RSIGAME: AUTONOMOUS AGENTIC GAME DEVELOPMENT WITH RECURSIVE SELF-IMPROVEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances in large language models have made automatic game generation increasingly feasible, yet reliably improving generated games beyond a playable version remains challenging. Naive iterative refinement can easily overfit a small set of test cases, producing fragile games with unresolved bugs, missing behaviors, and poor generalization to broader player interactions. We introduce RSIGAME, an autonomous agentic game development framework with recursive self-improvement. RSIGAME organizes development into complementary local and global loops. Concretely, a local *explore-diagnose-improve* loop broadly explores the executable game, diagnoses and prioritizes discovered issues, and performs evidence-grounded revision, where an evolving checklist continually accumulates new testing and improvement guidance. A global loop tracks overall quality, preserves the best checkpoint, and detects saturation or regression over long-horizon development. Beyond test-time improvement, RSIGAME further internalizes successful development experience into the generator through training. Across 140 GameCraft-Bench tasks, two game engines, and five generators, RSIGAME consistently improves game quality under matched development budgets. Notably, experience internalization enables Qwen3.8-27B to reach 61.38 on Godot and 58.53 on Phaser, exceeding GPT-5.5 one-shot scores while reducing Qwen's generation tokens by 11 times.

Anonymous Page **Anonymous Code**

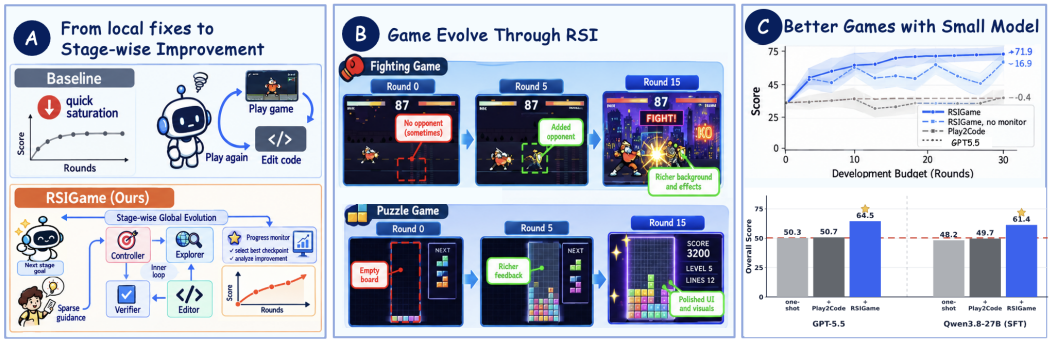

Figure 1: **RSIGAME turns agentic game development into autonomous recursive self-improvement.** It combines a local explore-diagnose-improve loop with global progress monitoring and control, enabling Qwen3.8-27B approach GPT-5.5-level performance.

1 INTRODUCTION

Recent advances in large language models (LLMs) and vision-language models (VLMs), have substantially expanded the capability of autonomous agents to perform complex real-world tasks, ranging from software engineering to computer-use automation Yang et al. (2024); Zhu et al. (2026).

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

Automatic game development represents another promising domain, as building a complete game requires jointly designing game mechanics and progression, create source code and multimodal assets, and organize these heterogeneous components into a coherent executable project Luo et al. (2026). It has become increasingly realistic with the rapid improvement of long-horizon generation, multimodal understanding, and tool-use capabilities in foundation models. Accordingly, recent benchmarks show that VLM-based agents can already turn natural-language specifications into playable games with recognizable mechanics and visual content (Luo et al., 2026; Chi et al., 2026).

However, game development remains highly challenging due to its nature as a complex multimodal engineering process. Even in mature human teams, unexpected bugs, missing functionalities, and overlooked corner cases frequently arise during development Roque et al. (2025). A common solution is therefore to adopt an iterative development workflow, in which developers repeatedly test the current build, collect feedback, and revise the project until it reaches the desired quality. This process can be naturally viewed as a form of recursive self-improvement Wu et al. (2024), where each development round builds upon the previous version based on newly observed evidence. Inspired by this workflow, we formulate agentic game development as an iterative loop: an agent first generates an initial game, then repeatedly tests and improves it until the target requirements are met.

Despite its natural fit for agentic game development, we find that recursive self-improvement can easily converge to fragile solutions that roughly satisfy the target while leaving many untested bugs, missing behaviors, and corner cases unresolved. In effect, the development loop may overfit a small set of test cases rather than improve the game as a whole, leading to poor generalization to broader player interactions. To address this issue, we strengthen the testing stage to broadly explore remaining failures and potential improvements, with the goal of developing a high-quality game rather than merely a playable one. Concretely, we devise an ***explore-diagnose-improve loop***: the agent first broadly explores the executable game to uncover bugs, weaknesses, and opportunities for enhancement; it then diagnoses the discovered issues and prioritizes them to plan the next round of improvement. Throughout this process, we maintain an *evolving checklist* that continually accumulates newly identified issues and actionable guidance for subsequent testing and revision. By expanding as new evidence is discovered, this checklist persistently pushes the development process beyond the limited test cases and toward more generalizable improvements in overall game quality.

To this end, we introduce RSIGAME (Figure 2), an autonomous agentic game development framework with recursive self-improvement. At its core, RSIGAME organizes development into complementary local and global loops. The local loop instantiates our explore-diagnose-improve process, repeatedly exploring the executable game, diagnosing and prioritizing discovered issues, and revising the project based on accumulated evidence. The global loop monitors development across iterations, tracks overall game quality, preserves the best checkpoint, and detects saturation or regression, enabling the system to maintain long-horizon progress rather than blindly continue editing. Beyond training-free self-improvement, RSIGAME further introduces training-based experience internalization, converting successful development trajectories, diagnostic decisions, and verified revisions into supervision for the generator. Such a way enables a broader RSI loop to boost the underlying generator. Across 140 GameCraft-Bench tasks, two engines, and five generators, RSIGAME consistently improves games under matched development budgets. With iterative development, Qwen3.8-27B reaches 47.77 on Godot versus Kimi-K2.6's 44.77 (Moonshot AI, 2026); experience internalization lifts it to 61.38 versus Codex+GPT-5.5's 50.26 one-shot score (OpenAI, 2025; 2026), while cutting generation tokens by 11 times. On Phaser, it rises to 50.24, past the 49.44 one-shot score of GPT-5.5.

2 PRELIMINARIES

Agentic Game Development. It aims to automate game creation with intelligent agents. Given a natural-language game specification, an agent translates user intent into a complete and playable game. Similar to human game development, agentic game development must satisfy two objectives: following user requirements and producing a functionally correct, bug-free game. Formally, given a game specification x, we aim to learn a generator policy π_θ that produces a game project

$$P_0 \sim \pi_\theta(\cdot \mid x), \tag{1}$$

where P_0 should both conform to the intended game design and execute correctly. Such a project consists of source code, visual and textual assets, and configuration files. Its construction therefore

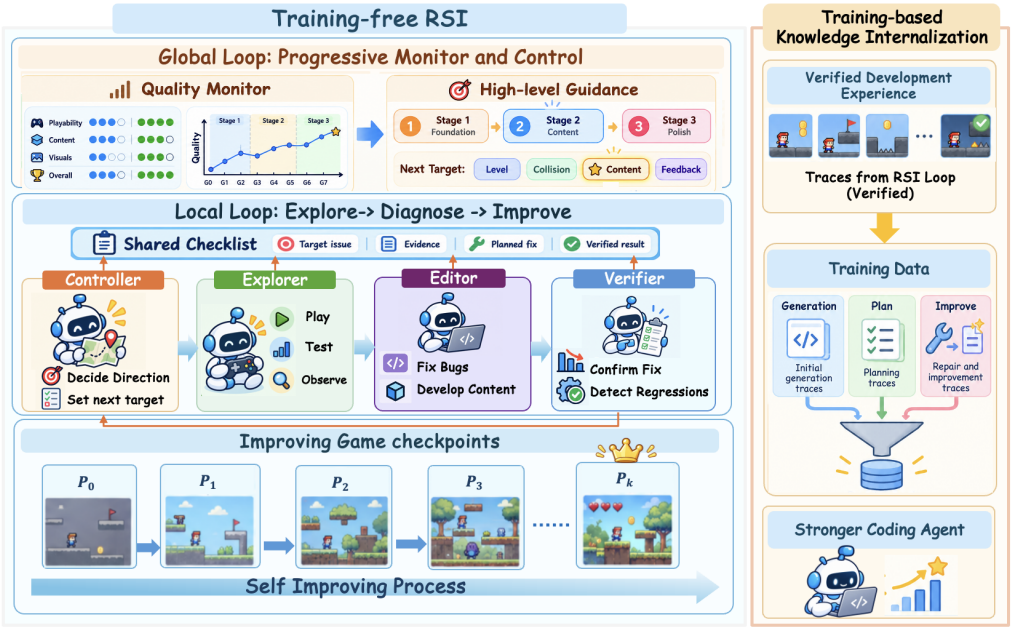

Figure 2: Overview of RSIGAME. The local loop autonomously evolves the game through direction decision, active exploration, evidence-grounded editing, and agentic verification. The global loop monitors global quality, preserves the best checkpoint, and introduces sparse high-level guidance when progress saturates, enabling progressive multi-stage game evolution.

requires joint generation and integration of coding, text, and visual elements into a coherent interactive system. Importantly, game quality is ultimately reflected in runtime behavior and player experience. Each generated game includes a set of replayable demonstration traces $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, which are replayed during evaluation and scored against a hidden task-specific rubric.

Our Focus. Because game generation involves many tightly coupled components, producing a fully correct project in one pass is difficult. Existing workflows therefore commonly adopt an iteration-based strategy: $P_0 \to P_1 \to \cdots \to P_T$. At each iteration, the current project is executed and tested, observed failures are diagnosed, and the project is revised accordingly. In conventional game development, this loop relies on human developers and playtesters. In this work, we bring the same principle into agentic game development, but replace human-driven testing and revision with a *fully autonomous process* that recursively tests, diagnoses, and improves its own generated game.

3 RSIGAME

Our RSIGAME incorporates both training-free and training-based strategies. The training-free strategy builds on local and global loops for iterative development, while the training-based strategy further internalizes accumulated development experience into the backbone game generation model.

3.1 TRAINING-FREE RECURSIVE SELF-IMPROVEMENT

Our training-free RSI strategy consists of a local loop that follows an iterative explore-diagnose-improve process to identify issues and refine the game, and a global loop that monitors long-horizon progress, preserves the best, and detects saturation.

3.1.1 LOCAL LOOP: EXPLORE-DIAGNOSE-IMPROVE

The local loop performs fine-grained recursive game improvement through three stages: autonomous exploration, issue diagnosis, and iterative improvement. The loop continuously explores the exe-

cutable game, accumulates newly discovered issues and improvement opportunities in an evolving checklist, and uses them to guide subsequent revisions.

Controller and Explorer for Autonomous Exploration. The autonomous exploration stage involves two agents with complementary roles. The controller is responsible for deciding what aspect of the game should be explored next, while the explorer interacts with the executable game to collect concrete behavioral evidence. Specifically, given the game specification x, the current project P_t, the development checklist $\mathcal{C}_t$, and optional stage-level guidance γ_s, the controller proposes a development direction d_t. Conditioned on this direction, the explorer performs targeted interaction with the game and produces an exploration trajectory τ_t:

$$d_t \sim \pi_{\text{controller}}(\cdot \mid x, P_t, \mathcal{C}_t, \gamma_s), \qquad \tau_t \sim \pi_{\text{explorer}}(\cdot \mid P_t, d_t). \tag{2}$$

Through this process, the controller provides high-level exploration intent, while the explorer grounds it in observed gameplay, uncovering concrete bugs, missing behaviors, and potential improvement opportunities for subsequent diagnosis and revision.

Editor and Verifier for Issue Diagnosis. The issue diagnosis stage involves two agents with complementary responsibilities. The editor converts the discovered issues and interaction evidence into concrete game modifications, while the verifier evaluates whether these modifications resolve the intended problems without introducing regressions. Specifically, conditioned on the current project, development direction, and observed playtest trajectory, the editor proposes an edit

$$\Delta_t \sim \pi_{\text{editor}}(\cdot \mid P_t, d_t, \tau_t), \qquad P_{t+1} = \text{Apply}(P_t, \Delta_t). \tag{3}$$

The verifier then interacts with the updated project and produces a verification outcome

$$v_t \sim \pi_{\text{verifier}}(\cdot \mid P_{t+1}, d_t, \tau_t), \tag{4}$$

which determines whether the targeted issue has been successfully addressed and whether the edit causes unintended side effects. The verified outcome is subsequently written back to the evolving checklist, providing updated evidence for the controller to plan the next development round.

Iterative Improvement with Checklist Update. To connect exploration, diagnosis, and revision across iterations, RSIGAME maintains a shared development checklist $\mathcal{C}_t$ as an evolving working state. The checklist records discovered issues, improvement opportunities, priorities, and verification outcomes, allowing all agents to operate on a consistent view of the current development status. At each round, the controller reads $\mathcal{C}_t$ to determine the next development direction d_t, the explorer augments it with newly observed evidence, the editor addresses prioritized items, and the verifier updates their status according to the resulting gameplay behavior. These interactions produce the next-round checklist

$$\mathcal{C}_{t+1} = U(\mathcal{C}_t, d_t, \tau_t, \Delta_t, v_t), \tag{5}$$

which is passed to the controller together with P_{t+1}. In this way, RSIGAME continually accumulates development knowledge and ensures that each iteration builds on verified progress rather than restarting from scratch.

3.1.2 GLOBAL LOOP: PROGRESS MONITORING AND CONTROL

While the local loop focuses on improving the game within each development round, the global loop governs the overall development process across stages. It consists of two components: the game quality monitor that tracks accumulated game quality and preserves the strongest checkpoint, and the progress and convergence control mechanism that determines whether development should continue, terminate, or enter a new stage under additional high-level guidance.

Game Quality Monitor. The game quality monitor tracks game-level progress across local development rounds. A stage begins from the checkpoint the previous stage retained, $P_s^\star \leftarrow P_{s-1}^\star$. At each global evaluation point, the monitor compares the checkpoint just produced against the retained one and updates

$$P_s^\star \leftarrow \text{SelectBest}\left(P_s^\star, P_{t+1}\right). \tag{6}$$

Unlike the local Verifier, which assesses whether a specific edit resolves its targeted issue, the game quality monitor evaluates the accumulated quality of the game as a whole and maintains a persistent best state throughout development. Importantly, this monitoring process is strictly isolated from the benchmark evaluator, including its scores, rubrics, and feedback. Details are in Appendix C.2.

