics measure the legality of game execution and output binary 0/1 values, whereas soft rubrics score content completeness, subjective experience, and other non-legality attributes as continuous values in $[0, 1]$ from an LLM judge. ❷ *fixed/dynamic:* fixed rubrics are dataset-agnostic and fixed throughout evolution, whereas dynamic rubrics are generated according to the task description and the current champion harness. For each a/b test, rubrics are instantiated before scoring and shared.

Our design arises from a deliberate trade-off. If every rubric were regenerated from the newest harness and task at each step, a model could draft criteria that flatter the current child harness, and verification would drift toward the local optimum of whichever task happens to be active. If every rubric were frozen instead, the evaluation could neither reveal the fine-grained improvements that a particular child harness introduces nor escape an excessive emphasis on the held-out set. Therefore, keeping hard rubrics fixed guards the legality gate against redefinition, and confining the dynamic share to soft rubrics captures newly observed behavior without letting legality itself shift.

Formally, let o_c^- and o_c^+ denote the parent and child outcomes for case c, and let $\mathcal{C}_{\text{valid}}$ contain the aligned, feasible, score-bearing pairs with matching budgets and no infrastructure failure. Let $\mathcal{R}_h$ and $\mathcal{R}_s$ index the hard and soft rubrics, with w_j the nonnegative weight of soft rubric j. The comparison selects held-out game tasks and evaluates the two rubric families separately:

$$\forall c \in \mathcal{C}_{\text{valid}}: \qquad \mathsf{HardGate}(c) = 1 \iff \bigwedge_{j \in \mathcal{R}_h} r^+_{c,j} \geq r^-_{c,j}, \qquad r^{\pm}_{c,j} \in \{0, 1\}. \tag{6a}$$

$$\forall c \in \mathcal{C}_{\text{valid}}: \qquad \mathsf{SoftGate}(c) = 1 \iff \sum_{j \in \mathcal{R}_s} w_j r^+_{c,j} \geq \sum_{j \in \mathcal{R}_s} w_j r^-_{c,j}, \qquad r^{\pm}_{c,j} \in [0, 1]. \tag{6b}$$

A child harness is admitted only if $|\mathcal{C}_{\text{valid}}| \geq K$, where K is the minimum number of valid cases, protocol checks pass, and both gates hold for every valid case, with soft-rubric weights fixed within the parent-child pair. Dynamic rubric generation may tailor the criteria to a task, but it neither changes this admission rule.

The ledger entry completes verification by turning every attempt, including rejected ones, into future evidence. It records the artifact hash, harness identity, intent and score change, and final status of each attempt, so later harness epochs can reason from auditable traces.

5 OUTER-LOOP EVOLUTION FOR THE HARNESS PROPOSAL AGENT

Loop Components. Section 4 keeps a harness fixed while it produces a game artifact; In this section, we evolve the GOA harness itself with the outer loop, which searches the reusable element library that supplies better harnesses for inner changes. Formally,

$$H_t^{\text{in}} = \langle \mathcal{M}_t, \mathcal{U}_t, \pi_{\text{ctx},t}, \pi_{\text{wf},t}, \pi_{\text{val},t}, \mathcal{Z}_t \rangle, \qquad H_t^{\text{out}} = \langle E_t, S_t, P_t \rangle,$$

where the inner tuple H_t^{in} is the harness profile of Section 4 with each component indexed by the epoch t, and the outer tuple $H_t^{\text{out}} = \langle E_t, S_t, P_t \rangle$ groups the reusable element library E_t, the seeded active set S_t (which elements of E_t are preloaded into the proposer), and the change policy P_t (which library elements to edit). Table 5 lists the concrete harness components.

Concretely, the outer loop makes these edits through a fixed pipeline: completed run records are first condensed into a report A_t of attributed evidence, which collects outcome counts, repeated quality-failure families, and infrastructure events. The proposer reads that report together with the rejection memory $\mathcal{B}_t$, the record of past proposals , then output the next child harness. Once completed, the run's harness profile is loaded into the GOA's context, and its identity records exactly which harness components produced the trajectory.

Evolution & Verification. Unlike the inner loop, the outer loop loads its components through progressive disclosure, and manages them with usage metadata.[1] Concretely, elements are judged through usage, admission, and failure signals accumulated during evolution: For an element e, the engine maintains a usage counter $u_t(e)$ and an admission counter $s_t(e)$, and derives four signals to judge e:

$$\rho_t(e) = \frac{u_t(e)}{\max(1, \sum_{e'} u_t(e'))}, \quad \beta_t(e) = \frac{s_t(e)}{\max(1, u_t(e))}, \quad \beta_{t,c}(e) = \frac{s_{t,c}(e)}{\max(1, u_{t,c}(e))},$$

$$\mathcal{C}_t(e) = \{c \in \mathcal{C}_{\text{ho}} : u_{t,c}(e) > 0\}, \quad v_t(e) = \text{Var}_{c \in \mathcal{C}_t(e)}\big(\beta_{t,c}(e)\big), \quad \phi_t(e) = \mathbf{1}[h_t(e) > 0],$$

where $u_{t,c}(e)$ and $s_{t,c}(e)$ are the corresponding per-case usage and admission counts over the held-out cases $\mathcal{C}_{\text{ho}}$ used for evaluation. Thus $\rho_t(e)$ is the usage rate of e among all loaded elements, $\beta_t(e)$ is its acceptance rate, and $v_t(e)$ is the variance of its observed per-case acceptance rates over $\mathcal{C}_t(e)$; we set $v_t(e) = 0$ when fewer than two cases have observations. Finally, $h_t(e)$ counts the runs that loaded e and then failed a hard rubric, so the hard flag $\phi_t(e)$ records whether e was ever implicated in a large legality failure. As one epoch ends, The library is operated with five actions from $\mathcal{A}$, where each element e of the library has the same bounded action space $\mathcal{A}$:

$$\mathcal{A} = \{\text{add}, \text{delete}, \text{revise}, \text{merge}, \text{keep}\}, \tag{7}$$

This dynamic library management, unlike A/B testing, employs a form of post-hoc bounded evolution that ensures the stability of the higher-level concept of "proposal capability" across iterations.

6 EXPERIMENTS

6.1 EXPERIMENT SETTINGS

Benchmarks. We evaluate on four game-oriented benchmarks (GameCraft-Bench (Luo et al., 2026), GameDev-Bench (Chi et al., 2026), V-GameGym (Zhang et al., 2026b), VeriGame (Jia et al., 2026)) and two general agent benchmarks (Terminal-Bench, τ^2-Bench), covering both engine-based game generation and broader agent tasks. Details are in Appendix B.

Implementation Details. We employ OpenGame (Jiang et al., 2026b) as the agent harness backbone, the first open-source agentic framework for end-to-end game creation. We use the following model backbones: GLM5.2 (Zeng et al., 2026), Qwen3.6-27B (Team, 2026), Kimi-K2.7-Code (Team et al., 2026) (low thinking), DeepSeek-V4-Flash (Xu et al., 2026) (preview version),

[1] We do not adopt the A/B test for two reasons: (1) inner-loop A/B only needs to evaluate the game artifact, whereas outer-loop A/B would require measuring proposal capability, which is even harder to quantify; (2) the outer loop loads elements via progressive disclosure, their usage metadata is tracked naturally.

Table 1: Results on GameCraft-Bench (Luo et al., 2026). Mechanics, Depth, Visuals, and Art correspond to the four official rubric categories. The best scores are in bold.