Progress and Convergence Control. The evolution of the retained checkpoint $P_s^\star$ provides a natural signal for controlling long-horizon development. If successive global evaluations fail to produce a better checkpoint, RSIGAME treats the persistent lack of progress as convergence. The system then either terminates and returns $P_s^\star$ as the final game, or optionally introduces new high-level guidance γ_{s+1} from a human or stronger model. Such guidance provides strategic directions or creative suggestions rather than explicit edit instructions. The next development stage therefore starts from the retained best checkpoint $P_s^\star$ under γ_{s+1}, allowing the local loop to explore new improvement directions without discarding previously verified gains. Repeating this process enables RSIGAME to control long-horizon progress while preserving the strongest solution reached so far.

3.2 TRAINING-BASED KNOWLEDGE INTERNALIZATION

During training-free RSI, RSIGAME also accumulates reusable experience about how games are planned, implemented, tested, and refined. Thus, we further internalize successful RSI knowledge into the backbone model, extending self-improvement from context optimization to parameter optimization. This broader RSI loop improves the underlying generator, enabling stronger initial generation and reducing the number of refinement iterations required for high-quality game development.

Development Experience Collection and Curation. We collect three types of supervision: *generation traces* from GPT-5.5 (OpenAI, 2026) constructing games in Godot (Godot Engine contributors, 2026) and Phaser (Photon Storm, 2026), *planning traces* collected from successful generations, and *improvement traces* produced by running RSIGAME with GLM-5.3-Flash (Zhipu AI, 2026). We retain only executable generations and improvements that are verified through post-edit interaction without breaking previously functional behavior. In total, we obtain 2,213 generation traces, 2,108 planning traces, and 2,013 verified improvement rounds from 4,003 candidates.

Experience Internalization. We apply supervised fine-tuning to Qwen3.8-27B (Qwen Team, 2026) on the curated planning, generation, and improvement trajectories. Beyond learning from final game artifacts, the model internalizes the intermediate planning, tool-use, diagnosis, and revision decisions that drive successful development. This transfers reusable RSI knowledge into model parameters, strengthening the refinement ability and reducing the required number of iterations to reach high-quality game generation. Such a way opens the door to continual RSI, in which newly acquired development experience can be repeatedly internalized to further improve the generator over time. Training details are provided in Appendix E.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmark and Engines. We evaluate on GameCraft-Bench (Luo et al., 2026), which contains 140 game-development tasks spanning 15 game families. We instantiate the same tasks in both Godot (Godot Engine contributors, 2026) and Phaser (Photon Storm, 2026), keeping the task specifications, evaluation rubrics, and development protocol fixed across engines. Phaser therefore serves as a cross-engine robustness evaluation of the findings on Godot.

Evaluation Protocol. Following GameCraft-Bench, each generated game is packaged with replayable demonstration traces provided by the game generator, which are replayed and scored against a hidden task-specific rubric over *Mechanics* (M), *Depth* (D), *Visuals* (V), and *Art* (A):

$$Q = \mathrm{BUILD} \times (0.15M + 0.35D + 0.15V + 0.35A),$$

where $\mathrm{BUILD} \in \{0, 1\}$ assigns zero score to non-playable games. We use Qwen3.8-27B (Qwen Team, 2026) as the judge and average three independent replay-and-score runs. The hidden rubric, scores, and judge feedback are never exposed to the development agents. Evaluation stability and free-play robustness are detailed in Appendices D.3 and D.4.

Baselines and Development Budget. We compare the frozen initial project P_0, with Play2Code (Huang et al., 2026), and with RSIGAME across multiple generators. For each generator, all development methods start from an identical clone of P_0 and use the same per-round tool

Table 1: **Main results on GameCraft-Bench (top: Godot; bottom: Phaser).** Methods are compared from matched base games under one backbone and budget, averaged over 140 tasks; **Tok.** and **Cost** are mean billable tokens and cost per task (Appendix D.1). **Bold** marks the best value per column within a group; deeper rows internalize development experience.

Method	Mechanics	Depth	Visuals	Art	Overall ↑	Tok. ↓	Cost ↓
Godot (Godot Engine contributors, 2026)							
Generator: Codex + GPT-5.5 (high)							
Base (frozen P_0)	58.9	51.7	51.4	44.6	50.26	0.26M	–
+ Play2Code	59.2	51.3	52.2	45.8	50.74	3.20M	$1.06
+ RSIGAME	**72.8**	**60.3**	**65.9**	**64.5**	**64.53**	**3.12M**	**$0.88**
Generator: Codex + Kimi-K2.6							
Base (frozen P_0)	40.2	30.3	35.4	21.9	29.63	3.22M	–
+ Play2Code	44.8	35.4	40.2	28.6	35.19	**7.07M**	$1.27
+ RSIGAME	**51.7**	**40.1**	**47.1**	**45.4**	**44.77**	7.39M	**$1.18**
Generator: Codex + GLM-5.3-Flash							
Base (frozen P_0)	35.7	30.2	32.0	24.9	29.46	0.44M	–
+ Play2Code	47.2	39.7	41.1	32.7	38.59	**3.18M**	**$0.99**
+ RSIGAME	**51.9**	**44.4**	**49.9**	**53.9**	**49.72**	5.06M	$1.25
Generator: Codex + Qwen3.8-27B							
Base (frozen P_0)	41.2	33.4	38.6	38.3	37.07	6.41M	–
+ Play2Code	44.6	36.1	42.1	42.5	40.53	9.92M	$1.32
+ RSIGAME	**51.2**	**40.4**	**47.5**	**53.8**	**47.77**	**9.13M**	**$1.09**
Generator: Codex + Qwen3.8-27B (SFT)							
Base (frozen P_0)	56.1	47.5	49.8	44.8	48.22	0.57M	–
+ Play2Code	57.4	48.1	51.6	47.2	49.71	3.74M	$1.08
+ RSIGAME	**68.7**	**55.8**	**62.8**	**63.2**	**61.38**	**3.49M**	**$0.94**
Phaser (Photon Storm, 2026)							
Generator: OpenGame + GPT-5.5							
Base (frozen P_0)	49.1	38.6	53.6	58.6	49.44	5.36M	–
+ Play2Code	57.1	42.0	60.0	65.2	55.10	**8.34M**	**$1.43**
+ RSIGAME	**59.8**	**44.2**	**62.4**	**69.7**	**58.21**	8.38M	$1.44
Generator: OpenGame + Qwen3.8-27B							
Base (frozen P_0)	40.6	29.5	43.7	48.7	40.03	9.03M	–
+ Play2Code	51.0	36.4	53.2	58.0	48.70	12.99M	$1.75
+ RSIGAME	**52.0**	**36.5**	**54.6**	**61.3**	**50.24**	**12.78M**	**$1.50**
Generator: OpenGame + Qwen3.8-27B (SFT)							
Base (frozen P_0)	47.1	35.4	48.9	52.4	45.14	5.42M	–
+ Play2Code	55.6	41.7	57.3	61.8	53.15	9.58M	$1.61
+ RSIGAME	**60.4**	**46.2**	**62.3**	**68.4**	**58.53**	**9.31M**	**$1.53**

budget. Both Play2Code (Huang et al., 2026) and RSIGAME are allowed up to 26 per-round tool calls and 30 development rounds. Full model configurations, row-specific budgets, cost accounting, and implementation details are provided in Appendix D.1 and Appendix D.2.

4.2 GODOT GAME RESULTS

Table 1 reveals three main findings. *First,* RSIGAME *delivers large and consistent gains across generators.* Across all five settings, it improves the frozen initial games by 10.7–20.3 Overall points, with gains consistently spanning Mechanics, Depth, Visuals, and Art. *Second, under the same compute budget,* RSIGAME *performs significantly better than other recursive-based methods.* Under matched development budgets, RSIGAME outperforms Play2Code by 7.2–13.8 Overall points. Play2Code brings almost no improvement to the strong Codex+GPT-5.5 initialization and Qwen3.8-27B variants, whereas RSIGAME substantially improves the same frozen games. This contrast shows that the benefit comes from how development is organized, rather than from iteration alone. *Finally, experience internalization further improves both quality and efficiency.* Fine-tuning

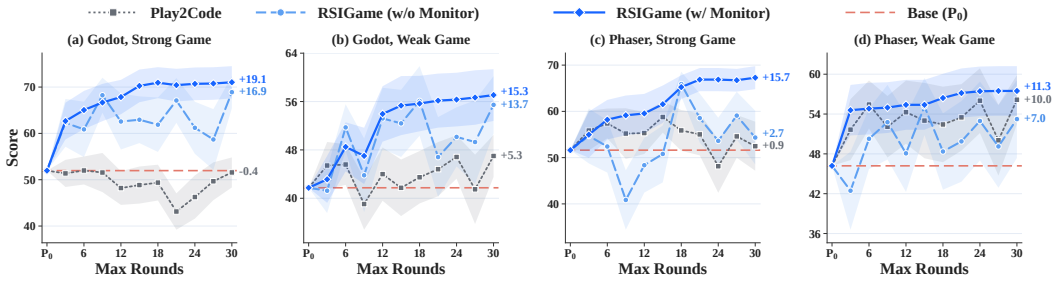

Figure 3: **Development-time scaling on GameCraft-Bench (40 tasks).** Across Godot and Phaser with strong (GPT-5.5) and weak (Qwen3.8-27B) initial generators, iterative development can plateau or regress, while RSIGAME achieves sustained improvement across development budgets. The Global Quality Monitor preserves the best checkpoint reached so far, improving the quality–cost tradeoff over returning the last checkpoint. Shaded regions denote ± 1 standard error.



Figure 4: **Global quality control and experience transfer.** (**a**) Best-checkpoint tracking protects earlier gains from later regressions. (**b**) Saturation-aware stopping achieves comparable quality with fewer development rounds. (**c**) Internalizing verified development experience improves one-shot generation across all quality dimensions.

raises Qwen3.8-27B's one-shot score from 37.07 to 48.22, within 2.0 points of Codex+GPT-5.5, while reducing generation tokens by 11 times (6.41M to 0.57M). With RSIGAME, the score further rises to 61.38 and total token usage decreases by 2.6 times (9.13M to 3.49M).

4.3 PHASER GAME RESULTS

The Phaser block of Table 1 shows that the gains of RSIGAME transfer across engines, improving all three generators by 8.8–13.4 Overall points over their frozen bases and achieving the strongest final quality in every setting. Interestingly, Play2Code is considerably stronger on Phaser than on Godot. The category breakdown reveals why: Phaser initializations tend to have weaker Mechanics and Depth but stronger Visuals and Art, yielding similar Overall scores while leaving more readily improvable functional headroom. Play2Code can therefore simply find these obvious deficiencies and recover substantial quality, narrowing its gap to RSIGAME. Nevertheless, RSIGAME still consistently produces the best final games, indicating that its advantage extends beyond low-level improvement even when such improvement already captures much of the available headroom.

5 FURTHER ANALYSIS

Can Global Quality Monitoring Improve Development-Time Scaling? Figure 3 reveals a clear difference in how development methods scale with additional compute. Simply extending the budget does not guarantee better games: Play2Code quickly plateaus and often regresses as more rounds are added. In contrast, RSIGAME consistently converts additional development rounds into higher game quality across both Godot and Phaser and for both strong and weak initializations. The local loop alone already produces substantial gains, but its trajectory remains volatile; the Global Quality Monitor stabilizes this process by retaining the best state reached so far, yielding sustained improvement throughout the development budget.

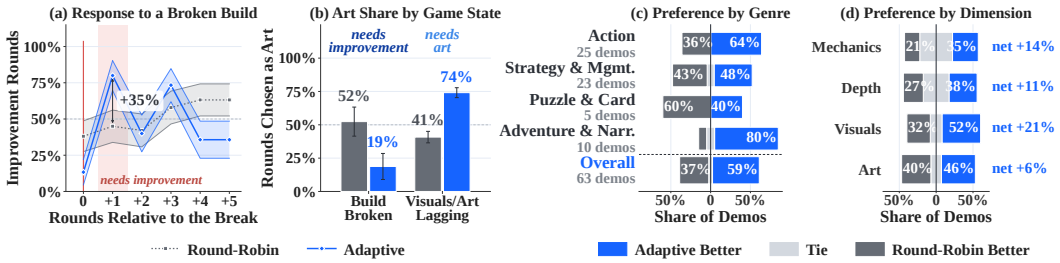

Figure 5: **Adaptive development follows the game state and improves final quality.** (**a, b**) Adaptive development reallocates effort according to the current bottleneck, shifting toward improvement after build failures and toward art when visual quality lags. (**c, d**) Blind pairwise evaluation consistently favors adaptive development over round-robin scheduling.

Method	Grounded Prec. ↑	Ungrounded / Round ↓
Free critic	58.6%	1.93
Agentic verification	**72.3%**	**0.50**

(a) Pre-improvement target grounding, 40 sessions.

Method	Failure Recall ↑	Spec. ↑	Bal. Acc. ↑
Build only	0.0%	**100.0%**	50.0%
Replay + verification	**76.2%**	92.6%	**84.4%**

(b) Post-improvement failure detection, 48 adjudicated rounds.

Table 2: **Agentic verification improves both target grounding and post-edit reliability.** Before editing, it filters unsupported improvement targets; after editing, replay-based verification detects failed changes and regressions that compilation alone cannot reveal.

Figure 4(a)(b) further isolates the benefit of this global control. The retained checkpoint closely tracks the oracle best one within each budget, preventing later edits from erasing earlier gains, while saturation-aware stopping achieves comparable quality to much longer fixed-budget runs with fewer rounds. Together, these results show that the monitor makes development-time scaling both more reliable and more compute-efficient. Detailed stopping criteria are provided in Appendix C.2.

Does Adaptive Evolution Matter? Figure 5 examines whether the local loop benefits from choosing its own development focus adaptively rather than following a fixed round-robin schedule. Adaptive Evolution puts its effort to the current game state: after a build failure it shifts strongly toward improvement, while once visual quality becomes the bottleneck it allocates substantially more rounds to art (Figure 5(a) (b)). This adaptive behavior also leads to better final games: blind pairwise evaluation prefers adaptive development on 59% of comparisons versus 37% for round-robin, with consistent advantages across all four quality dimensions (Figure 5(c)(d)). These results show that long-horizon development benefits from letting the local loop respond to the evolving needs of the game rather than following a fixed schedule. Full evaluation details are provided in Appendix D.6.

Does Agentic Verification Make Local Improvement More Reliable? Table 2 evaluates verification both before and after editing. Evidence-grounded pre-improvement verification raises grounded precision from 58.6% to 72.3% and reduces unsupported targets from 1.93 to 0.50 per round. After editing, replay-based verification detects 76.2% of unsuccessful improvements and reaches 84.4% balanced accuracy, whereas build-only checking detects none of these behavioral failures. Together, these results show that agentic verification improves both the quality of development decisions and the reliability of the feedback returned to subsequent rounds. The audit protocol and metric definitions are given in Appendix D.7.

Can Development Experience Transfer to Future Generation? Figure 4(c) shows that development experience transfers back to the generation model. Training on generation traces alone yields uneven gains across quality dimensions, whereas incorporating planning and verified improvement experience produces consistent improvements in Mechanics, Depth, Visuals, and Art. The gains are particularly pronounced in Mechanics $+14.9$ and Depth $+14.1$, leading to a $+11.1$-point Over-

all improvement before any test-time development. This suggests that planning and improvement trajectories provide reusable knowledge beyond simply imitating successful game generations.

6 Related Work

Game Generation and Development Benchmarks. Recent benchmarks increasingly evaluate agents on complete, executable games rather than isolated code snippets. OpenGame-Bench (Jiang et al., 2026) evaluates web-based games through build validity, visual usability, and instruction alignment; GameCraft-Bench (Luo et al., 2026) extends evaluation to engine-based games with interaction-grounded measures of mechanics, content, visuals, and art; and GameDevBench (Chi et al., 2026) studies multimodal game development over larger codebases and assets. Related work further examines interactions in game-playing agents and broader web- or code-based game generation, through both benchmarks and generation systems (Lin et al., 2026; Hu et al., 2024; Zhang et al., 2025; 2026; Ma et al., 2026). Together, these efforts shift evaluation from static code correctness toward the quality of executable, interactive game artifacts. However, these works score a single submitted build, leaving open how an agent should keep improving a game across rounds of development, which is the setting we study.

Iterative Agentic Game Development. Recent systems extend game generation into iterative development. Play2Code (Huang et al., 2026) alternates between gameplay and code revision, using interaction feedback from each build to guide subsequent edits, while VibeGame (Hu et al., 2026) coordinates multiple specialized agents for continued game development. OpenGame (Jiang et al., 2026) further accumulates reusable development skills across projects to support subsequent generation. These efforts follow broader agentic coding paradigms based on repeated reasoning, execution, and feedback (Yang et al., 2024; Hong et al., 2024; Yao et al., 2023; Shinn et al., 2023). Rather than introducing iteration itself, RSIGame focuses on how iterative development is sustained reliably over long horizons. It combines adaptive, evidence-grounded local improvement with global best-checkpoint tracking and saturation detection. High-level guidance can further reopen development after autonomous progress stalls, enabling multi-stage game evolution.

Verification and Learning from Development Experience. Agentic verification has been studied through general LLM-as-judge protocols (Zheng et al., 2023), reliable process control method in agent loop (Wu et al., 2026), and game-specific methods that validate behavior through runtime interaction (Jia et al., 2026; Huang et al., 2026). In RSIGame, independent verification closes each local development round: the updated game is replayed to confirm the intended improvement and detect new regressions, with verified outcomes written back into the shared development state. Beyond individual tasks, prior work learns from feedback, memory, or verified trajectories (Shinn et al., 2023; Madaan et al., 2023; Chen et al., 2024; Wang et al., 2025; Pan et al., 2026; Zelikman et al., 2022; Zhou et al., 2026), while game systems reuse experience across generations (Jiang et al., 2026; Hu et al., 2026). RSIGame extends this idea by jointly internalizing planning, generation, and verified improvement experience to strengthen future game generation and development.

7 Conclusion

We presented RSIGame, an autonomous agentic game development framework that formulates game creation as a recursive self-improvement process. RSIGame strengthens the development loop with broad exploration, structured diagnosis, prioritized improvement, and an evolving checklist that continuously expands the set of identified issues and improvement opportunities. Its local explore-diagnose-improve loop enables evidence-grounded revision of the executable game, while the global loop tracks long-horizon progress, preserves the best checkpoint, and detects saturation or regression. We further introduced training-based experience internalization to transfer successful development trajectories back into the backbone game generation model, improving future game creation beyond test-time refinement alone. Experiments across 140 GameCraft-Bench tasks, two game engines, and five generators demonstrated that RSIGame consistently improves game quality under matched development budgets, enabling smaller open models to reach or even surpass substantially stronger generators after iterative development.

AI USE STATEMENT

Large language models are integral to the research methodology of this work. GPT-5.5 is used for game generation and planning-data collection; GLM-5.3-Flash is used for game improvement, verification, and improvement-data collection; and Qwen3.8-27B serves as the gameplay test agent, the benchmark evaluation judge, and the base model for experience internalization. Claude Opus 5 is additionally used for the independent verification audit.

LLM-based assistants were also used during research and manuscript preparation for language editing, presentation refinement, and programming assistance. The authors designed the experiments, reviewed the resulting artifacts and measurements, and verified **all** reported results and scientific claims. The authors take full responsibility for the content of the paper.

ETHICS STATEMENT

Our study focuses on general-purpose game-generation and game-development tasks. We apply safety constraints during generation and exclude artifacts containing unsafe or sensitive content from evaluation and data curation. The experiments do not use private user data, personal information, or user-generated content.

Human involvement is limited to game evaluation and high-level development guidance. We study the resulting game assessments and guidance rather than participant attributes, and collect no sensitive personal information. Released datasets and artifacts are anonymized and screened to remove machine-specific paths, credentials, and identifying information.

REPRODUCIBILITY STATEMENT

All reported results are traceable to retained experimental artifacts and documented protocols. Table 3 provides a single index to the released artifacts and to the appendix sections specifying each component of the experimental pipeline.

Controlled comparisons. For each comparison, the initial project P_0 is frozen and every development method receives an identical clone, the same improvement backbone, and the same per-round tool budget. Row-specific development budgets are reported explicitly in Appendix D.1. Each resulting game is independently replayed and scored three times, while benchmark rubrics, scores, and judge outputs remain hidden from all development agents. Replay and judge variability are quantified in Appendix D.3.

Artifact availability. The full implementation, prompts, and evaluation pipeline are available in the anonymous repository, and representative evolved games are playable from the anonymous project page. Frozen base projects, training trajectories, and per-round run trees have been prepared and anonymized for release. We also retain the complete evaluation artifacts underlying the reported results, including 51,644 scoring files containing per-task scores, judge outputs, and replay reports. Their release status and corresponding protocol documentation are summarized in Table 3.

REFERENCES

Anthropic. Claude Opus 5 system card. https://www.anthropic.com/claude-opus-5-system-card, July 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2304.05128.

Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. Gamedevbench: Evaluating agentic capabilities through game development. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. arXiv:2602.11103.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.14314.

Godot Engine contributors. Godot engine. https://godotengine.org, 2026.

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 (ICLR)*, 2024. arXiv:2308.00352.

Chengpeng Hu, Yunlong Zhao, and Jialin Liu. Game generation via large language models. In *2024 IEEE Conference on Games (CoG)*, 2024.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2106.09685.

Wenbo Hu, Ken Li, Jiazhe Wei, Yukang Cao, Weiyi Hong, Jiayi Dai, Chenjun Bai, Jiajun Liang, Yucheng Liao, Ruichuan An, Zeyu Lou, Haofan Wang, Yueming Lyu, Ziwei Liu, and Chenyang Si. Vibegame: Prompt-to-game development with ai-native engine and self-evolving adversarial agent team. Technical report, 2026. https://vibegame.tettet.org/technical_report.pdf.

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, 2026.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

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, 2026.

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, 2026.

Mingxian Lin, Shengju Qian, Yuqi Liu, Yi-Hua Huang, Yiyu Wang, Wei Huang, Yitang Li, Fan Zhang, Zeyu Hu, Lingting Zhu, Xin Wang, and Xiaojuan Qi. Omnigamearena: A unified ue5 benchmark for vlm game agents with improvement dynamics, 2026.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. arXiv:1711.05101.

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

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.17651.

Moonshot AI. Kimi-K2.6. Model card, Hugging Face, April 2026. https://huggingface.co/moonshotai/Kimi-K2.6.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, May 2025.

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

Wenbo Pan, Shujie Liu, Xiangyang Zhou, Shiwei Zhang, Wanlu Shi, Mirror Xu, and Xiaohua Jia. M*: Every task deserves its own memory harness, 2026.

Photon Storm. Phaser: A fast, free and fun open source html5 game framework. https://phaser.io, 2026.

Qwen Team. Qwen3.8-27B. Model card, Hugging Face, August 2026. https://huggingface.co/Qwen/Qwen3.8-27B.

Alejandro Roque, Juan P. Sotomayor, Dionny Santiago, and Peter J. Clarke. A literature review of software testing practices and frameworks in the video gaming industry. *Software Testing, Verification and Reliability*, 35(2):e70001, 2025. doi: 10.1002/stvr.70001.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.11366.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. Spotlight. arXiv:2506.03136.

Tianhao Wu, Weizhe Yuan, Olga Golovneva, Jing Xu, Yuandong Tian, Jiantao Jiao, Jason Weston, and Sainbayar Sukhbaatar. Meta-rewarding language models: Self-improving alignment with LLM-as-a-meta-judge, 2024. https://arxiv.org/abs/2407.19594.

Wenyi Wu, Sibo Zhu, Kun Zhou, Aayush Salvi, Zixuan Song, and Biwei Huang. Structagent: Harness long-horizon digital agents with unified causal structure, 2026. https://arxiv.org/abs/2607.11388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv:2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. arXiv:2203.14465.

Wei Zhang, Jack Yang, Renshuai Tao, Lingzheng Chai, Shawn Guo, Jiajun Wu, Xiaoming Chen, Ganqu Cui, Ning Ding, Xander Xu, Hu Wei, and Bowen Zhou. V-gamegym: Visual game generation for code large language models, 2025.

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

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023. arXiv:2306.05685.

Zhipu AI. GLM-5.3-Flash. Model card, Hugging Face, August 2026. https://huggingface.co/zai-org/GLM-5.3-Flash.

Chenyu Zhou, Qiliang Jiang, Shuning Wu, and Xu Zhou. The verifier is the curriculum: Precision sets the return on search in code self-distillation, 2026.

Sibo Zhu, Shicheng Fan, Xinyue Wang, Wenyi Wu, Kun Zhou, and Biwei Huang. Rsiagent: Autonomous exploration for recursive self-improvement in new environments, 2026. https://arxiv.org/abs/2609.15364.

Appendix Overview

This appendix provides implementation details, experimental protocols, training specifications, and extended analyses supporting the main paper.

A Reproducibility Index **15**

Artifacts, release status, and where each protocol detail is specified.

B Limitations **15**

Scope of the autonomous pipeline, the dimensions evaluated, and project size.

C RSIGAME System Design and Implementation **16**

Complete specification of the recursive development system and its agent interfaces.

C.1 Recursive Development Procedure 16
C.2 Global Quality Monitoring 16
C.3 High-Level Director Interface 17
C.4 Agent Prompts and Output Contracts 20

D Experimental Protocols and Evaluation Reliability **24**

Configuration, accounting, and reliability protocols underlying the reported experiments.

D.1 Experimental Configuration 24
D.2 Token and Cost Accounting 24
D.3 Replay and Scoring Stability 25
D.4 Free-Play Evaluation 28
D.5 Statistical Reliability of Main Results 28
D.6 Adaptive Direction Policy Evaluation 29
D.7 Agentic Verification Audit 30
D.8 Multi-Stage Guidance Evaluation 31

E Training Data and Experience Internalization **31**

Construction, curation, and internalization of game-development experience.

E.1 Training Corpus Construction and Curation 31
E.2 Fine-Tuning Configuration 33
E.3 Training-Data Ablation 33

F Extended Evaluation and Case Studies **34**

Extended comparisons, qualitative trajectories, and task-family breakdowns.

F.1 Comparison with a Multi-Agent Generate-and-Verify System 34
F.2 Qualitative Evolution Case Studies 35
F.3 Per-Family Results 43

A REPRODUCIBILITY INDEX

Table 3 is the single index referred to by the Reproducibility Statement: for each artifact it gives what the artifact contains, whether it is released, and where the corresponding protocol or implementation detail is specified.

Table 3: **Reproducibility index.** Artifacts, release status, and locations of the corresponding implementation and protocol details.

Item	Contents / specification	Location
Released artifacts		
Implementation	Local/global loops, Global Quality Monitor, director-review tool, and scoring harness	Code, live
Prompts and output contracts	Complete prompts and structured outputs used by all agents	Code, live; App. C.4
Playable evolution	Six representative games with retained checkpoints	Project page, live
Frozen base projects P_0	Starting project for each task and generator	On release
Evaluation artifacts	Per-task rubric scores, judge outputs, and replay reports (51,644 files)	Anonymous repo[†]
Training corpus	2,213 generation trajectories, 2,108 plans, and 2,013 verified improvement rounds	On release
Run trees	Per-round project snapshots for long-horizon development runs	On release
Protocol and implementation details		
Recursive development loop	One local round, global checkpoint update, and stopping logic	App. C.1
Global Quality Monitor	Best-checkpoint selection, saturation criterion, and patience K	App. C.2
Model assignment	Models used for generation, exploration, editing, verification, and judging	Tab. 6
Hyperparameters	Loop, monitor, and scoring parameters	Tab. 7
Development budgets	Rounds and tool-call budgets for each reported row	App. D.1
Token and cost accounting	Billable tokens, excluded costs, and pricing	App. D.2
Scoring stability	Replay and judge variability	App. D.3
Corpus construction	Extraction, verification filtering, and contamination checks	App. E.1
Fine-tuning	Training mixture and hyperparameters	App. E.2
Director review	Director inputs, interaction budget, and guidance schema	App. C.3
Per-family results	Main-table results broken down by task family	App. F.3

[†] Anonymous evaluation repository: `anonymous312874/rsigame-scoring-artifacts`. Machine-specific paths and identifying terms are removed before release, and the exported files are re-parsed and re-scanned.

B LIMITATIONS

The core RSIGAME pipeline is fully autonomous: adaptive local development, global quality monitoring, best-checkpoint tracking, and saturation-aware stopping require no human intervention. We additionally study an optional directed extension in which sparse high-level guidance from a human or a stronger model is introduced only after autonomous improvement saturates. Our current study evaluates this mechanism on a limited set of multi-stage development cases; a broader investigation of when guidance should be invoked, how much guidance is beneficial, and how human and model guidance differ remains future work.

Our evaluation is also centered on the four dimensions of Mechanics, Depth, Visuals, and Art. These dimensions do not fully capture higher-level properties such as originality, narrative quality, long-term player engagement, or subjective enjoyment.

Finally, our experiments focus on relatively compact games that can be iteratively developed within practical agent budgets. Extending autonomous recursive development to substantially larger projects with longer horizons, richer assets, and more complex cross-system dependencies remains an important direction.

C RSIGAME SYSTEM DESIGN AND IMPLEMENTATION

C.1 RECURSIVE DEVELOPMENT PROCEDURE

Algorithm 1 is the loop of Section 3 written out: one round of the inner loop, the checkpoint at which the Global Quality Monitor runs, and the two things saturation can lead to – returning the champion, or taking a piece of guidance and opening the next stage.