Model	Method	Overall	Mechanics	Depth	Visuals	Art
Kimi-K2.7-Code	Baseline	20.06	28.25	18.30	25.23	16.09
	Awesome-Skills	18.37	23.35	**19.16**	**28.90**	10.93
	Game-Evolver	**24.45**	**39.61**	13.46	26.21	**28.20**
Qwen3.6-27B	Baseline	4.23	3.43	1.50	4.46	7.21
	Awesome-Skills	8.42	2.24	8.92	4.90	12.09
	Game-Evolver	**11.96**	**8.38**	**14.02**	**9.95**	**12.30**
GLM5.2	Baseline	18.00	26.50	20.17	21.15	10.84
	Awesome-Skills	20.67	25.24	27.30	20.01	12.35
	Game-Evolver	**35.33**	**43.69**	**36.71**	**37.98**	**29.22**
DeepSeek-V4-Flash	Baseline	6.31	6.10	7.13	9.18	4.35
	Awesome-Skills	9.69	6.03	13.95	5.89	8.62
	Game-Evolver	**12.67**	**13.42**	**14.27**	**13.10**	**10.55**
GPT-5.5	Baseline	39.47	54.36	38.61	41.84	**32.94**
	Awesome-Skills	38.12	55.44	42.32	42.30	24.70
	Game-Evolver	**42.17**	**58.49**	**48.87**	**54.00**	23.40
Claude-Sonnet-4.6	Baseline	22.64	37.02	24.38	20.38	15.71
	Awesome-Skills	17.68	31.95	19.99	24.91	6.15
	Game-Evolver	**26.96**	**46.30**	**27.50**	**30.53**	**16.59**

Table 2: Results on GameDev-Bench (Chi et al., 2026). 2DGA: 2D Graphics & Animation, 3DGA: 3D Graphics & Animation. The best scores are in bold.

Model	Method	Overall	2DGA	3DGA	User Interface	Gameplay Logic
Kimi-K2.7-Code	Baseline	3.00	2.76	3.03	4.11	2.44
	Awesome-Skills	3.60	2.76	9.09	1.37	**4.88**
	Game-Evolver	**6.90**	**6.20**	**15.15**	**6.85**	**4.88**
Qwen3.6-27B	Baseline	30.93	28.28	18.18	36.99	35.37
	Awesome-Skills	34.23	39.31	18.18	35.62	30.49
	Game-Evolver	**43.24**	**51.72**	**24.24**	**38.36**	**40.24**
GLM5.2	Baseline	53.75	53.79	42.42	53.42	**58.54**
	Awesome-Skills	60.66	77.93	30.30	57.53	45.12
	Game-Evolver	**73.57**	**85.52**	**75.76**	**75.34**	50.00
DeepSeek-V4-Flash	Baseline	12.31	7.59	12.12	10.96	21.95
	Awesome-Skills	48.05	41.38	**54.55**	**56.16**	50.00
	Game-Evolver	**49.25**	**46.21**	48.48	52.05	**52.44**
GPT-5.5	Baseline	55.56	77.24	21.21	41.10	**43.90**
	Awesome-Skills	50.75	62.76	39.39	**46.58**	37.80
	Game-Evolver	**57.96**	**80.00**	**45.45**	42.47	37.80
Claude-Sonnet-4.6	Baseline	50.75	**54.48**	18.18	63.01	**46.34**
	Awesome-Skills	41.44	41.38	21.21	54.79	37.80
	Game-Evolver	**52.85**	51.72	**36.36**	**72.60**	43.90

GPT-5.5, and Claude-Sonnet-4.6. The vanilla baseline and awesome-gamedev-agent-skills [2] (a fixed skill library) serve as comparisons in all experiments. For each dataset, all non-held-out tasks are available for evolution; We use our self-constructed rubrics for validation, instead of the dataset's public evaluation benchmarks, to avoid data leakage. The maximum number of evolution epochs is set to 35, and in each epoch we randomly sample 10% of the tasks from the original dataset (held-out sets excluded). Inner-loop rubric scoring is measured on a fixed 20% held-out set.

6.2 Experiment Results

As shown in Tables 1–3, Game-Evolver improves the overall score of every backbone across all four game benchmarks, and the gains are largest where the baseline is weakest. On GameCraft-Bench, Qwen3.6-27B and DeepSeek-V4-Flash roughly double their scores. We also find that:

① The improvement is concentrated on metrics that measure runnable, visible behavior rather than intended semantics. On GameCraft-Bench, Mechanics and Visuals rise on all six backbones; On GameDev-Bench, the weakest metric, 3D Graphics & Animation, shows some of the largest gains (Kimi-K2.7-Code from 3.03 to 15.15). On VeriGame, Build Health rises on all six backbones and Visual Usability gains 26% to 112%, while the Code metric on V-GameGym even declines.

②A fixed skill library is not a substitute for this evolution. Awesome-Skills lifts the three weakest backbones on GameCraft-Bench and nearly quadruples DeepSeek-V4-Flash on GameDev-Bench (12.31 to 48.05), but it lowers the strongest backbones, and on V-GameGym and VeriGame it beats Game-Evolver on no backbone. A fixed library helps only where it happens to match the runtime; the evolved harness helps everywhere.

[2] https://github.com/gamedev-skills/awesome-gamedev-agent-skills

Table 3: Results on V-GameGym (Zhang et al., 2026b) and VeriGame (Jia et al., 2026). The detailed definitions of rubrics can be found in Section B. The best scores are in bold.

Model	Method	V-GameGym				VeriGame			
		Overall	Code	Screenshot	Video	Overall	BH	VU	IA
Kimi-K2.7-Code	Baseline	36.59	78.96	17.01	13.79	40.90	63.20	23.50	40.20
	Awesome-Skills	31.31	63.91	15.36	14.67	43.12	65.46	22.76	41.14
	Game-Evolver	**39.69**	**82.42**	**20.46**	**16.18**	**55.96**	**73.09**	**49.87**	**48.96**
Qwen3.6-27B	Baseline	30.03	73.00	11.30	5.79	35.38	52.10	27.86	36.57
	Awesome-Skills	29.93	73.50	9.49	6.80	35.00	52.76	30.92	35.08
	Game-Evolver	**34.20**	**74.63**	**15.94**	**12.04**	**47.65**	**65.13**	**43.91**	**38.17**
GLM5.2	Baseline	37.60	72.78	17.89	22.13	46.19	60.56	37.19	42.06
	Awesome-Skills	42.46	82.42	23.16	21.80	41.22	70.87	35.70	30.83
	Game-Evolver	**48.45**	**90.79**	**29.43**	**25.11**	**56.39**	**73.75**	**50.63**	**48.18**
DeepSeek-V4-Flash	Baseline	37.90	**76.46**	18.37	18.86	21.87	42.50	11.91	13.65
	Awesome-Skills	33.06	72.16	12.60	14.43	23.21	43.96	8.49	7.17
	Game-Evolver	**41.60**	76.38	**27.12**	**21.30**	**30.35**	**45.74**	**24.26**	**20.13**
GPT-5.5	Baseline	43.87	**85.86**	23.53	22.21	50.50	56.25	44.50	50.75
	Awesome-Skills	40.49	80.89	16.10	24.47	48.54	56.90	40.44	48.94
	Game-Evolver	**45.67**	84.05	**24.52**	**28.43**	**65.45**	**72.06**	**56.08**	**59.56**
Claude-Sonnet-4.6	Baseline	28.59	**63.12**	11.81	10.84	22.93	26.67	19.47	**22.67**
	Awesome-Skills	28.47	60.59	14.03	10.78	18.47	20.00	17.07	18.33
	Game-Evolver	**30.66**	62.12	**15.05**	**14.81**	**38.50**	**47.40**	**29.54**	22.16

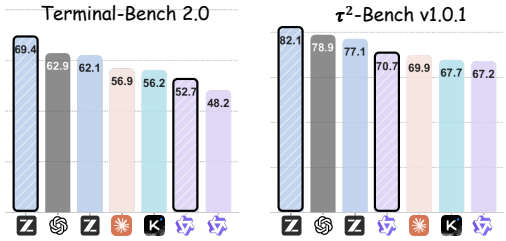


Figure 4: Results on Terminal-Bench and τ^2-Bench. Bars with black contours are those with Game-Evolver.