The multi-stage structure is the counter c and the stage index s. A checkpoint either promotes a new champion, which sets c back to zero, or finds nothing better, which advances it; K checkpoints in a row without a promotion is what the paper calls saturation. Saturation does not by itself end development: it ends the *stage*. If guidance is available, a brief is written on the champion, the next stage restarts from that champion rather than from the last round's build, c is reset, and the loop continues with γ_{s+1} steering every subsequent round. Development ends only when a stage saturates and no further guidance is given. The runs in this paper use $E = 3$ and $K = 3$.

Algorithm 1 RSIGAME Recursive Game Development

Require: Game specification x, initial project P_0, initial checklist $\mathcal{C}_0$, checkpoint interval E, patience K

$P_0^\star \leftarrow P_0,\ \gamma_0 \leftarrow \varnothing,\ t \leftarrow 0,\ s \leftarrow 0,\ c \leftarrow 0$ ▷ champion, guidance, round, stage, idle checkpoints
while development budget remains **do**
  $d_t \sim \pi_{\text{controller}}(\cdot \mid x, P_t, \mathcal{C}_t, \gamma_s)$ ▷ the checklist and guidance steer the round
  $\tau_t \sim \pi_{\text{explorer}}(\cdot \mid P_t, d_t)$
  $\Delta_t \sim \pi_{\text{editor}}(\cdot \mid P_t, d_t, \tau_t), \quad P_{t+1} \leftarrow \text{Apply}(P_t, \Delta_t)$
  $v_t \sim \pi_{\text{verifier}}(\cdot \mid P_{t+1}, d_t, \tau_t)$ ▷ is the improvement observable, and what regressed
  $\mathcal{C}_{t+1} \leftarrow U(\mathcal{C}_t, d_t, \tau_t, \Delta_t, v_t)$ ▷ the working memory the next round builds on
  if $t \bmod E = 0$ **then** ▷ a checkpoint: the Monitor compares, the loop does not
    $P' \leftarrow \text{SelectBest}(P_s^\star, P_{t+1})$
    if $P' \neq P_s^\star$ **then**
      $P_s^\star \leftarrow P'$; $c \leftarrow 0$ ▷ a new champion; the stage is still paying
    else
      $c \leftarrow c + 1$ ▷ another checkpoint with nothing better
    end if
    if $c \geq K$ **then** ▷ saturated: K checkpoints without a new champion
      if no guidance is available **then return** $P_s^\star$
      $\gamma_{s+1} \leftarrow \text{GUIDANCE}(x, P_s^\star)$ ▷ a brief on the champion opens the next stage
      $P_{s+1}^\star \leftarrow P_s^\star,\ P_{t+1} \leftarrow P_s^\star,\ s \leftarrow s + 1,\ c \leftarrow 0$ ▷ the next stage restarts here
    end if
  end if
  $t \leftarrow t + 1$
end while
return $P_s^\star$

C.2 GLOBAL QUALITY MONITORING

What it compares. The Monitor never scores a version on its own. At each checkpoint it replays the retained champion $P_s^\star$ and the candidate build on the *same* set of demonstrations, pairs the two recordings scenario by scenario, and asks only what changed between them. The comparator is not told which round produced the candidate, what the improvement was trying to achieve, or how much budget has been spent: direction and budget are decided one layer up, and supplying them here would let development intent colour what the model reports seeing.

Scale. Each criterion of the proxy rubric is read on three anchored levels — 1.0 works as described, 0.5 works partly, intermittently, or only in some cases, 0.0 does not work — for the champion and the candidate separately, and only on criteria that *both* recordings show, so that a difference is a difference in the game rather than in what the two replays happened to reach. Criterion readings are combined into a weighted overall value in $[0, 1]$, reported as two halves: the criteria from the current stage brief, and those from the task document.

Adoption. The candidate replaces the champion when the brief half improves by at least 0.05 and the task half falls by no more than 0.05; otherwise the champion is kept. The task half acts as a floor rather than a target, so a build may be adopted for advancing the stage's own goals provided it does not regress what the task already required. Without stage criteria the rule reduces to a 0.05 gain on the overall value.

Checkpoints and stopping. Checkpoints are taken every three rounds, which is the shortest interval that contains an improvement, its verification and a replay, and therefore the shortest window in which a difference can be observed at all. Development stops once the champion has gone unchanged for K consecutive checkpoints; we use $K = 3$. Figure 4 (b) places $K = 1 \dots 4$ against fixed budgets on the same runs: smaller K stops sooner and delivers less, larger K approaches the fixed-budget score at a fraction of its rounds, and $K = 3$ sits at the knee.

Isolation from the evaluator. The Monitor and the benchmark judge share only the replay mechanism. The Monitor reads a separate proxy rubric, never the benchmark's held-out rubric, and never its scores or feedback; the module does not import the judge at all. The separation is also internal: the comparator is the only component that sees pixels, and the stage that decides direction and stopping reads its structured counts rather than the frames, so neither layer can both interpret an image and act on its own interpretation.

C.3 High-Level Director Interface

Section 3 describes what a stage brief does to the loop. This appendix describes where briefs come from. Every brief reported in this paper — human and model alike — was written in the interface below, under one protocol, so that "a person directed this run" and "a model directed this run" differ in the director and in nothing else.

C.3.1 What the Director Is Given

A director reviews one build of one game. Figure 6 is the page: the game runs live in the left panel and takes the keyboard when the frame is focused; the right column holds the budget, the task specification the game was generated from, and a summary of the development that has already happened; the form below is the brief.

The summary is deliberately thin. It reports how many autonomous rounds have run, which of them became champions at the Global Quality Monitor's three-round checkpoints, which build is under review, and how those rounds divided between functional and presentational work. It reports no score, at any round, for any build.

That omission is the point. The director is asked what the game needs next, and a score would answer a different question — whether the last round worked — in a currency the loop already optimises. Nothing else about the run reaches the page either: no source, no logs, no improvement transcripts, no other director's review. The instrument shows a playable game and the document it was supposed to become.

C.3.2 One Instrument, One Budget

A director gets 60 actions and 2 resets. A key press costs one action however long it is held, a click costs one, and watching and waiting are free, so the budget buys interaction rather than patience. The counters in Figure 6 are the ones the session enforces, and the count each director spent is stored with the brief.

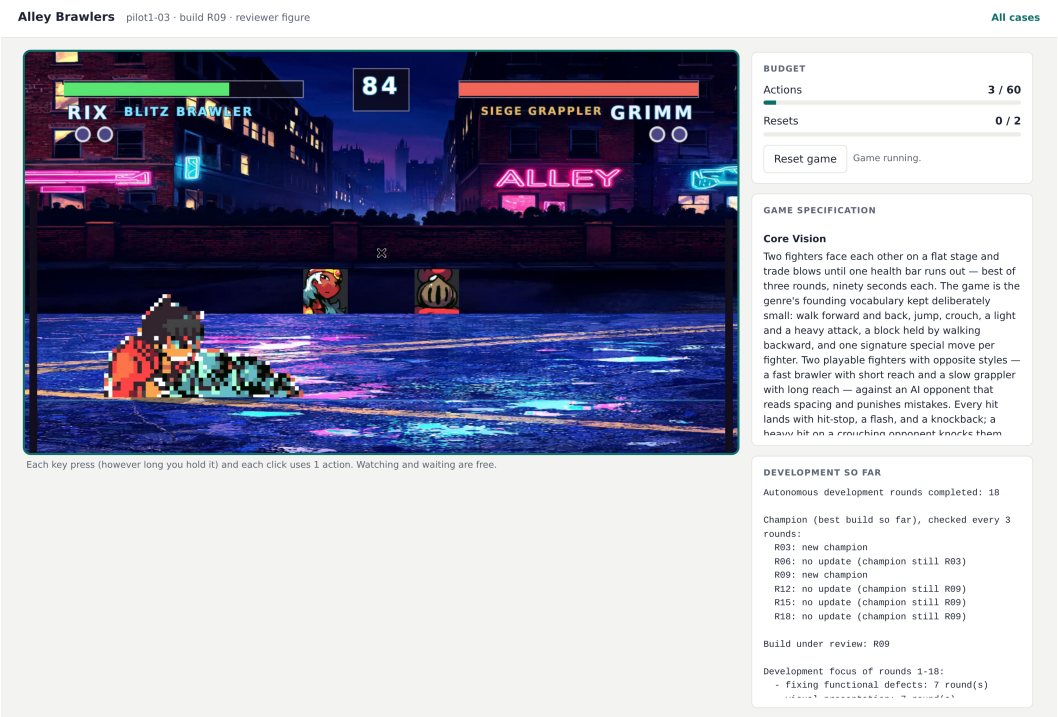

Figure 6: The director review page, mid-session, on the *Alley Brawlers* build that had survived six monitor checkpoints: the live game, the budget, the task specification, and the development summary — champions at R03 and R09, no update since, and the split of the eighteen rounds between functional and presentational work. The frame is a real moment of play and shows the defect both directors went on to describe in different words: the player's fighter has come apart into a band of scrambled sprite tiles. The brief form below it is Figure 7.

A person plays through the browser. A model plays through a command-line client — `open`, `press`, `hold`, `click`, `wait`, `look`, `reset`, `state`, `submit` — that drives the same session object on the same server, spends the same budget through the same counters, and returns the frames captured after each action for the model to read. Neither director can see the other's transport; both are talking to one running game.

The two directors reviewed the same bytes. Each brief records the hash of the project tree it was written against, and for *Alley Brawlers* both read `c12fe90b30b6c176`. The human director spent the full 60 actions and no resets; the model spent 25 and no resets.

C.3.3 How the Model Directs

The model director is given one frozen prompt, versioned (`director-prompt/2`) and unchanged within a run; editing it makes a new run. It casts the model as a senior game director reviewing a build whose autonomous development has plateaued, tells it to decide what the next *stage* should be rather than to enumerate defects, and forbids implementation-level instructions.

Two rules do the methodological work. Everything the model knows about the game must come from the session: reading, listing or searching any other file — any source, any log, any score, any other review — discards the review. And the brief is submitted through the same endpoint the form posts to, in the same schema, once; submission ends the session.

C.3.4 What a Brief Is

Four fields, identical for both directors: a one-sentence `stage_objective`; `why_now`, grounded in what the director saw while playing; one to three `priorities`, stated as directions rather than code changes; and zero to three items to `preserve`. The loop consumes the brief in three places

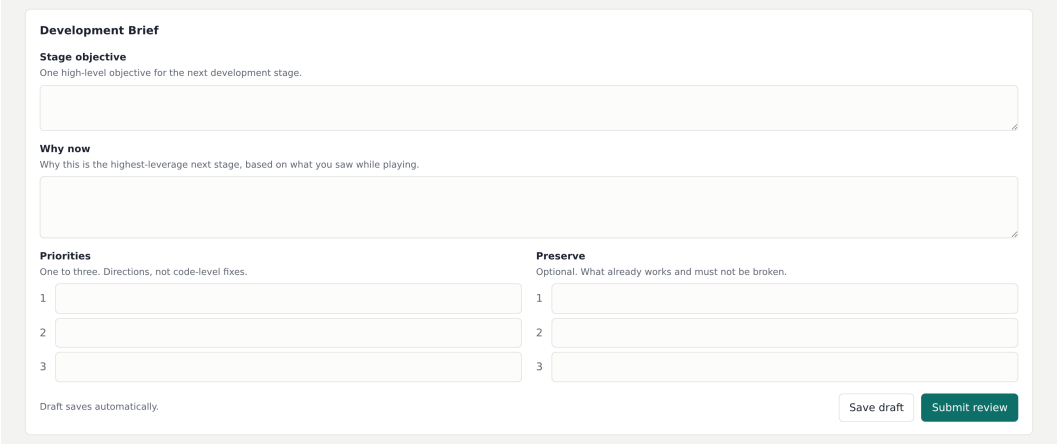

Figure 7: The brief form, the lower half of the page in Figure 6. The model director submits the same four fields as JSON through the endpoint this form posts to.

— the planner, stage ranking, and the improvement and art prompts — and the stage checklist that the Monitor scores against is rebuilt so that the brief's own criteria carry the most weight (Appendix C.2).

Table 4 puts both briefs for *Alley Brawlers* side by side. They were written four minutes apart against the same tree, and they do not overlap. The human read the build as a presentation problem and asked for art and onboarding; the model played two full matches, never took a single point of health off the opponent, and read the build as a mechanics problem. Neither is the correct answer, and the outcome was close: from this checkpoint the human-directed branch reached 62.0, the model-directed branch 61.5, and a control branch restarted from the same build with no brief reached 60.0. What the comparison shows is that a brief supplies a direction the loop's own evidence had not produced in eighteen rounds, and that two directors looking at one build through one instrument can supply entirely different ones.

Human director — 60/60 actions, 0 resets.	
Objective	The art needs work: the title and game-over screens are cluttered and have no proper background, the character art is ugly and should be regenerated in a comic style, and attacks have no effects. The game also needs to teach how it is played — right now I do not know what to press.
Why now	The characters move and there are title and game-over screens, but it is still too rough to be appealing.
Model director (Claude Opus 5) — 25/60 actions, 0 resets.	
Objective	Turn the match from a scripted beatdown into a real two-sided fight: player inputs must visibly and reliably land, block and escape, against an AI that can be beaten, with two fighters that genuinely look and play differently.
Why now	Across two full matches (one as each fighter) the AI never lost a pixel of health. Every round ended in about 18 seconds with me pinned in the left corner in a hit, knockdown, hit loop. Light, heavy, the special input, holding back, crouching and jumping all failed to trade, block or get me out. Nobody can learn to play when nothing they press matters. [...] Seven rounds of visual polish have gone into the frame around the fight (title, background, banners, results screen), but the fight itself is still missing.

Table 4: The two briefs written for the same *Alley Brawlers* build (tree `c12fe90b30b6c176`), abridged to their first two fields. Priorities and preserve items are omitted for space.

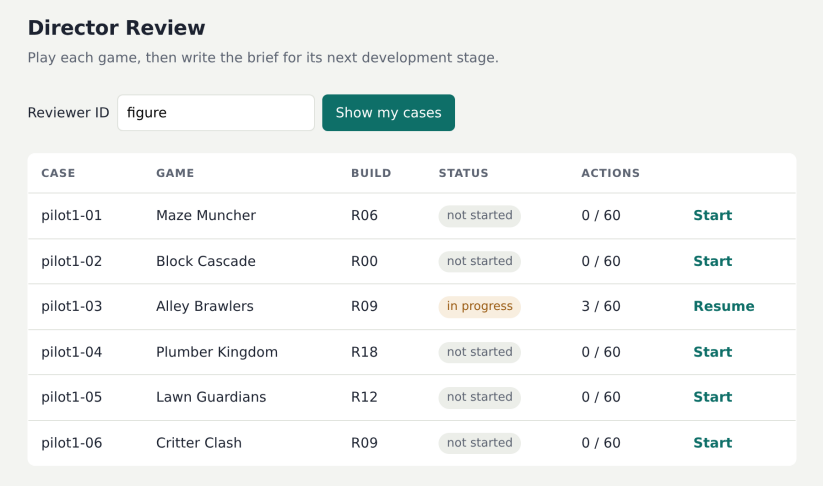

Figure 8: The case list a director opens: six games, the build under review for each, and the budget remaining. Case identifiers are internal labels.