Table 4: Per-class results on τ^2-Bench. $*$: results contain recurring infrastructure issues.

Model	Avg	Airline	Retail	Telecom	Banking
Kimi-K2.7-Code	67.7	78.0	81.6	93.0	16.5
Qwen3.6-27B	67.2	86.0	76.3*	92.1	17.5
GLM5.2	77.1	92.0	83.3	99.1	36.1
Claude-Sonnet-4.6	69.9	68.0	89.5	80.7	35.1
GPT-5.5	78.9	94.0	88.6	98.2	37.1
Qwen3.6-27B	67.2	**86.0**	76.3*	**92.1**	17.5
+Game-Evolver	**70.7**	84.0	**80.7**	**92.1**	**26.8**
GLM5.2	77.1	**92.0**	83.3	99.1	36.1
+Game-Evolver	**82.1**	88.0	**91.2**	**100.0**	**47.4**

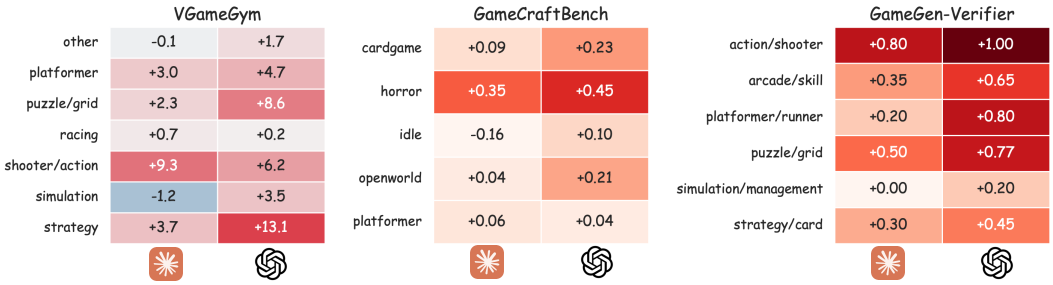


Figure 5: By-type improvement of Game-Evolver compared with the baseline on V-GameGym, GameCraft-Bench, and VeriGame (labeled GameGen-Verifier in the source plot). The left and right columns in each panel correspond to Claude-Sonnet-4.6 and GPT-5.5, respectively.

③ The gains depends on both the runtime and the model capability. While GLM5.2 is not the strongest baseline, yet it gains 22% to 96% and overtakes GPT-5.5 on GameDev-Bench and V-GameGym. Claude-Sonnet-4.6 records the largest VeriGame gain (67.9%) despite starting near the bottom there. These results show that the improvement is a property of the (backbone, runtime) pair, not of the backbone's starting score.

Generalization to broad agent tasks. We also transfers Game-Evolver to general agent tasks, and the results are shown in Figure 4 / Table 4. On Terminal-Bench and τ^2-Bench, GLM5.2 and Qwen3.6-27B gain 3.5 to 7.3 points. The largest τ^2-Bench gains occur in Banking, the weakest class, and the evolved GLM5.2's 82.1 surpasses the GPT-5.5's 78.9. These results echoes the results seen on the game benchmarks, showing that the essence of Game-Evolver is wide applicable.

7 Analysis

By-type Analysis. Figure 5 decomposes the overall gain by game type, and shows that improvement is concentrated rather than uniform. On V-GameGym, strategy (+13.1 on GPT-5.5) and shooter/action (+9.3 on Claude) account for most of the gain while racing moves by less than

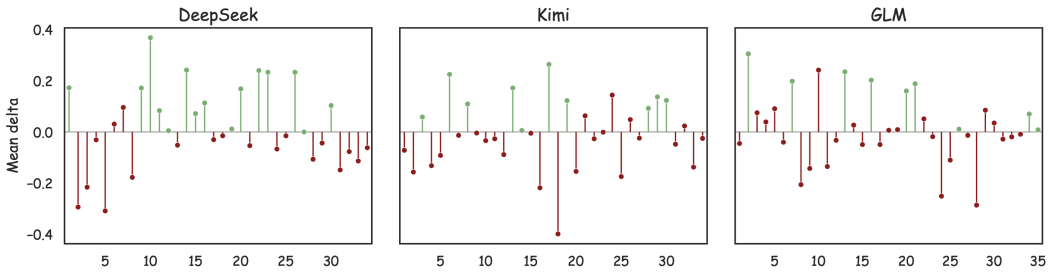


Figure 6: Per-epoch change of the inner loop's self-built dual-rubric scores for harnesses proposed by the HPA, for the DeepSeek, Kimi, and GLM backbones. Green marks accepted child harnesses and red marks rejected child harnesses.

one point; on GameCraft-Bench, horror leads while platformer is nearly flat; on VeriGame, action/shooter leads while simulation/management barely changes. Evolution therefore fixes a few weak game types instead of lifting all types equally. This concentration also reconciles an apparent tension with the aggregate tables: although weaker backbones show the largest relative overall gains, GPT-5.5 still makes larger absolute per-type gains than Claude-Sonnet-4.6 in most cells. The two views use different denominators (absolute per-type change versus relative overall change), so they complement rather than contradict each other.

Scoring Dynamics. Figure 6 traces how the inner loop scores each harness modification proposed by the HPA. Accepted modifications almost always raise the mean dual-rubric score, and rejected ones are mostly negative. The exceptions are informative: some rejected modifications improve the mean score but fail another condition, such as a hard rubric, a protocol check, or a per-case non-regression constraint. The admission gate is therefore a multi-condition filter, not a scalar hill-climber. The three backbones also differ: DeepSeek accepts more modifications and shows several gains above +0.2, while GLM's late epochs settle near zero with only two accepted changes after epoch 30. The same rule thus produces different evolution behavior on different backbones.

Ablation and Cost Analysis. Figure 7 ablates evolvable harness on GameCraft-Bench, plotting overall score against time cost (token consumption shown as point size). The three context-only variants L0-L2 cluster in the low-score region and are not even monotonic; L3 additionally evolves tools and workflows, and it jumps to about 0.22 for Kimi and 0.30 for GLM5.2. "Ours" adds the outer loop on top of L3 and reaches the highest score for both backbones, at a time cost that grows by roughly 1.5 to 1.7 times. Since "Ours" differs from L3 only by the outer loop, the extra gain and cost can be attributed to that component. In other words, the outer loop is what turns a harness evolution from a static procedure to a evolvable process.

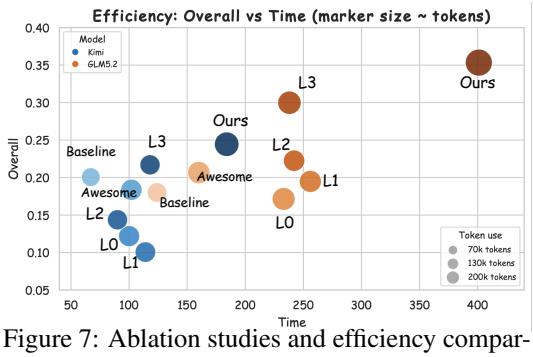


Figure 7: Ablation studies and efficiency comparisons on GameCraft-Bench. The evolvable components form a strict containment hierarchy. L0: prompts; L1: L0+policies; L2: L1+skills; L3: full Game-Evolver inner-loop.

8 Conclusion

We presented Game-Evolver, a dual-loop framework that couples game-artifact execution with harness evolution. In the inner loop, a game orchestration agent (GOA) turns a specification into a runnable game under the current champion harness and collects the evidence needed for judgment, while a matched-replay A/B test admits a new harness only when dual-rubric verification passes. A harness proposal agent (HPA) drives the outer loop, distilling recurring failure families into a bounded harness element change. Our design enables both the game-making harness and the harness-proposal loop to keep improving after deployment, boosting long-horizon evolution while remaining agnostic of game or infrastructure types.