C.3.5 The Case Set

Six games were reviewed, each at the build its own run had converged on, which ranges from the generated project to eighteen rounds in (Figure 8). Every game was reviewed independently by the human director and by the model director, and each brief was then given to a branch restarted from that same build, against a control branch given no brief. Three of the six are the case studies of Appendix F.2.

C.4 Agent Prompts and Output Contracts

This appendix reports the prompts of the five modules that carry the loop's decisions, one per stage of Section 3. Each is abridged to its decision skeleton: fixed catalogues, worked examples, frame-labelling conventions and harness housekeeping are cut, and every cut is marked `[...]`. Braces such as {`items`} are runtime slots filled with the development state, the playtest record, or the replayed frames.

One design choice recurs across all five and is easier to see stated once. Each prompt is told what it needs to decide its own question and deliberately not told the rest. The planner is given the round's direction as a conclusion, not as an argument. The Verifier is told what the round was trying to do, because that is the claim it must check, while the comparator is told neither that anything was improved nor which build is newer. The Editor is told that the problem is already established, so that it starts at "where in the code does this live" rather than re-deriving the diagnosis. Withholding is what keeps a stage from confirming its own expectation.

C.4.1 Controller

The planner decides what to *find out*, never what to change. It is given the round's direction as a settled conclusion, the requirement state, what the build did when last played, the demos available and the budget left; it returns one question and the cheapest way to answer it. Most of the prompt's length goes to ruling out questions that no observation can settle.

Prompt P1. Controller

```
You plan what to investigate next in a game that is being developed one round
at a time.

You do not write code, propose patches, or say what should be changed. Another
agent does that, after the evidence you ask for has been collected. Your output
```

Table 5: The five LLM-facing modules of the RSIGAME loop, by the stage of Section 3 they implement.

Stage	Module	Output contract	Consumed by
Direction Decision	Controller	One `development_question`; `action` in {*reuse*, *explore*, *escalate*}; an `exploration_plan` naming the cheapest mode that can answer it.	Sets the round's objective; *escalate* reports the direction exhausted rather than inventing work.
Agentic Exploration	Explorer	The 1–3 requirements a session should settle, the scenario to boot, and one actionable instruction for the play agent.	Drives the test agent's interaction with P_t, producing the trace τ_t.
Evidence-Grounded Editing	Editor	Edits to the project under a build constraint, plus a report of what was changed.	Produces P_{t+1}.
Agentic Verification	Verifier	`goal_achieved` in {yes, no, unclear}; `broken_now`; `next_goal` when that list is non-empty.	Accepts or rejects the round; anything broken re-enters the state as pending work.
Global Quality Monitoring	Global Quality Monitor	Per-criterion readings for both builds, changes with direction and magnitude, and regressions — never a score.	Aggregated into the proxy value that decides the champion (Appendix C.2).

```
is a decision about what to FIND OUT.

[... direction, requirement state, last playtest, demos, budget ...]

====================
WHAT TO DECIDE
====================
One primary question. It must be:
  - inside the direction above;
  - grounded in the state and evidence above, not in what a game like this
    usually has;
  - answerable by watching the game: someone replays it and looks. Not by
    reading the source, which nobody will do for you;
  - concrete enough that you can say what that observation would be;
  - useful for deciding whether something needs repairing.

Not a question: Can the game be improved? -- nothing observable answers it.
Not a question: Improve the HUD layout. -- that is an instruction to change
something, and nothing has been observed yet.

Then choose one action:
  reuse      the evidence above already settles what the problem is.
  explore    the evidence is missing or ambiguous, and a specific observation
             would settle it.
  escalate   there is no grounded unresolved question left in this direction.
             Say so rather than inventing one.

If exploring, choose the cheapest mode that can answer the question:
  replay_existing_demo    a demo already exercises what you need to see
  extend_existing_demo    a demo reaches the state but stops before the answer
  custom_targeted_probe   no demo can reach it. The expensive option.

If a question has already been asked several rounds running without being
settled, asking it again the same way is not a plan. Either say what would be
different this time, or pick something else.

Reply with strict JSON and nothing else:
{action: reuse|explore|escalate,
 development_question: {question: ..., why_now: ...},
 exploration_plan: {mode: ..., demo_id: ..., focus: ...,
                    stop_when: ...}}
```

C.4.2 EXPLORER

Exploration is aimed, not generic. The probe selector turns the requirements the stage cannot yet answer into one instruction a play agent can carry out, and is asked to weigh importance against what a short session can reach.

Prompt P2. Explorer

```
You are deciding what ONE play session should go and find out about this game
next.

These are the requirements this stage still cannot answer, with where each
could be checked:

[... candidate requirements, with status and scenario ...]

Pick the 1 to {cap} that matter most RIGHT NOW and write a single instruction
for the person who will play the build. Judge importance -- a requirement the
rest of the game depends on beats a detail, and something you can settle in a
few actions beats something that needs a long grind.

If the ones you pick share a place to look, say so; a session that has to boot
two different states spends half its steps travelling.

The instruction has to be ACTIONABLE: name the input to make or the screen to
reach and what to watch. Check whether the plane rotates is weak. In a
sortie, press Left and then Right for about a second each and report whether
the plane's heading changes can be carried out.

Reply with JSON and nothing else:
{items: [T04], scenario: , ask: ..., why: one sentence}
```

C.4.3 EDITOR

The Editor receives a problem that has already been established, and the prompt's first job is to stop it from establishing it again. It is given the source, the build command that must pass, and a way to run and watch the build; it is not given the authority to change the harness or the evaluation traces.

Prompt P3. Editor

```
You are improving a small game. This is round {r} of at most {R}.

You did not play this build. Someone else did: a play agent drove it, the
frames were read, and what follows is what that evidence supports and what was
chosen for this round.

So the problem below has already been established. You do not have to find out
whether it is real, and you should not replay the game to confirm it -- that
work is done, and repeating it spends the calls you need for the repair. Your
job starts at where in the code does this live.

[... the round's evidence packet, recent history, project map ...]

WHAT YOU HAVE
  - the source, in the directory you are working in
  - `{build_cmd}` -- it must pass when you stop
  - you can run the build and watch it: [...]

RULES
  - Change the game, not the harness. Do not edit `node_modules`, `dist`,
    `demo_outputs/`, or anything under `_repair_evidence/`.
  - Do not edit `demo_outputs/`. Those are the input traces the evaluator
    replays; changing them changes the exam, not the game.
  [... process and server hygiene ...]
```

C.4.4 VERIFIER

After the edit, every shipped demo is replayed and one reader decides whether the round earned its commit. The prompt separates two failures that are easy to conflate: a feature the game never had is not a defect, and an input that visibly does *something* has been received even if the script never reaches the outcome it was written for. Only an input that changes nothing counts as broken.

Prompt P4. Verifier

```
A repair agent has just edited a game. Your job is to look at what the game
does now and answer two questions. You are the only check on this edit: after
you, it ships.

WHAT THE ROUND WAS TRYING TO DO
{goal}

[... the requirements this goal came from; the replayed frames ...]

ANSWER THESE TWO, IN THIS ORDER.

1. goal_achieved -- yes / no / unclear.
   Is the thing the round set out to do visibly true now? unclear is a real
   answer: say it when the demos never reach the part of the game the goal is
   about, rather than guessing.

2. broken_now -- a list, possibly empty.
   Anything the game does that is plainly wrong to a player watching these
   frames. Be concrete about what you SAW and in which frame. The kinds that
   matter most, because they are invisible in source code:
     - an input produces no change at all -- the before and after frames are
       the same, again and again, so the game is not listening
     - something covers the playfield: one sprite, one effect, one panel that
       hides the characters, the HUD or the action
     - the screen is frozen: the last frame equals the first
     - text is unreadable, clipped, or drawn on top of itself
   Do not list a missing feature here. A feature the game never had is a task
   requirement, not something that is broken.

AND ONE THING NOT TO REPORT. An input that visibly DOES something -- a mode
label appears, a prompt is shown -- has been received, even when the demo never
goes on to reach the outcome it was written for. That is the game asking for
more steps than this fixed script performs, and it is not a fault in the game.
Only an input that changes NOTHING belongs in broken_now.

Then, only if broken_now is not empty, write next_goal: one sentence telling
the repair agent what to fix first. Fix the worst thing, not all of them.
```

C.4.5 GLOBAL QUALITY MONITOR

The comparator is the only component in the method that reads pixels. It is shown the champion and the candidate replayed under the same fixed script and asked what changed; it is told neither which build is newer nor that an improvement took place. Most of the prompt defends against the two ways a pairwise reading goes wrong: assuming the newer build is better, and mistaking recorder timing for a behavioural change.

Prompt P5. Global Quality Monitor

```
Two builds of the same game were replayed with the SAME fixed input script.
Your job is to say what changed in this scenario, and nothing else.

[... the scenario, the criteria it can speak to, the paired frames ...]

The script is fixed, so both builds received the same inputs at the same
moments. For each moment below you get the OLD build just before and just after
the input, then the NEW build at the same two moments.

The recording is not frame-exact: replaying one build twice cuts the same
moment up to a frame or two apart, which by itself makes a score read one point
different, an animation look half a beat behind, or a falling piece sit one row
lower. That is the recorder, not the build.

You get two frames after each input for exactly this reason. A difference that
is gone by the second frame, or that later moments undo, is timing. A
difference that is still there at the next moment and at the FINAL frame is the
build. Say `same` for the first kind.

RULES
- Compare only what these frames support.
- Do not assume the newer build is better, and do not assume a difference you
```

```
  cannot explain is a defect. Both tilts are errors.
- If a behaviour is not exercised here, its criterion is `unobserved`. Not
  seeing something is not seeing it missing.
- Report anything that got worse separately, however small the rest.
- Judge relative change. Do not score either build.
- Do not propose code edits, repairs, or next steps. That is not your job.
```

D Experimental Protocols and Evaluation Reliability

D.1 Experimental Configuration

This appendix lists, in one place, which model plays which role, how each is served and decoded, and every budget a run is given. Table 6 covers the models, and Table 7 the loop and scoring parameters.

Scoring protocol. Every number in Table 1 is the mean over the 140 tasks of a score produced by the Qwen3.8-27B judge from three independent replay-and-score runs of the same artefact. A task whose generator produced no project scores 0; so does a task whose development has not finished. Base rows score the frozen P_0, Play2Code rows the version thirty rounds deliver, and RSIGame rows the version the Global Quality Monitor is holding when the saturation stop fires.

Table 6: Models by role. The generator differs per row group; every other role is the same model in every row, so that a comparison between rows is a comparison of development methods and not of the models inside them. Decoding is greedy wherever the answer is a claim about the game: a coordinate, a verdict, a rubric item. **Served by** matters for cost and for reproducibility — one model on OpenRouter is sold by a dozen providers, so the judge and the monitor pin theirs.

Role	Model	Served by	Decoding
Generator	one per row group of Table 1	harness default	harness default
Controller	GLM-5.3-Flash	OpenRouter	$T = 0$, top-p 1
Explorer	Qwen3.8-27B	local vLLM (TP $= 2$)	$T = 0$, top-p 1
Verifier	GLM-5.3-Flash	OpenRouter	$T = 0$, top-p 1
Editor	GLM-5.3-Flash	OpenRouter	$T = 0$, top-p 1
Global Quality Monitor	Qwen3.8-Flash	OpenRouter	$T = 0$
Asset generation	Seedream-5.0-lite	OpenRouter (images)	—
Evaluation judge	Qwen3.8-27B	local vLLM	$T = 0$, thinking off, $\leq$2048 output

D.2 Token and Cost Accounting

Scope. We count two phases per task. *Generation* is the generator's single agent run that produces the frozen base P_0. *Development* is every model call made during the development rounds: the controller that determines the direction, the explorer that plays the game, the verifier that inspects it, the editor that edits it, and the Global monitor's periodical check. Scoring by the evaluation judge is excluded for all methods.

Tokens. The **Tok.** column counts billable tokens: input excluding cache reads, plus output. Cache reads are left out of this count because long agent sessions re-read the same prefix on every call; they are still paid, at the cache price, in **Cost**.

Cost. Per call,

$$\text{cost} = n_{\text{in}}\, p_{\text{in}} + n_{\text{cache}}\, p_{\text{cache}} + n_{\text{out}}\, p_{\text{out}},$$

with n_{in} the uncached input tokens and OpenRouter list prices (Table 8). Development is not priced this way but summed from the amount the API returns with each call, which the proxy records for

Table 7: Loop, monitor and scoring parameters. One value per knob, the same in every run reported here.

	Parameter	Value
Round	Tool calls the Editor may make per round	26
	Wall-clock budget per improvement round	1800 s
	Improvement attempts per round before the round is given up	3
	Verification samples per claimed fix	3
	Probe steps taken before an edit is proposed	8
Run	Generated assets per run	20
	Project snapshot kept	every round
	Replay recording kept	every 3rd round
Monitor	Checkpoint interval	every 3 rounds
	Demos replayed per checkpoint	$\leq$3
	Frames per demo the comparison reads	16
	Margin a candidate must beat the champion by	0.05
	Checkpoints with an unchanged champion before the saturation stop (K)	3
Scoring	Replay-and-score runs per artefact	3
	Uniform frames sampled per demo	16
	Rubric items per task (M/D/V/A)	12–21 (median 18)

every call the Controller, Explorer, Verifier, Editor, and Monitor make; the formula above is used where only token counts survive, which is generation. **Cost** covers development only; generation is reported separately in Table 9. Generation cost is dominated by the harness rather than the model: the same GPT-5.5 hits 94% cache under Codex and 42% under OpenGame, and the resulting bills differ by a factor of eight.

Which price. A model on OpenRouter is served by a dozen providers whose prices differ by up to a factor of two, and the model page quotes only one of them, which for several models is the dearest on the board. We therefore price each model at its cheapest *standard* endpoint (Table 8). Two exclusions make that well-defined. Latency tiers are not alternative sellers of the same service and are left out: the flex tier halves GPT-5.5 by deferring the request. And endpoints are compared on what these runs would actually cost there rather than on input price alone, because a quoted cache price of zero means the endpoint offers no prompt caching, not that cache reads are free – read literally it would make a provider without caching the cheapest seller of a workload that is 42–96% cache.

Sources. Generation usage comes from each generation run's own record (input, cache-read and output tokens), released with the base-game corpora; for Codex + GPT-5.5 on Godot only corpus-level means are recorded. Development usage is logged per call by a proxy between the agents and the API.

Table 8: OpenRouter list prices ($ per million tokens), retrieved 2026-09-16. Each model is priced at its cheapest standard endpoint for the cache mix of the runs reported here; the serving provider is named because prices for one model differ by up to a factor of two across providers.