AI USE STATEMENT

In this work, we used generative AI tools to edit the manuscript for readability, grammar, and presentation. We did not use generative AI tools for the required research tasks listed in the ICLR 2027 AI Policy for Authors; all other such tasks are not applicable to this work. The authors reviewed all AI-assisted text and take responsibility for the final content of this paper, including its text, claims, and artifacts.

ETHICS STATEMENT

This work develops an evaluation-driven framework for improving game-development agents. The study involves no human subjects, human-participant experiments, or collection of personal data. All experiments use public benchmarks, automated evaluation procedures, and an open-source game project. Our evaluation protocol combines matched replay with fixed legality checks and quality rubrics, supporting reliable and auditable assessment of generated game artifacts. We follow responsible research practices and respect the licenses and terms governing the benchmarks, models, engines, and assets used in this work.

REPRODUCIBILITY STATEMENT

To support reproducibility, the paper and appendix describe the dual-loop algorithm, admissible harness components, benchmark protocols, model backbones, baselines, evolution budget, task sampling rule, held-out evaluation split, matched-replay admission rule, and validation rubrics. We will release a documented repository containing the implementation, configurations, harness libraries, prompts, benchmark preprocessing and splits, evaluation scripts, and experiment records, subject to the licenses and access conditions of the underlying benchmarks, models, and game assets. The release will also document the model identifiers, runtime environment, and evaluation configuration used for the reported experiments.

REFERENCES

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

Bowen Baker, Ilge Akkaya, Peter Zhokhov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining: Learning to act by watching unlabeled online videos. In *Advances in Neural Information Processing Systems*, 2022.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. tau^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

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.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *International Conference on Machine Learning*, 2024.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *Advances in Neural Information Processing Systems*, 2022.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin

Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

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.

Xue Jiang, Yihong Dong, Mengyang Liu, Deng Hongyi, Tian Wang, Yongding Tao, Zhi Jin, Wenpin Jiao, and Ge Li. Coderl+: Improving code generation via reinforcement with execution semantics alignment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3608–3622, 2026a.

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

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

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. Computerrl: Scaling end-to-end online reinforcement learning for computer use agents. In *International Conference on Learning Representations*, volume 2026, pp. 23553–23591, 2026.

Changxin Lao, Fei Pan, Guozhuang Ma, Han Li, Huihuang Lin, Jijun Shi, Kangzhi Zhao, Kun Gai, Mo Zhou, Qinqin Zhou, et al. Agentx: Towards agent-driven self-iteration of industrial recommender systems. *arXiv preprint arXiv:2606.26859*, 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.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for mind exploration of large language model society. In *Advances in Neural Information Processing Systems*, 2023.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026a.

Minhua Lin, Juncheng Wu, Zijun Wang, Zhan Shi, Yisi Sang, Bing He, Zewen Liu, Tianxin Wei, Zongyu Wu, Zhiwei Zhang, Dakuo Wang, Xiang Zhang, Benoit Dumoulin, Cihang Xie, Yuyin Zhou, Suhang Wang, and Hanqing Lu. Harness updating is not harness benefit: Disentangling evolution capabilities in self-evolving llm agents. *arXiv preprint arXiv:2605.30621*, 2026b.

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

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *ACM Symposium on User Interface Software and Technology*, 2023.

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 Annual Meeting of the Association for Computational Linguistics*, 2024.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, et al. Cradle: Empowering foundation agents towards general computer control. *arXiv preprint arXiv:2403.03186*, 2024.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Qwen Team. Qwen3. 5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575:350–354, 2019.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. In *Advances in Neural Information Processing Systems*, 2024b.

Xucong Wang, Zhe Zhao, Liheng Yu, Di Wu, Xiaofeng Cao, and Pengkun Wang. Didpo: Diff-in-diff policy optimization for coding agent training. *arXiv preprint arXiv:2608.07147*, 2026.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026.

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. *arXiv preprint arXiv:2603.07106*, 2026.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026a.

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

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.

Inner loop: game-orchestration harness		Outer loop: harness-proposal harness	
harness type	**Allowed scope and action**	**harness type**	**Allowed scope and action**
Instruction modules	Planning, gameplay, observability, repair, and strategy modules; one epoch may add, drop, or swap one module while the other dimensions remain frozen.	Reusable library	*skill*, *MCP*, *tool*, *context*, *protocol*, and *workflow* entries; one epoch may add, prune, merge, or replace reusable entries.
Tool interfaces	Bounded `mcp_server`, `command_wrapper`, `engine_probe`, and `env_binding` interfaces; one epoch may expose or retract one interface under a fixed safety scope.	Seeded active set	Per-category seed elements; one epoch may change which reusable elements are preloaded into the proposer before it writes an inner-harness gradient.
Context compiler	Task anchors, probe summaries, failure traces, and edit history; one epoch may change their ordering, truncation, or inclusion policy.	Active caps	Per-category element budgets; one epoch may raise or lower how many elements of each category can be active together.
Workflow	candidate game workspace workflow sequencing; may order and gate probe, diagnose, plan, patch, verify, checkpoint, rollback, and regression actions within the current task workspace;	Change policy	Usage-driven removal, rewrite, and merge thresholds; one epoch may bias which stale or low-accuracy elements are changed after enough evidence accumulates.
Validation policy	Replay gates, rubric gates, and targeted repair branches; one epoch may change when a repair branch is entered or when a child harness is rejected.	Library workflows	Add, prune, merge, and replace workflows; one epoch may turn a rejected diagnosis into a reusable update for future inner proposals.

Table 5: The two nested loops expose different harness types. The inner loop edits the game-related harness that creates runnable artifacts, while the outer loop edits the reusable element library that shapes future inner proposals.

A Optimizable harness in Game-Evolver

Excessively high degrees of freedom in harness evolution only reduce exploration efficiency, whereas evolution with a moderate degree of freedom can simultaneously ensure the generality and generalizability of the evolution process. By combining expert knowledge with exploratory attempts, Game-evolver ultimately delineates the types of harnesses that can be optimized, shown in Table 5.

B Detailed Introduction of Datasets

This appendix details the benchmarks used in the experiments of Section 6.1, one per subsection: the four game generation benchmarks GameCraft-Bench (Luo et al., 2026), GameDev-Bench (Chi et al., 2026), V-GameGym (Zhang et al., 2026b), and VeriGame (Jia et al., 2026), together with the scoring protocols we apply to them, and the two general agent benchmarks Terminal-Bench (Merrill et al., 2026) and τ^2-Bench (Barres et al., 2025; Yao et al., 2024) used for the transfer evaluation. Inside the loop we do not rely on any dataset's public evaluator, using our own rubrics for validation and feedback to avoid leakage; the three LLM-judge rubrics of Section B.3 are the self-constructed protocol.

B.1 GameCraft-Bench

GameCraft-Bench (Luo et al., 2026) asks the agent to produce a runnable Godot game from a task description: 140 tasks spanning 15 game families, such as platformers, strategy, and roguelikes.

Each task pairs a public specification with a staged starter project, and the agent must edit scenes, scripts, and resources until the intended game actually plays. Evaluation replays recorded input traces inside a Godot runtime and scores hidden rubric categories spanning mechanics, depth, visuals, and art. Because scoring replays the trace in the real engine, a candidate that imports but never responds to the recorded inputs earns no credit. GameCraft-Bench is the main example of the executable-gated principle in our experiments, the setting of the harness and rubric analyses in Section 7, and the source of the dynamic-rubric instantiations in Table 7.