Model	Provider	Input	Cache read	Output
GPT-5.5	OpenAI	5.000	0.500	30.00
Kimi-K2.6	Baidu	0.408	0.069	1.72
Qwen3.8-27B	DeepInfra	0.150	0.037	1.88
GLM-5.3-Flash	DeepInfra	0.075	0.015	0.25

D.3 Replay and Scoring Stability

Replay Variability. GameCraft-Bench scores a game from its observed gameplay rather than its source code alone. Even for the same frozen project, repeated playbacks can produce slightly dif-

Table 9: Generation of the frozen base P_0, mean per task. **Billable** is the **Tok.** of the Base rows; **Total** adds cache reads; **Cost** applies Table 8 to all three token counts. The two GPT-5.5 rows are the same model under two harnesses and differ in price: Codex re-reads a cached prefix, OpenGame does not. Over all 140 tasks.

Engine	Generator	Billable	Total	Cache	Output	Cost
Godot	Codex + GPT-5.5 (high)	0.26M	3.83M	94%	28k	$ 3.77
Godot	GLM-5.3-Flash	0.44M	11.43M	96%	36k	$ 0.20
Godot	Kimi-K2.6	3.22M	8.98M	65%	123k	$ 1.87
Godot	Qwen3.8-27B	6.41M	11.62M	46%	179k	$ 1.47
Godot	Qwen3.8-27B (SFT)	0.57M	8.21M	95%	140k	–
Phaser	OpenGame + GPT-5.5	5.36M	9.21M	42%	97k	$31.17
Phaser	Qwen3.8-27B	9.03M	26.69M	67%	292k	$ 2.51

ferent trajectories because game execution, input scheduling, and frame capture are not perfectly synchronized. The magnitude of this variation depends on the game, and Table 11 measures which kind of game it depends on. Therefore, repeatedly judging the same recording does not capture the full uncertainty of the evaluation process.

Evaluation Protocol. To account for this variability, every artifact is evaluated with three independent end-to-end runs. Each run starts from a fresh replay, generates its own gameplay recording, and is scored independently by the same Qwen3.8-27B judge used for every score in this paper. We report the mean score across the three runs. The judge configuration is held fixed across all evaluations, so the repeated runs primarily capture variation introduced by gameplay replay rather than changes in the evaluation setup.

Observed Stability. Table 10 summarizes the variation across 529 frozen artifacts, every Godot artifact in this paper that was scored three times: the frozen base of two generators, the versions Play2Code delivered, and the versions RSIGAME delivered.

Most artifacts are close to deterministic. The median range across three independent runs is 0.73 points, 225 of the 529 score identically in all three, and a quarter of them vary by less than a hundredth of a point. Improved artifacts vary somewhat more than frozen ones (median 0.87 against 0.36), which is what a build with more of a game in it should do.

Table 10: Score variation across three independent end-to-end evaluations of the same frozen artifact. Each run uses a fresh gameplay replay and fresh judging by the Qwen3.8-27B judge. Range denotes the difference between the highest and lowest Overall scores across the three runs; the columns are its quartiles.

Subset	n	p25	Median	p75	Identical 3/3
All artifacts	529	0.00	0.73	3.33	225
Base P_0	266	0.00	0.36	2.33	131
Improved	263	0.00	0.87	3.50	94

Table 11: Replay variation by task family, for the families with at least twelve scored artifacts, ordered by median range. Same 529 artifacts and same definition of range as Table 10.

Family	n	Median	Family	n	Median
sports	14	4.66	shooter	24	0.44
horror	19	3.37	puzzle	32	0.42
openworld	62	1.41	simulation	24	0.28
visualnovel	32	1.17	strategy	63	0.00
platformer	77	0.87	rhythm	21	0.00
tycoon	58	0.73	idle	16	0.00
roguelike	53	0.62	racing	16	0.00
cardgame	18	0.50			

Which Games Vary. Splitting the 529 artifacts by task family locates the variance rather than leaving it as a property of "game dynamics" (Table 11). The split is not fast against slow: the three most variable families are *sports*, *horror* and *openworld*, while *racing* and *rhythm* — as real-time as anything in the benchmark — have a median range of exactly zero, alongside *strategy*. What the variable families share is that their rubric is answered late. Whether a match was won, whether a haunting resolved, whether a world was traversed is decided by where twenty seconds of play happens to end up; whether a beat landed on time or a lap timer ran is decided by frames the replay reproduces every time. Variance tracks how far into a session the evidence for a criterion arrives, not how fast the game moves.

This also says where a single-run comparison is least safe. A one-point difference is meaningful for a rhythm game and is noise for a sports game, and is one reason the per-family tables of Appendix F.3 should be read with their family's variance in mind.

Cross-Judge Robustness. To test whether the main results depend on a particular evaluation model, we re-score 40 family-stratified Godot tasks with GPT-5.5, using exactly the same replays, sampled frames, task-specific rubrics, and scoring formula as the Qwen3.8-27B judge; only the judge model is changed.

As shown in Table 12, the two judges produce the same method-level ordering, RSIGAME $>$ Play2Code $>$ Base, with similar improvement margins. The gain of RSIGAME over the frozen base is $+14.04$ points under Qwen3.8-27B and $+12.04$ under GPT-5.5, while its gain over Play2Code is $+13.47$ and $+11.76$, respectively; all four bootstrap intervals remain well above zero. The judges also agree on the direction of the RSIGAME-over-base comparison for 37 of 40 tasks. Although their absolute scores are only moderately correlated ($\rho = 0.64$), the conclusions drawn from the relative comparisons remain stable across model families.

Table 12: **Cross-judge robustness on 40 family-stratified Godot tasks.** Qwen3.8-27B and GPT-5.5 score the same replay recordings with the same rubric. Both judges preserve the method ordering and yield similar pairwise improvement margins. Confidence intervals are 95% task-level bootstrap intervals.

	Overall				
Method	Qwen3.8-27B	GPT-5.5			
Base (frozen P_0)	49.07	50.34			
+ Play2Code	49.63	50.62			
+ RSIGAME	**63.10**	**62.38**			
Comparison	Δ Qwen	Δ GPT-5.5	95% CI Qwen	95% CI GPT-5.5	Agreement
RSIGAME − Base	$+14.04$	$+12.04$	$[+9.84, +18.59]$	$[+9.53, +14.60]$	37/40
RSIGAME − Play2Code	$+13.47$	$+11.76$	$[+8.95, +18.19]$	$[+7.79, +16.26]$	34/40
Play2Code − Base	$+0.56$	$+0.28$	$[-3.45, +4.21]$	$[-3.83, +3.83]$	24/40

Agreement counts tasks on which the two judges prefer the same method.

D.4 FREE-PLAY EVALUATION

Protocol. GameCraft-Bench evaluates a game by replaying the demonstration traces packaged with its submitted project. These traces are visible development resources, rather than hidden benchmark trajectories, and RSIGAME reuses them for controlled before–after comparisons across checkpoints. Although the hidden rubric, scores, and judge feedback are never exposed during development, repeated use of the same demonstrations could still favor improvements specific to those interaction trajectories. We therefore evaluate whether the gains persist under independently generated gameplay.

We sample 20 family-stratified Godot tasks. For each task, the frozen base, Play2Code, and RSIGAME builds are independently played by blind agents with 40 actions and one reset. The agents see only the public game specification and the running game—not the submission demonstrations, rubric, score, source code, development history, or method identity—and return factual reports of their observations. A separate judge compares these reports pairwise, yielding 60 comparisons in total.

Evaluation. The pairwise judge follows the same four quality dimensions and relative weighting used in the main evaluation, while judging only evidence observed during free interaction. Importantly, the replay scripts are never shown to the play agents: they choose their own actions, timing, and trajectories. This changes the interaction distribution while keeping the definition of game quality aligned with the main evaluation.

Results. As shown in Table 13, RSIGAME is preferred over the frozen base on 17 of 20 tasks and over Play2Code on 18 of 20, while Play2Code does not show a clear advantage over the base. Across all decided comparisons, the free-play evaluation agrees with the benchmark ordering in 45 of 59 cases. These results indicate that the gains of RSIGAME persist under independently chosen gameplay trajectories rather than being confined to the fixed replay scripts used during development and benchmark evaluation.

Table 13: **Free-play evaluation on 20 family-stratified Godot tasks.** Blind agents independently play each build, and a separate judge compares their reports using the same quality dimensions and weighting as the main evaluation. p is a two-sided sign test over decided comparisons.

Comparison	W/L/T	p	Agreement
RSIGAME – Base	**17/3/0**	**0.003**	
RSIGAME – Play2Code	**18/2/0**	< 0.001	45/59
Play2Code – Base	9/11/0	0.824	

W/L/T denotes wins/losses/ties for the first method. Agreement counts decided comparisons for which free-play evaluation and the benchmark score prefer the same build.

D.5 STATISTICAL RELIABILITY OF MAIN RESULTS

Paired task-level uncertainty. All main-table comparisons are paired by task. We estimate uncertainty in the mean Overall difference Δ using $20{,}000$ bootstrap resamples of the 140 benchmark tasks, and additionally report a Wilcoxon signed-rank test over the paired task-level differences. We separately resample the three replay-and-score runs of each fixed task to measure evaluation noise; these intervals are substantially narrower than the task-bootstrap intervals, indicating that benchmark-task variation dominates replay and judge noise.

Main comparisons. RSIGAME significantly improves over its frozen starting point in every engine–generator setting inlcuded in Table 14, with gains of $+8.8$ to $+14.3$ Overall points and task-bootstrap intervals well above zero. Against Play2Code, the improvement is significant for both Godot generators and for GPT-5.5 on Phaser; the Qwen3.8-27B Phaser comparison remains unresolved ($+1.55$, $p = 0.394$).

Table 14: **Paired significance of the main results.** Δ is the mean task-level Overall difference. Confidence intervals are obtained from $20{,}000$ paired bootstrap resamples of benchmark tasks; p is from a Wilcoxon signed-rank test. W/L counts tasks on which the first method scores higher/lower.

Comparison	Δ	95% CI	p	W/L
Godot				
GPT-5.5: RSIGAME – Base	+14.26	[11.49, 17.06]	$< 10^{-4}$	112/16
GPT-5.5: RSIGAME – Play2Code	+13.79	[10.89, 16.65]	$< 10^{-4}$	118/19
Qwen3.8-27B: RSIGAME – Base	+10.70	[7.75, 13.77]	$< 10^{-4}$	100/31
Qwen3.8-27B: RSIGAME – Play2Code	+7.24	[4.49, 10.04]	$< 10^{-4}$	92/41
Phaser				
GPT-5.5: RSIGAME – Base	+8.77	[6.13, 11.51]	$< 10^{-4}$	101/35
GPT-5.5: RSIGAME – Play2Code	+3.11	[0.53, 5.61]	0.008	83/53
Qwen3.8-27B: RSIGAME – Base	+10.21	[6.99, 13.70]	$< 10^{-4}$	95/36
Qwen3.8-27B: RSIGAME – Play2Code	+1.55	[-1.50, 4.66]	0.394	73/59

D.6 Adaptive Direction Policy Evaluation

Experimental Setup. We compare adaptive development with a round-robin direction policy on 16 paired games initialized from the same frozen base projects. Each policy is run for 30 development rounds with the same agent models, tools, and per-round budget; the only difference is how the development focus is selected. Figure 5a,b analyzes the resulting trajectories using signals that are not exposed to either policy during development. We define a *broken build* as a round whose resulting project fails to produce a runnable game; the following round is then aligned at offset $+1$ relative to this failure event. For state-conditioned analysis, we additionally use a held-out diagnostic rubric to identify the currently lagging quality dimension. This rubric is used only for post-hoc analysis and is never provided to the development loop. Shaded areas in Figure 5a are ± 1 standard error over the corresponding break events or grouped development rounds.

Blind Pairwise Evaluation. To compare the final games produced by the two policies, we perform 63 blind pairwise comparisons using their round-30 checkpoints. For each comparison, the same demo trace is replayed on both builds and the resulting observations are presented side by side to the evaluator. The left–right assignment is randomized independently for each comparison, and no information about the underlying direction policy is revealed. Evaluators report an overall preference as well as preferences along the Mechanics, Depth, Visuals, and Art dimensions; ties are allowed. To check for position bias, a subset of comparisons is evaluated again with the two sides swapped. These repeated judgments are used only as a consistency check and are excluded from the reported 63 comparisons. The pairwise evaluator is independent of the GameCraft-Bench benchmark evaluator and has no access to its scores, rubrics, or feedback.

D.6.1 Pairwise Judging Criteria

The criteria given to the blind pairwise judge of Section 5, reproduced in full. They are deliberately not the benchmark's rubric: the judge compares two builds on what its frames show, while the benchmark scores one build against the task. Rule 1 is the one that does the most work — without it a broken build collects credit for the frames it does manage to render.

Comparison Rubrics. Blind pairwise judge

```
You are an expert game reviewer. You compare two builds of the same game, Game
1 and Game 2. Both were developed from the same starting game and the same
design document, which is given below. Each build was played with exactly the
same scripted input (a demo), and you see frames sampled at the same fixed
interval from that recording, in time order. Judge only what is visible in the
frames.

For each dimension below, decide which build is better: 1, 2, or tie.

## Core Mechanics
```

```
Does the scripted input visibly produce the gameplay the design document
describes?
- Better: actions have clear on-screen effects (movement, attacks, placement,
  state changes, score or resource changes); the core loop of the design
  document can be observed.
- Worse: input has no visible effect; the game stays on a title or menu screen;
  a scene fails to load; the screen is blank or frozen.

## Content Depth
How much of the designed game is present and reached in this demo?
- Better: more of the distinct mechanics, enemies, levels, events, progression,
  or win/lose states from the design document appear.
- Worse: a single repeated screen; placeholder content; systems that are
  announced in the UI but never shown.

## Functional Visuals
Can a player read the game state?
- Better: HUD, feedback, and important objects are legible and clearly
  separated from the background; text is not clipped or overlapping; sprites
  and animations are stable from frame to frame.
- Worse: unreadable or overlapping text; flickering or inconsistent sprites;
  key objects hidden behind others; no visible feedback for important events.

## Presentation & Art
Does it look like a coherent, finished game?
- Better: a consistent art style across screens; intentional composition;
  assets that match the theme of the design document.
- Worse: primitive shapes where art is expected; mixed or clashing styles;
  visual noise; decoration that makes the game harder to read.

## Overall
Which build would a player of this design document prefer, all things
considered?

## Rules
1. A build that is broken in these frames (blank, crashed, frozen, stuck on a
   screen the demo should have left) loses every dimension it fails to show,
   however good its other frames look.
2. More assets, more effects, or busier frames are not better by themselves.
   Prefer them only when they serve the design and keep the game readable.
3. Ignore which side a build is shown on. Game 1 and Game 2 are in random
   order.
4. Answer tie when the two builds are not visibly different on a dimension,
   or when both fail it equally.
5. Each reason must point to something visible, naming the build and the frame
   (for example Game 2, frame 7: the score counter overlaps the timer).
```

D.7 AGENTIC VERIFICATION AUDIT

Audit Protocol. We evaluate the reliability of verification on development trajectories from 40 games initialized by GPT-5.5. We sample pre- and post-improvement decisions from these trajectories and provide the complete interaction evidence associated with each decision to Claude Opus 5 (Anthropic, 2026) for independent adjudication. Each case is evaluated twice; disagreements between the two judgments are manually reviewed to obtain the final label. This evaluator is used only for the verification audit and is fully isolated from the GameCraft-Bench evaluator and its scores, rubrics, and feedback.