B.2 GAMEDEV-BENCH

GameDev-Bench (Chi et al., 2026) raises the difficulty to 132 multi-file Godot development tasks derived from web and video tutorials, covering gameplay logic, 2D and 3D graphics, and user-interface work. Each task combines code with multimodal assets, such as shaders, sprites, and animations, and the average solution requires more than three times the lines of code and file changes of typical software-engineering benchmarks. Every candidate must remain importable and pass deterministic execution-based checks, so a solution receives no credit unless it loads in the Godot editor and satisfies the task's runtime assertions. The benchmark is organized into four sub-types, 2D Graphics & Animation (2DGA), 3D Graphics & Animation (3DGA), User Interface, and Gameplay Logic, which Table 2 reports separately, and its tutorial-derived tasks are where the silent code-level failures of the case studies in Section C are most prominent.

B.3 V-GAMEGYM

V-GameGym (Zhang et al., 2026b) contains 2,219 Pygame game samples across 100 thematic clusters, each refactored over a fixed recording horizon so that code, screenshots, and video are all scored at once. The agent receives a short game description and must produce a Pygame project whose recorded session shows playable behavior. Because the official channels, Code, Screenshot, and Video, are graded together over the same recording, a static code patch that never renders visible gameplay receives no credit. Our in-loop protocol adds a Build Health (BH) rubric, defined below, which reports whether the project runs at all.

Inside the loop we score V-GameGym tasks with three LLM-judge rubrics of our own, defined below, and we do not rely on the benchmark's public evaluator for validation or feedback, which avoids leakage. The same Visual Usability and Intent Alignment rubrics also score the VeriGame columns of Table 3, complementing that benchmark's verifier chain and its Build Health channel, while the reported V-GameGym numbers in that table keep the official Code/Screenshot/Video channels.

Build Health. Whether a generated project builds and serves. The artifact must be structurally complete, launch without critical errors, and stay runnable under headless execution; the judge inspects the project structure and runs the entry point, so a project that fails to build or serve receives a much lower score, keeping build failures low across a healthy corpus. Build Health is the legality gate in the A/B admission of Section 4 and the only one of the three that is hard.

Visual Usability. Whether the rendered game is visibly playable, judged from screenshots captured during runtime: scenes are coherent, animated, and interactable. When no screenshot is available, the judge infers visual usability from code and UI clues, such as scene composition, layout, and draw calls, instead of defaulting to a neutral score.

Intent Alignment. The degree to which the implemented mechanism matches the intent expressed in the task instruction. The judge decomposes the instruction into requirements and checks executable behavior against each one, so a project that only resembles the intended game without exposing the requested mechanic scores low. This is the criterion behind the interface-versus-intent failures in the case studies (Cases #7 and #8).

B.4 VERIGAME

VeriGame (Jia et al., 2026) evaluates web games on the JavaScript/TypeScript/HTML stack, covering 100 games across seven genres. Instead of replaying recorded input, the benchmark injects

runtime state directly into the running game and runs a verifier chain over the exposed gameplay variables, asserting that the intended mechanics act on them. The chain therefore measures behavior through the game's own state, which makes it well suited to catching the interface-versus-intent failures reported in Cases #7 and #8 of Section C. Build Health is the benchmark's hard channel, reporting whether the web build loads and serves. For the VeriGame columns of Table 3, we keep the verifier chain and Build Health and add the Visual Usability and Intent Alignment rubrics of Section B.3, which let the loop grade visible playability and intent matching for mechanics the verifier chain does not assert.

B.5 TERMINAL-BENCH

Terminal-Bench (Merrill et al., 2026) evaluates agents on realistic command-line workflows: 89 curated tasks spanning domains such as software engineering, system administration, and machine learning. Each task runs in a containerized official runner that provisions a fresh shell, and the agent must operate through the constrained environment until the task's automated checks accept the outcome. Success therefore exercises tool use, command sequencing, and recovery from errors under a modality different from game development. We use Terminal-Bench 2.0 and report the official runner's task-success metric. Because its environment and checks differ entirely from the game benchmarks, applying Game-Evolver here tests whether the dual-loop mechanism transfers beyond game generation.

B.6 τ^2-BENCH

τ^2-Bench (Barres et al., 2025; Yao et al., 2024) evaluates conversational agents in a dual-control customer-service environment, extending the tool-agent-user setup of τ-Bench (Yao et al., 2024) to the Telecom and Banking domains. The agent converses with a simulated user in a shared environment and completes service requests through tool use, so the official simulator scores long-horizon communication and policy compliance rather than artifact construction. We use the official v1.0.1 simulator and evaluation protocol, reporting the Airline, Retail, Telecom, and Banking classes separately as in Table 4. As with Terminal-Bench, the benchmark shares no harness or artifact machinery with game generation, so it tests whether the framework transfers to a dialogue-driven agent task.

C MORE CASE STUDIES

This appendix presents the remaining case studies from the four benchmarks, each contrasting the baseline output with the Game-Evolver output.

C.1 EVOLVED HARNESS FOR GOA AGENTS.

Table 6 lists eight admitted harness changes and their observed mean-rubric-score differences. All changes are positive, from +0.058 to +0.258. The two largest increases are associated with replacing a phase-handoff requirement by a successful deep runtime probe (+0.258) and general import validation by executable-entry-point discovery (+0.225); engine-level import validation is close behind at +0.220. The table also contains dependency validation, replay auditing, checkpoint replacement, and context-history changes, so the admitted set is broader than runtime probing alone. Two rows reverse the same pair of rules: the phase-handoff direction records +0.058, whereas the deep-runtime-probe direction records +0.258. The largest observed gains are attached to changes that expose executable runtime evidence.

C.2 RUBRIC CASES

Dynamic rubrics are not fixed checklists: they are regenerated for each epoch from the current harness and the active task description, so their dimensions track what the harness can actually observe. Table 7 shows three examples from the GameCraft-Bench evolution.

The first two rows keep the dataset fixed (idle-dungeon-guild) and vary the harness. Under the process-oriented harness, which centers on edit-verify handoff and probe-first repair, the rubric is process-centric (workflow coherence, repair quality, exploration efficiency, skill selection, tool/MCP

Old harness	New harness	Change Description	Rubric Score
Validated Godot resource imports before making gameplay edits.	Discovered and validated the runnable project entry point, including the project manifest, main scene, and entry scripts, before editing.	Shifted preprocessing from general import validation to identifying the executable path used by subsequent playtests and verification.	0.242 → 0.467 (+0.225)
Validated declared dependencies, add-ons, and referenced resources before gameplay edits.	Ran Godot import validation before editing and recorded scene, script, and resource parsing failures.	Replaced dependency-level validation with engine-level import and parsing checks.	0.198 → 0.418 (+0.220)
Validated Godot resource imports and scene parsing before gameplay edits.	Validated declared dependencies, add-ons, imported resources, and referenced files before gameplay edits.	Broadened pre-edit validation from engine import errors to missing or inconsistent project dependencies.	0.193 → 0.328 (+0.135)
Blocked candidate acceptance until at least one deep runtime probe succeeded.	Required explicit planning, editing, and verification phases before candidate acceptance.	Replaced a single runtime gate with a structured edit-to-verification handoff requiring a separate verification phase.	0.307 → 0.365 (+0.058)
Required an explicit transition from editing to verification before accepting a candidate.	Blocked candidate acceptance until at least one deep runtime probe completed successfully.	Replaced a procedural handoff requirement with an evidence-based acceptance gate tied to successful runtime execution.	0.353 → 0.612 (+0.258)
Probed whether the project had a valid executable entry point.	Audited deterministic demonstration replays for observable gameplay evidence.	Shifted validation from confirming that the game could launch to confirming that demonstrations exercised meaningful gameplay states and interactions.	0.153 → 0.303 (+0.150)
Created a checkpoint before editing and rolled back changes after a hard regression.	Ran deep runtime probes before diagnosing problems and proposing gameplay patches.	Changed the workflow from reactive rollback to evidence-first diagnosis, so patches were informed by runtime behavior before implementation.	0.350 → 0.438 (+0.088)
Added engine-specific hints inferred from the workspace structure to the agent context.	Added summaries of the four most recent patches and their outcomes to the agent context.	Replaced static engine guidance with short-term edit history intended to prevent repeated unsuccessful modifications.	0.152 → 0.327 (+0.175)

Table 6: Accepted harness evolution cases and their score change.

effectiveness, context compilation), with playtest signal quality and feature progress visibility as the only player-facing dimensions. Under the evidence-gathering harness, which adds dependency checks and file-tree inspection so it observes the candidate project itself, the rubric shifts to the artifact: public feature completion, core gameplay depth, interaction correctness, progression and end state, playability and balance, runtime feedback quality, and demo coverage.

The last two rows hold the harness fixed and vary the dataset. The seven outcome dimensions survive intact for visualnovel-grimfable, with anchors localized to the task, such as the UI feedback emphasized by the visual novel against the level progression emphasized by the dungeon game. Dynamic rubrics thus inherit the evidence structure of the harness and specialize it to the active task, while the hard rubrics of the admission gate (Section 4) stay fixed, so legality never moves with the re-drafted soft criteria.

harness	Dynamic Rubric
gcbench / idle-dungeon-guild *harness summary: process-oriented inner harness. It anchors the public task spec, summarizes probe outputs and recent failures, requires edit-verify handoff, enforces workspace boundaries, runs Godot headless playtests, runtime smoke tests, demo replay audits, artifact inventory, entrypoint discovery, checkpoint rollback, and probe-first repair workflow.*	workflow_coherence: Multi-step workflow (plan → edit → verify) is coherent and avoids thrashing. playtest_signal_quality: Runtime/playtest observations meaningfully informed edits (task family: Godot game crafting with deterministic input replay evidence). feature_progress_visible: Player-visible mechanics progressed toward the public task goal. repair_quality: Failures were repaired without regressing previously working behavior. exploration_efficiency: Agent explored alternatives efficiently under harness budget discipline. skill_selection_usefulness: Skill loading/curator modules improved decision quality on this task. tool_mcp_effectiveness: Tool/MCP interfaces produced actionable evidence before patching. context_compilation_clarity: Context compilation kept relevant history and trimmed noise.
gcbench / idle-dungeon-guild *harness summary: stronger evidence-gathering harness. It keeps the same task anchoring, probe summaries, failure history, workspace boundary, Godot headless playtest, runtime smoke, regression suite, checkpoint rollback, and probe-first workflow, but adds tool_dependency_check and mcp_file_tree_inspect to inspect candidate project structure and dependency state before/while repairing gameplay.*	public_feature_completion: Observable gameplay implements the important requirements in the public task instruction, including level progression and UI feedback; nominal files, labels, or planned features without executable behavior receive no credit. core_gameplay_depth: The core game loop has meaningful state transitions, consequences, and enough mechanical depth to sustain play in this Godot game-crafting task with deterministic input replay evidence. interaction_correctness: Real player inputs trigger correct, deterministic gameplay responses, and the runtime evidence shows those responses rather than only a launch or title screen. progression_and_end_state: The game supports coherent progression and reaches task-appropriate success, failure, result, or completion states through actual play. playability_and_balance: The game is practically playable: controls, pacing, challenge, resources, and failure recovery do not block or trivialize the intended experience. runtime_feedback_quality: Runtime presentation clearly communicates actions, state changes, hazards, progress, and outcomes through readable visual or textual feedback. demo_coverage: Completed deterministic replay evidence covers the core loop and important states; demo JSON or nominal scenarios without successful replay receive no credit.
gcbench / visualnovel-grimfable *harness summary: same evidence-gathering harness as above, but applied to a different public task. It anchors the visual-novel task spec, uses file-tree inspection, dependency checks, Godot replay evidence, demo replay audits, and rollback-based verification to guide repair.*	public_feature_completion: Observable gameplay implements the important requirements in the public task instruction, including UI feedback; nominal files, labels, or planned features without executable behavior receive no credit. core_gameplay_depth: The core game loop has meaningful state transitions, consequences, and enough mechanical depth to sustain play in this Godot game-crafting task with deterministic input replay evidence. interaction_correctness: Real player inputs trigger correct, deterministic gameplay responses, and the runtime evidence shows those responses rather than only a launch or title screen. progression_and_end_state: The game supports coherent progression and reaches task-appropriate success, failure, result, or completion states through actual play. playability_and_balance: The game is practically playable: controls, pacing, challenge, resources, and failure recovery do not block or trivialize the intended experience. runtime_feedback_quality: Runtime presentation clearly communicates actions, state changes, hazards, progress, and outcomes through readable visual or textual feedback. demo_coverage: Completed deterministic replay evidence covers the core loop and important states; demo JSON or nominal scenarios without successful replay receive no credit.

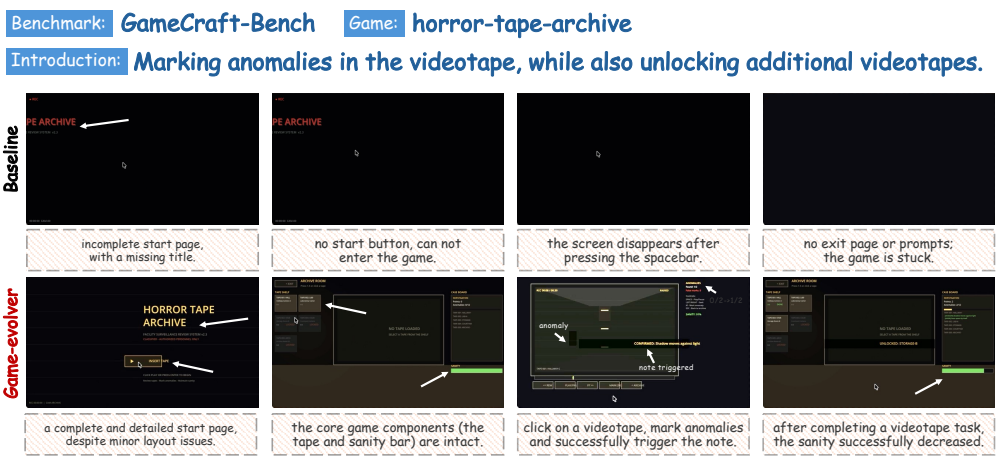


Figure 8: Case #1: horror-tape-archive from GameCraft-Bench. The images above are screenshots of games generated by the baseline, which scored 0.5763; the images below are those generated by Game-Evolver, which scored 0.7702.

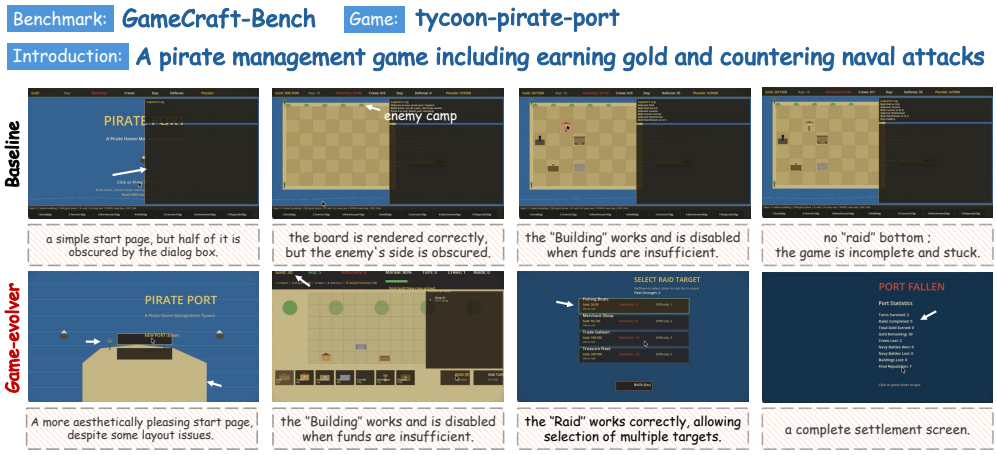


Figure 9: Case #2: tycoon-pirate-port from GameCraft-Bench. The images above are screenshots of games generated by the baseline, which scored 0.3250; the images below are game screenshots from Game-Evolver, which scored 0.4483.