Pre-Improvement Grounding. We evaluate whether proposed improvement targets are supported by the evidence available before editing. For each audited session, a target is labeled as *grounded* when the observed gameplay evidence supports the claimed problem or improvement opportunity, and *ungrounded* when it is unsupported or contradicted by the evidence. We report *grounded precision*, the fraction of decidable targets that are grounded, and the average number of *ungrounded targets per round*. We compare the agentic evidence-grounding procedure used by RSIGAME against a free-form critic operating on the same development context.

Post-Improvement Failure Detection. We further evaluate whether verification can detect unsuccessful edits after the updated game is executed. An improvement is labeled unsuccessful when the intended change is not observable in gameplay or when the edit introduces a regression. Treating unsuccessful improvements as the positive class, we report failure recall, specificity, and balanced

accuracy over 48 adjudicated rounds. We compare replay-based agentic verification with a build-only baseline that checks whether the updated project executes successfully but does not inspect its interactive behavior.

D.8 Multi-Stage Guidance Evaluation

Section 5 tests one stage of development. This appendix specifies how we ask whether a second and a third stage keep paying: each pair of stage champions is compared by blind pairwise play, so a later stage has to be preferred as a game rather than merely score higher.

What is compared. Three pairs of builds, each the champion the Global Quality Monitor delivered at the end of a stage: stage 2 against stage 1, stage 3 against stage 2, and stage 3 against stage 1. Both builds of a pair come from the same game, so each comparison is within-task, with the same generator, the same task specification and the same per-stage budget on both sides. The result shows that later stages are consistently preferred over earlier ones, indicating that high-level guidance can repeatedly reopen development headroom after saturation and support continued multi-stage evolution.

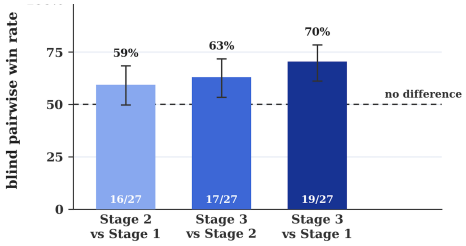

Figure 9: Blind pairwise win rate of the later stage in each pair, with ± 1 s.e.

Why pairwise. A stage that repairs a broken mechanic and a stage that adds an unused menu can move the benchmark score by the same amount. This evaluation therefore asks which build a player would rather have, and do not use the benchmark score as the outcome.

Three agents, one build each. A comparison uses three agents. Two *play agents* each receive one build and never see the other, nor any score, source file, round number or development history; each returns a fixed-format factual report — whether the game starts, whether the core loop can be completed, how inputs respond, what rendered, and what is missing or broken — and is explicitly forbidden to give a verdict. A third *judge agent* reads only the two reports, in an order randomised per comparison and recorded, and returns *report 1 better*, *report 2 better* or *tie* with a confidence level. The judge is instructed to weigh a concretely named defect above a general impression, and to treat anything about the review harness — how much game time a call advances, how long a build takes to boot, the action budget — as shared by both builds and therefore not evidence about either.

Equal budgets. Each play agent drives its build through the review tool of Appendix C.3 under a fixed budget of 40 actions and one reset, with `wait`, `look` and `state` free. Sessions are keyed by reviewer id and a reopened session resumes with the budget it has already spent, so each agent is given an id unique to its comparison and checks its own action counter immediately after opening; a session that reports a non-zero count is discarded and that comparison re-run. Two builds compared under unequal budgets yield a verdict that follows the budget rather than the build, which is why this check is part of the protocol rather than an afterthought.

Reporting. A comparison counts as a win for the later stage when the judge prefers the report belonging to it. Ties are reported separately and excluded from the win rate rather than split, since a tie states that the two reports do not separate the builds. Win rates carry ± 1 standard error in Wilson form, which keeps the interval inside $[0, 1]$ near the ends; at a few dozen games per pair these intervals are wide, and the text says so rather than reading a ranking out of overlapping bars.

E Training Data and Experience Internalization

E.1 Training Corpus Construction and Curation

The released corpus has two parts. **Base games** are the generated projects themselves, with the run that produced each one: 1,998 de-duplicated games (1,446 Godot, 552 Phaser), 1,070 of them

paired with the generation trajectory, plus 910 complete per-trial run trees over three generation batches. **Supervision** is what we train on, extracted from those runs in three stages and summarised in Table 15: S1 game generation, S2 planning, and S3 improvement. Generation trajectories are produced by GPT-5.5 and improvement trajectories by GLM-5.3-Flash. Plans are not free-written: 1,237 are distilled from the file tree of a finished game, and 870 come from an earlier plan set and are kept only where every path they name exists in the matching game.

Table 15: The released training corpus, counted from the published tables. A *decision* row is one assistant turn together with the history that preceded it, so decision rows repeat context and their token counts are not independent; the trajectory rows are the same recordings un-windowed and are what to count tokens over. *Verified* improvement rounds are those an independent post-improvement verification judged to have achieved the stated goal; the remainder are released as negatives rather than dropped.

Stage	Unit	Teacher	Godot	Phaser	Total
S1 generation	trajectory	GPT-5.5	1,929	284	2,213
S1 generation	decision row	GPT-5.5	66,312	28,097	94,409
S2 planning	plan	distilled	1,979	129	2,108
S3 improvement	round	GLM-5.3-Flash	3,730	273	4,003
of which verified	round	—	1,869	144	2,013
S3 improvement agent	decision row	GLM-5.3-Flash	111,302	9,930	121,232
of which verified	decision row	—	48,341	5,098	53,439
Games improved	game	—	345	43	388
Distinct tasks	task	—	2,189	322	2,511
Training rows	row	—	183,063	38,689	221,752

Trajectory Serialization. Each generation session is serialized as a single multi-turn agent trajectory containing the task specification, assistant outputs, tool calls, and tool results. Training loss is applied only to assistant-generated outputs, avoiding repeated supervision over the shared interaction context. Planning examples are extracted from the agent's initial implementation plan before coding, while later progress updates to the same plan are discarded. A decision row keeps a pinned prefix (system prompt, task, and the plan where one exists), then as many recent whole decisions as fit, then the target decision last, within a 32k-token budget.

What is filtered out. A generation trajectory is retained only when the session produced an executable artifact; artifact existence is used rather than the agent SDK's reported success status, which does not reliably indicate whether a runnable project was produced. Empty trajectories and duplicated tasks are removed. A plan is kept only if every file path it references exists in the finished game and its engine matches the brief. An improvement round is dropped when its session failed or its diff is empty, and is labelled positive only on an independent post-improvement verification — 1,869 of 3,730 Godot rounds and 144 of 273 Phaser attempts. Verbose tool outputs such as build logs are middle-truncated, preserving the executed command and its verdict. Whole-project diffs are compacted to the source hunks the improvement actually changed, which shrinks an improvement example by roughly $48\times$ at the median.

Evaluation contamination. Nothing from a baseline, ablation or evaluation run enters the corpus, and the 140 distinct benchmark task ids are excluded at build time. The published tables were rescanned against those ids after the fact: zero overlaps in all 13 configurations, over a union of 2,511 task ids.

Recording gaps are flagged, not hidden. One agent SDK stored a one-line UI summary of a tool result instead of the content the model actually received. The affected rows are kept and carry explicit flags (`row_has_summary_only`, `target_follows_summary_only`, `tool_results_source`, `trajectory_incomplete`), so a clean training subset is a filter rather than a different release. This affects all 284 Phaser generation trajectories (20.2% of their rows have the target decision immediately after a summary) and one of the two Godot improvement arms (31.7% of multi-turn improvement rows); the other Godot arm is complete.

Anonymization. Every row, metadata column and document was rewritten member by member. Absolute machine paths became `/workspace/game`, `/workspace/harness` or `<path>` (403,280 rewrites); user, host, account and internal project names became `<redacted>` (13,610); model-gateway hosts, internal mirrors and cluster names were replaced; e-mail addresses and non-loopback IP literals were removed; owner columns in captured directory listings were rewritten; and 26,577 configuration fields were neutralised in place, so the schema is unchanged. Tool outputs that dumped the build host's process table were *deleted* rather than scrubbed (26 listings, 379 lines). Scientific content — game code and assets, rubric text, per-requirement scores, model identifiers, reasoning effort, timings — is untouched. The cleaner was verified by a second, independently written scanner carrying 47 identity and credential patterns plus a planted-leak self-test that requires every planted leak to be removed; both report zero hits.

E.2 Fine-Tuning Configuration

We fine-tune Qwen3.8-27B (Qwen Team, 2026) using parameter-efficient supervised fine-tuning with LoRA (Hu et al., 2022) over 4-bit quantized base weights (Dettmers et al., 2023). Each agentic trajectory is treated as one long training document, allowing the model to learn planning, coding, tool-use, and improvement decisions within their original development context. Unless otherwise specified, all training configurations use the same hyperparameters; only the composition of the training corpus changes.

Training Mixture. The three variants are not three separate mixtures but three nested corpora: each adds one stage of supervision to the one before it, so S1 $\subset$ S2 $\subset$ S3 and any difference between two rows of Table 18 is attributable to the stage that was added. Both engines are trained together; Table 16 gives the composition.

Table 16: The training corpus of each variant in Table 18. Each stage adds rows to the previous corpus; the cumulative column is what that variant was trained on. Tasks in the held-out brief list are filtered from every stage.

Stage added	Unit	Godot	Phaser	Rows added	Cumulative
S1 generation	agent trajectory	876	418	1,294	1,294
S2 planning	brief $\rightarrow$ file plan	216	92	308	1,602
S3 improvement	improvement round	509	0	509	2,111
Total		1,601	510	2,111	

Long-Context Training. Agentic development trajectories can span tens of thousands of tokens. We therefore use a maximum context length of 57,344 tokens and distribute each sequence across four GPUs using Ulysses-style sequence parallelism (Jacobs et al., 2023). Gradient checkpointing and a fused cross-entropy implementation are used to reduce activation and output-head memory consumption.

Training Objective. For each trajectory, the original interaction context is preserved, while the supervised loss is applied only to assistant-generated outputs. This trains the model on the sequence of development decisions without treating tool observations or environment feedback as prediction targets. The full RSIGame model is trained on the union of planning, generation, and independently verified improvement examples.

E.3 Training-Data Ablation

Table 18 gives every variant compared in Section 5 (Figure 4c draws +Gen and the full model). Base and the full model are the Qwen3.8-27B and Qwen3.8-27B (SFT) rows of Table 1.

Table 17: Fine-tuning configuration used for experience internalization. These are the settings of the released adapter; every variant in Table 18 is trained with them and differs only in the corpus of Table 16.

Configuration	Value
Base model	Qwen3.8-27B
Adaptation	LoRA (Hu et al., 2022)
LoRA rank r	16
LoRA α	32
LoRA dropout	0.05
LoRA targets	All linear projections
Rank-stabilized LoRA (rsLoRA)	Yes
Base-weight precision	4-bit NF4 (Dettmers et al., 2023)
Double quantization	Yes
Compute precision	bfloat16
Maximum context length	57,344 tokens
Epochs	1
Optimizer	AdamW (Loshchilov & Hutter, 2019), fused
Learning rate	5×10^{-5}
Learning-rate schedule	Cosine
Warmup ratio	0.03
Weight decay	0.01
Per-device batch size	1
Gradient accumulation	None
Sequence-parallel degree	4
Hardware	$4\times$ H100 80 GB

Table 18: One-shot Godot generation by training variant. Every row is over the 140 tasks.

Variant	Mechanics	Depth	Visuals	Art	Overall ↑
BASE	41.2	33.4	38.6	38.3	37.07
+GEN	49.0	41.4	42.0	38.0	41.45
+GEN+PLAN	48.5	40.8	42.6	39.5	41.77
+GEN+PLAN+IMPROVE	**56.1**	**47.5**	**49.8**	**44.8**	**48.22**

F EXTENDED EVALUATION AND CASE STUDIES

F.1 COMPARISON WITH A MULTI-AGENT GENERATE-AND-VERIFY SYSTEM

Why this comparison. This paper does not claim that an agent can improve a game it generated — several systems do that. It claims that doing so *reliably* needs a controller: something that decides where the headroom is, keeps the best version, and stops. The closest public system we could run is VibeGame[1], an eight-role agent team whose pipeline already contains verification and optimisation roles. If a team of that shape closed the gap on its own, a controller would be an ornament.

Setup. Forty Phaser tasks of GameCraft-Bench, the VibeGame code unmodified, all eight of its roles set to GPT-5.5, one trial per task, fully automatic, no human intervention. The builds are scored with the same verifier, rubrics and judge as every other Phaser arm in this paper.

What it shows. An agent team that already verifies and optimises scores **30.5**, against **46.0** for a single generation pass on the same tasks: its loop does not convert rounds into quality. On the twelve tasks we share it reaches 23.3, where the frozen P_0 of our Phaser arm is already at 50.1 and RSIGAME brings those same twelve to 62.4. The presence of verification and optimisation *roles* in a pipeline is not what produces sustained improvement.

Reading it fairly. One trial per task, and a system we configured only to the extent of setting its role models; we make no claim about VibeGame under other models or a larger budget.

[1] `https://github.com/tettethu/VibeGame`, Apache-2.0.

Table 19: VibeGame against one-shot generation and against RSIGAME. Left: the twelve tasks shared by VibeGame's forty and our forty-task Phaser development subset, so every column is the same twelve games. Right: VibeGame's own forty tasks, against the one-shot reference its release quotes on them.

	the 12 shared tasks	VibeGame's own 40
VibeGame	23.3	30.5
OpenGame + GPT-5.5, one shot	50.1	46.0
+ Play2Code	56.4	—
+ RSIGAME	62.4	—

F.2 QUALITATIVE EVOLUTION CASE STUDIES

We follow three games from the build a code model generates to the build we ship. Each case opens with a timeline: the official score at every third round of the 30-round autonomous run, the version the Global Quality Monitor holds, the round at which the saturation stop ends the run, and, past an axis break, the passes the loop did not run — the stage briefs written once the champion stopped being beaten, and the development passes after them. Each build on that timeline then gets a card of its own: what that round or pass found, what it changed, and the frames it is judged on, with boxes placed by eye over the screenshots, green where something arrived and red where it is still wrong. Every build of a game is driven through the same script, so its panels are comparable.