C.3 Game Cases

Case #1: horror-tape-archive from GameCraft-Bench. Figure 8 shows a GameCraft-Bench case and the interaction gap behind the baseline and evolved results. In the baseline trace, the title page is incomplete, no start action is available, and later frames become blank or stuck after input. The evolved trace shows a complete start flow, the tape and sanity interfaces, and an anomaly-marking response followed by a sanity decrease. The screenshots therefore document a change from a broken interaction path to the requested visible state transitions, consistent with the framework's runtime-oriented verification.

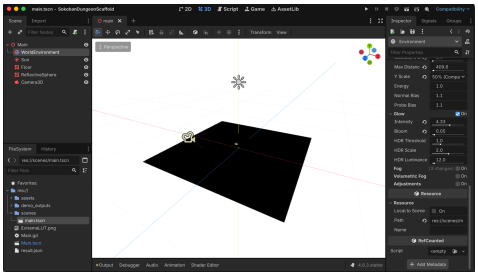
Case #2: tycoon-pirate-port from GameCraft-Bench. Figure 9 shows a second case from GameCraft-Bench. In the baseline trace, building is available but no raid action or settlement state is shown, and the final frame is labeled incomplete and stuck. The evolved trace retains building, adds a raid-target selection screen, and reaches a settlement screen with statistics. The before/after pair therefore shows the evolved artifact completing an interaction path absent from the displayed baseline run. Additional cases from all four benchmarks appear in Section C.

Case #3: Depth Fog / LUT from GameDev-Bench. The GameDev-Bench failure in Figure 10 is semantic rather than structural: the scene loads, but the shared `WorldEnvironment` leaves depth fog, volumetric fog, and LUT correction disabled, whereas the evolved workspace enables all three

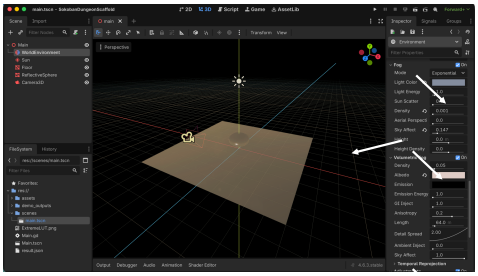
together with the required renderer setting. The before/after pair directly documents correction of semantic incompleteness that a file-existence check would not detect.

Benchmark: GameDev-Bench Game: task_0015 Depth Fog / LUT

Introduction: Enable depth fog, volumetric fog, and LUT color correction in the scene's shared WorldEnvironment.



Error: fog/volumetric fog/LUT not enabled.



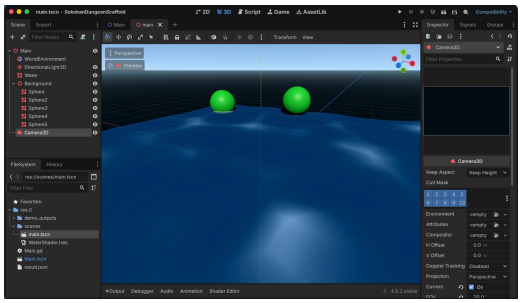
Correct: fog/volumetric fog/LUT enabled.

Figure 10: Case #3: task_0015 Depth Fog / LUT from GameDev-Bench. The left image is the Godot workspace of the baseline (score: 0.0); the right image shows Game-Evolver (score: 1.0).

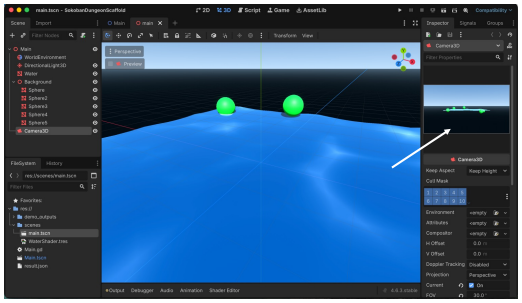
Case #4: Water Depth Scene from GameDev-Bench. The GameDev-Bench failure in Figure 11 is visual rather than structural: the water scene is present, but the baseline camera preview does not frame all five green spheres. The evolved workspace shows the corrected camera framing and receives a score of 1.0. The figure directly supports the camera-framing correction.

Benchmark: GameDev-Bench Game: task_0077 Water Depth Scene

Introduction: Build a water-depth visualization scene with a shader water plane, five green spheres, lighting, sky, and a camera that frames all spheres.



Error: Camera should be angled such that all spheres are visible.



Correct: Depth scene geometry matches tutorial setup.

Figure 11: Case #4: task_0077 Water Depth Scene from GameDev-Bench. The agent is asked to build a water-depth visualization scene with a shader water plane, five green spheres, lighting, sky, and a camera that frames all spheres. The left image is the Godot workspace of the baseline (score: 0.0); the right image shows Game-Evolver (score: 1.0).

Case #5: SpaceStoneDodger from V-GameGym. The baseline trace in Figure 12 starts directly in gameplay and shows a controllable spaceship, but its later frames contain a key-state `TypeError`. The evolved trace adds start and options pages, retains controllable gameplay, and reaches a game-over screen. The screenshots thus distinguish early renderability from completion of a playable recorded session.

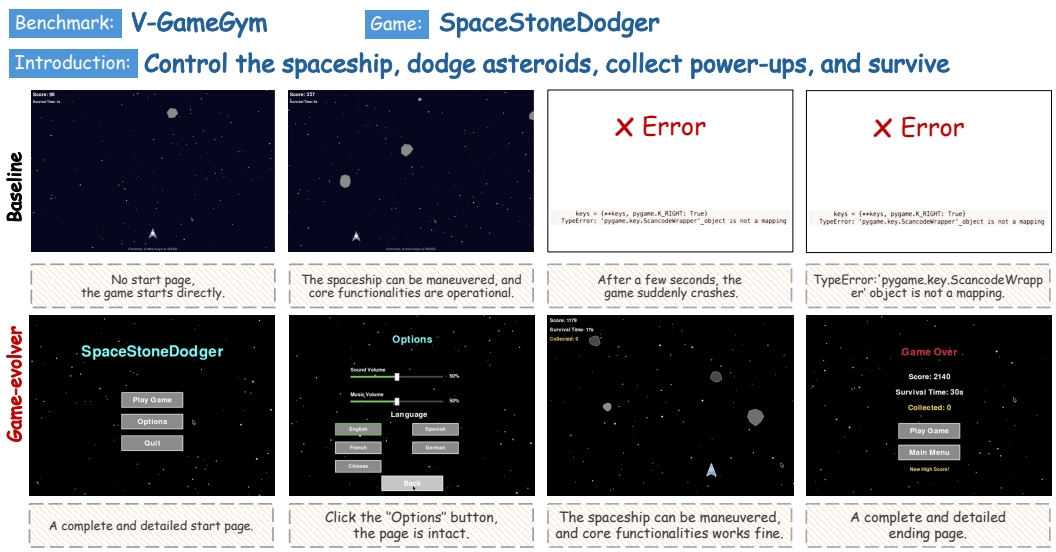


Figure 12: Case #5: SpaceStoneDodger from V-GameGym. The player controls the spaceship to dodge asteroids and collect power-ups. The images above are screenshots of games generated by the baseline, which scored 0.1980; the images below are screenshots of games generated by Game-Evolver, which scored 0.3520.

Case #6: Dark Tower Maze from V-GameGym. Figure 13 shows that the baseline renders a maze and moves the explorer automatically, but the annotated trace states that the player cannot control it and that the route demonstration ends after waiting. In the evolved trace, a recorded button press moves the explorer, and later frames reach Levels 2 and 3. The figure therefore supports the change from an automated demonstration to player-controlled progression.



Figure 13: Case #6: Dark Tower Maze from V-GameGym. This is a top-down maze exploration whose goal is to advance through the levels. The images above are screenshots of games generated by the baseline, which scored 0.3260; the images below are screenshots of games generated by Game-Evolver, which scored 0.4470.

Case #7: Crossy Road from VeriGame. The baseline trace in Figure 14 already lets the bird cross the road and reaches an ending page, but it begins without a start page and its later scene layout changes abruptly. The evolved trace adds a start page, preserves controllable crossing, extends the scene as the bird moves upward, and reaches an ending page. The visible difference is therefore improved flow and scene continuity, not the creation of movement or an ending that the baseline entirely lacked.



Figure 14: Case #7: Crossy Road from VeriGame. The player controls the bird to move upward and dodge cars. The left images are screenshots generated by the baseline, and the right images are those generated by Game-Evolver.

Case #8: Dwarf Fortress from VeriGame. The baseline trace in Figure 15 presents the game interface, but its annotations show that Adventure/Look is inert, a dwarf cannot be placed, and the run only waits for an enemy. The evolved trace places a dwarf through the Create Dwarf action, shows a stage event, and changes the Z-axis to manage another level. These are directly visible state changes rather than interface-only differences.

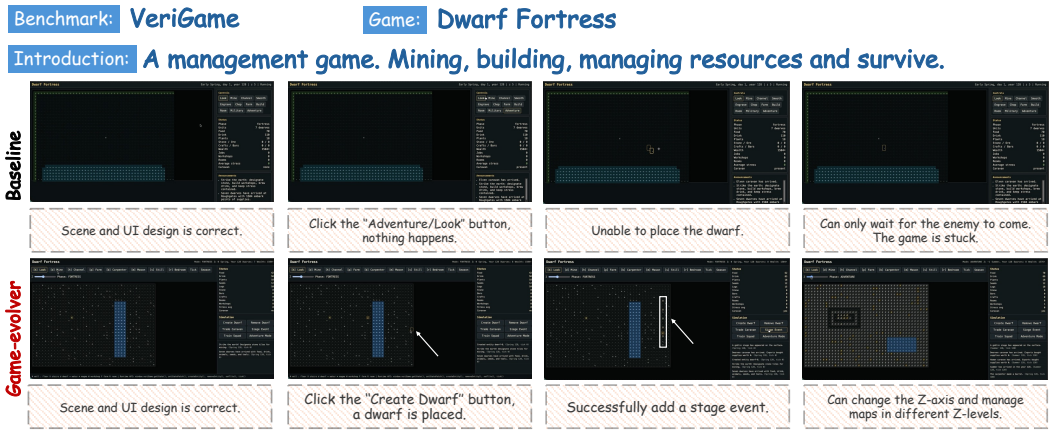


Figure 15: Case #8: Dwarf Fortress from VeriGame. A management game: mine, build, manage resources, and survive. The upper images are screenshots generated by the baseline, and the lower images are those generated by Game-Evolver.

D Application to Real Game Projects

To examine use beyond curated benchmarks, we apply Game-Evolver to godot-tiny-mmo [3], an open-source multiplayer game project built with Godot. Unlike the benchmark tasks, the project already runs, so the target is a feature-level extension rather than recovery of a broken scaffold. Figure 16 directly shows two pieces of evidence: the code diff adds boss-state update logic, and the after frame contains a boss-health bar beneath the dungeon title that is absent from the before frame. The admission uses the same matched-replay schema as elsewhere, and the feedback comes from the framework record, not from the screenshot. The case therefore demonstrates a visible feature addition on a maintained project.

[3] https://github.com/SlayHorizon/godot-tiny-mmo

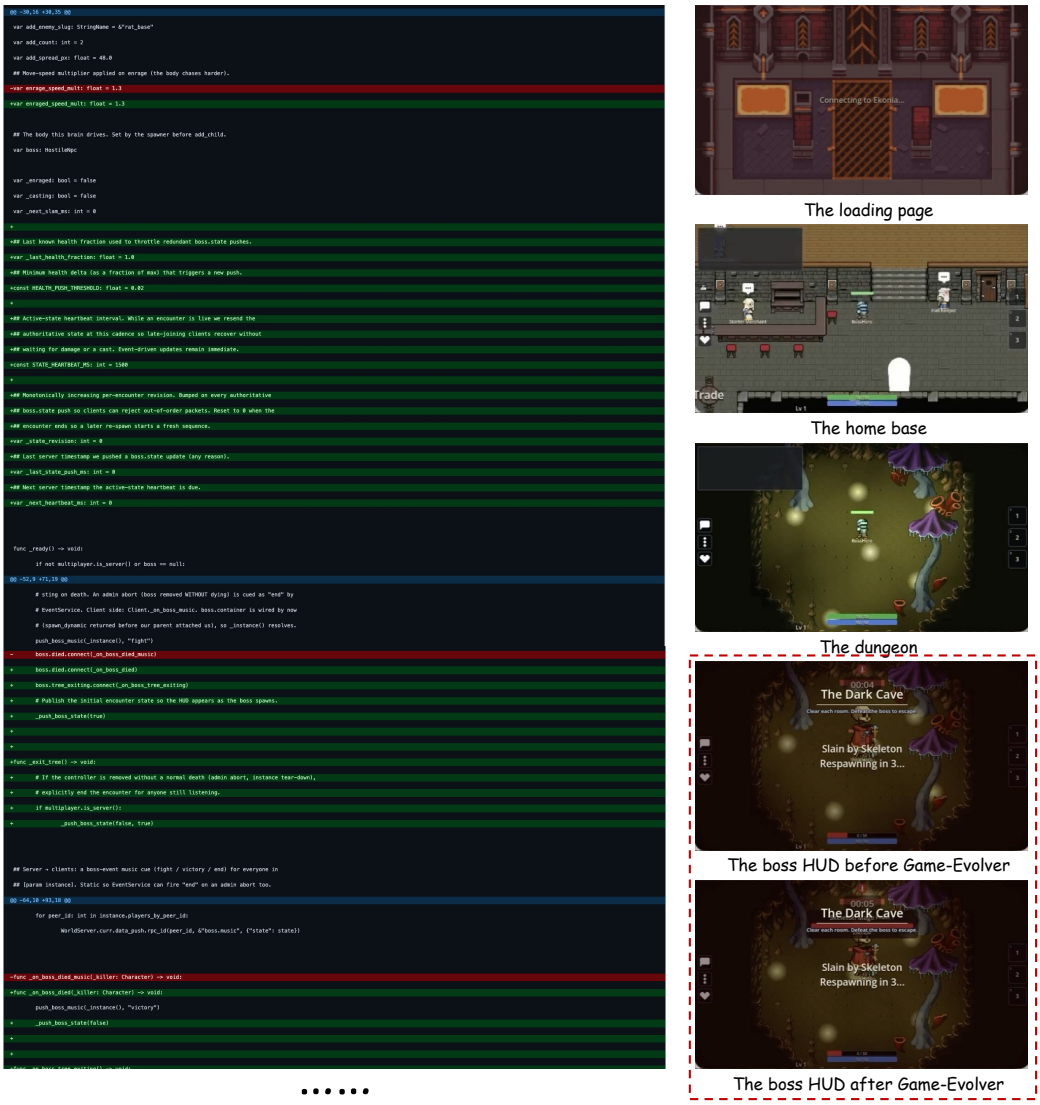


Figure 16: Application of Game-Evolver to an open-source Massively Multiplayer Online (MMO) game. The code diff adds boss-state updates, and the before/after frames show the resulting boss-health bar beneath the dungeon title.