F.2.1 HOLDING THE BEST VERSION THROUGH REGRESSIONS

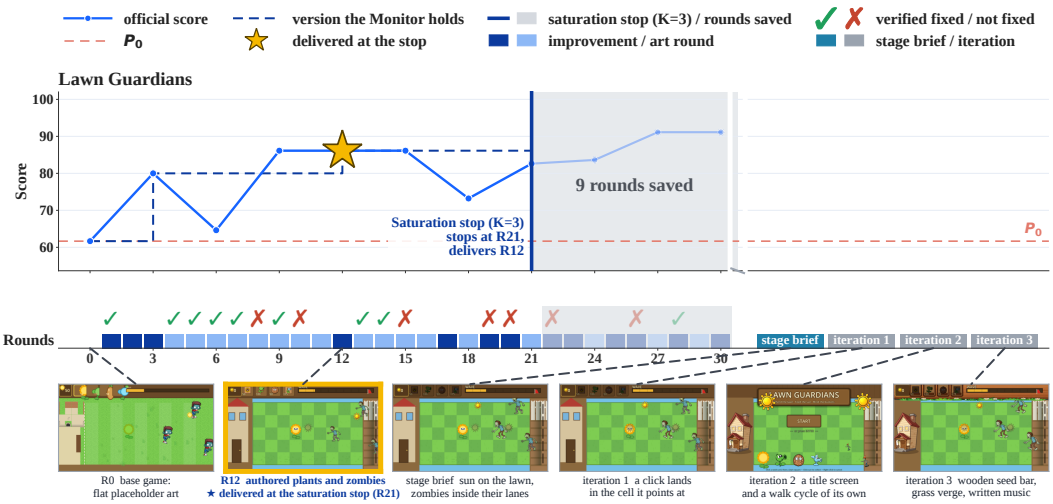

Figure 10: *Lawn Guardians*, from the generated game to the build we shipped. The score band gives the official score at every third round of the 30-round autonomous run, the version the Global Quality Monitor holds, and the star where the saturation stop delivers it at round 21, nine rounds early. The round band gives one cell per round — improvement or art — with the post-improvement verdict above it. Past the axis break are the passes the loop did not run: the stage brief written once the champion had survived three checkpoints, and the three iterations after it, which are development passes rather than loop rounds and so carry no round numbers. Each build below is opened up as a card in Figures 11–F.2.1.

Development does not end where the run does. The Global Quality Monitor's stop criterion is about the run's own scores, not about the game: once the champion survives three checkpoints, nothing in the loop's evidence points anywhere, and a stage brief written from outside supplies the direction the loop no longer has. Three further passes follow it — controls, animation, and set dressing with

a written soundtrack — each changing what the game is like to play rather than what it does. The six builds are shown one card each in Figures 11–F.2.1.

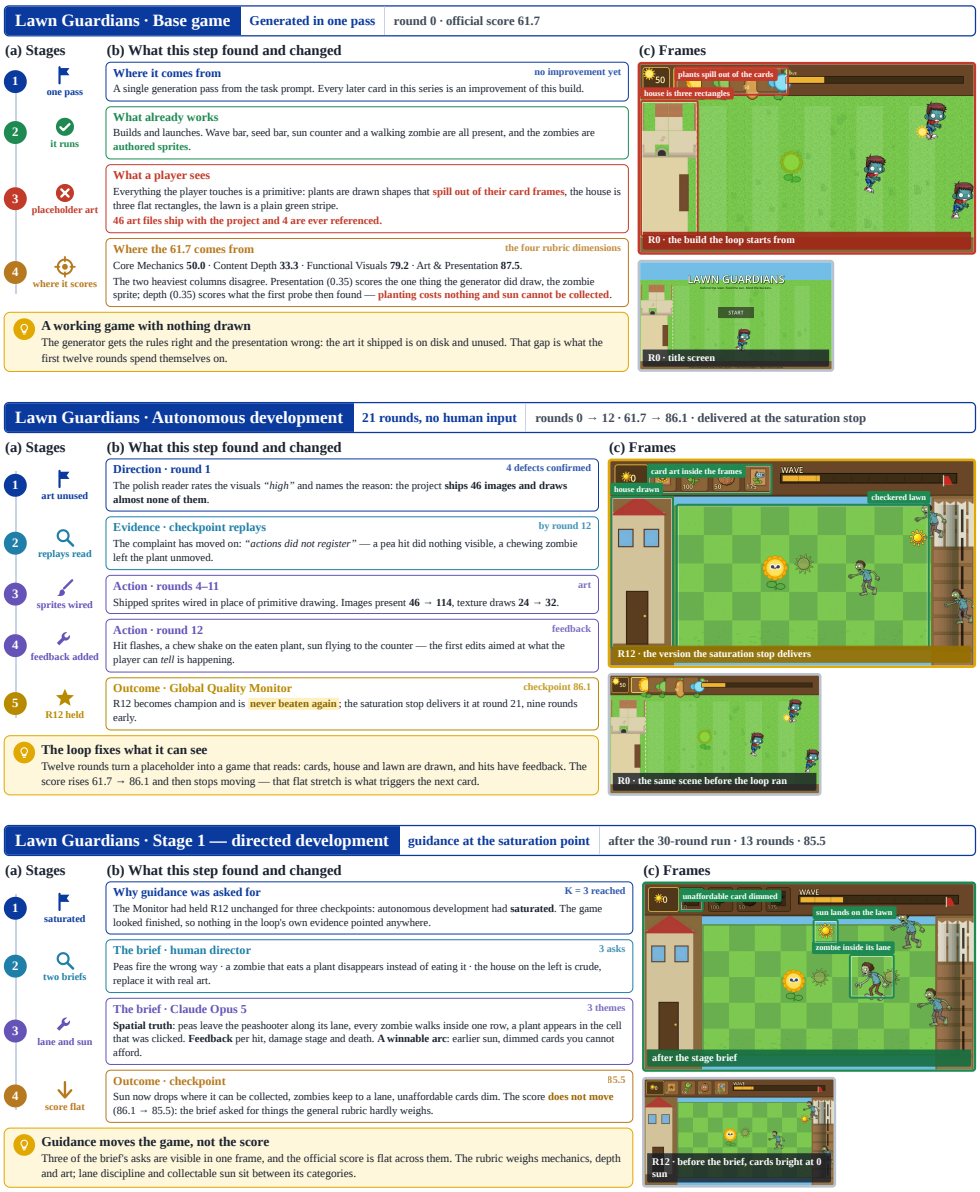

Figure 11: *Lawn Guardians* from the generated build to the stage brief. Each card carries what that round or pass found, what it changed, and the frames it is judged on; boxes over the screenshots are placed by eye, green where something arrived and red where it is still wrong, and every build is driven through the same script so the scenes are comparable. **Base game:** the generator has the rules right and the presentation wrong — the art it shipped is on disk and unused. **Autonomous:** twelve rounds wire that art in and add hit feedback, and the champion that emerges is never beaten. **Stage 1:** with the champion held for three checkpoints the loop has run out of its own evidence, and a brief written from outside — by Claude Opus 5, alongside the human's — puts sun on the lawn and zombies inside their lanes.

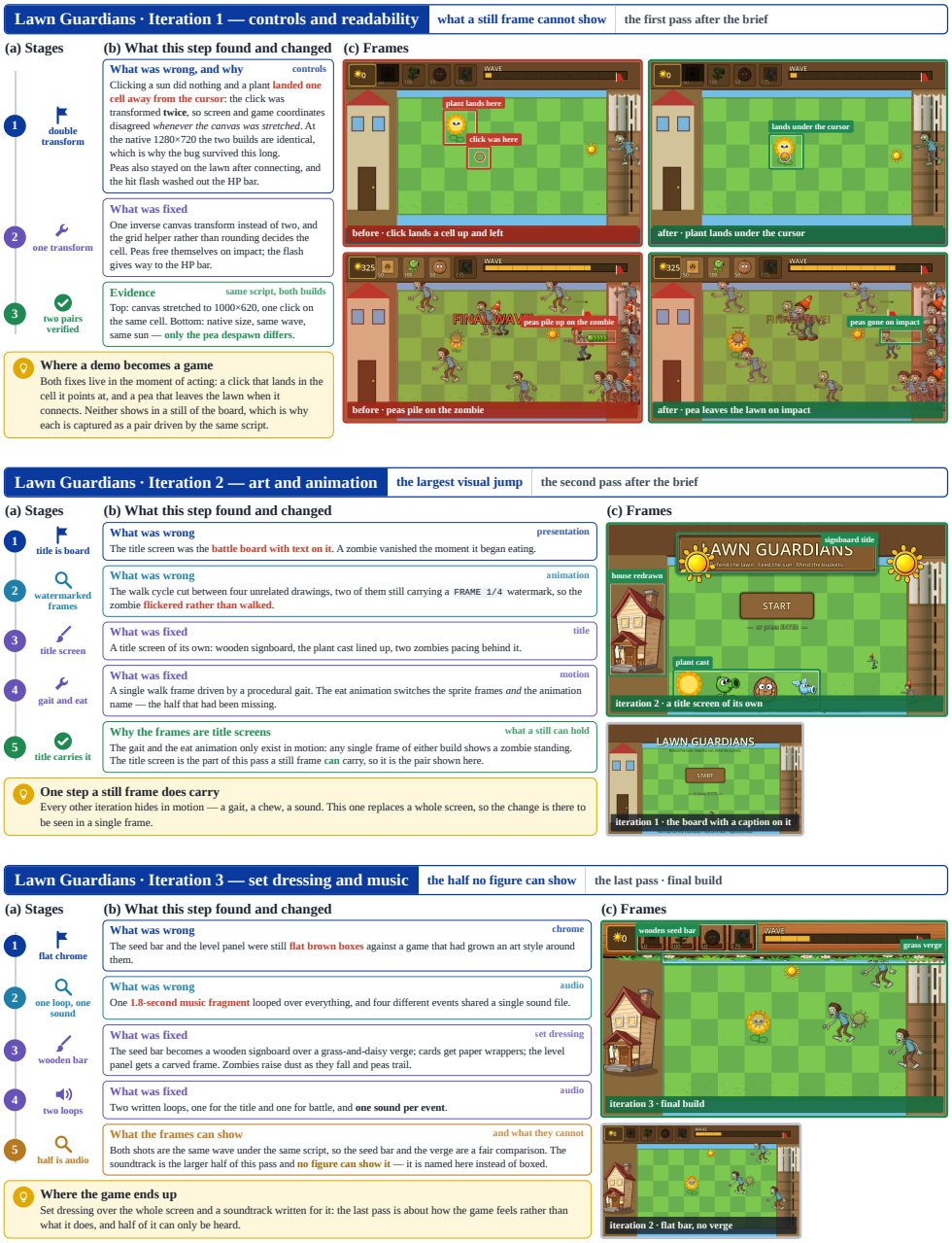

Figure 12: The three development passes that follow the stage brief. **Iteration 1** is a fix no still of the board can show, captured instead as two before/after pairs under the same script: at a stretched canvas the click was transformed twice, so a plant landed a cell from the cursor, and peas stayed on the lawn after connecting. **Iteration 2** replaces the title screen and the walk cycle, the one step a still frame does carry. **Iteration 3** is set dressing and a rewritten soundtrack; the seed bar and verge are in the frames, and the audio is the half no figure can show.

In *Lawn Guardians* (Figure 10), progress is not monotone: an art round (R6) and a feedback round (R18) each lower the score, and the Global Quality Monitor adopts neither. It holds R12 (86.1, up from 61.7) until the saturation stop delivers it at round 21. Two limits also show: a verification replay can be too short to see a real fix (R26), and the official judge can reward a build whose art covers the play field (R27). Play2Code peaks at 81.5 and ends at 65.7.

F.2.2 TWO BRIEFS ON THE SAME CHECKPOINT

Alley Brawlers is the case where the loop's own evidence buys the scenery and loses the actors. Thirty autonomous rounds draw a lit street behind the fight — the largest single change in the run — and in the same rounds the fighters stop resolving: in the delivered build a body in motion is a torn smear. The two cancel, and the champion the saturation stop delivers scores exactly what the base game scored.

What the loop could not produce is the observation that nobody can win. Both stage-1 directors played the build and said so, and the run that followed the model's brief moved the score for the first time. A control branch, restarted from the same checkpoint with no brief at all, reached 60.0; the briefs reached 61.5 and 62.0. A second brief, written at the stage-1 champion, then went after what still made the match unplayable — two bodies standing in the same space — and bought a readable exchange rather than a higher score.

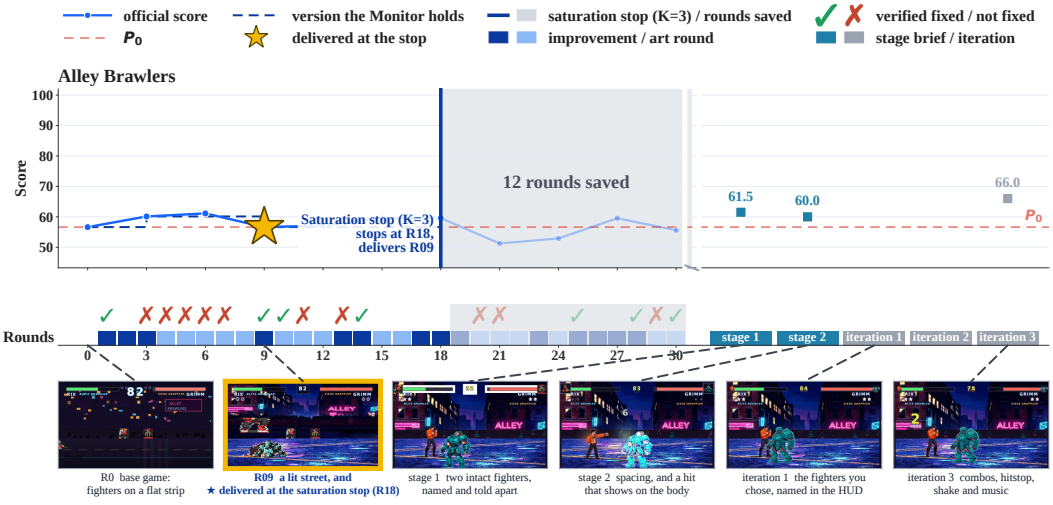

Figure 13: *Alley Brawlers*, from the generated game to the build we shipped. Score band, round band and frames as in Figure 10. The saturation stop ends the autonomous run at round 18 and delivers round 9. Past the axis break are the passes the loop did not run: two stage briefs, each written by a director who played the build in front of it, and three development passes. Stage 2 and the iterations both continue from the stage-1 champion, so they are alternatives rather than a sequence.

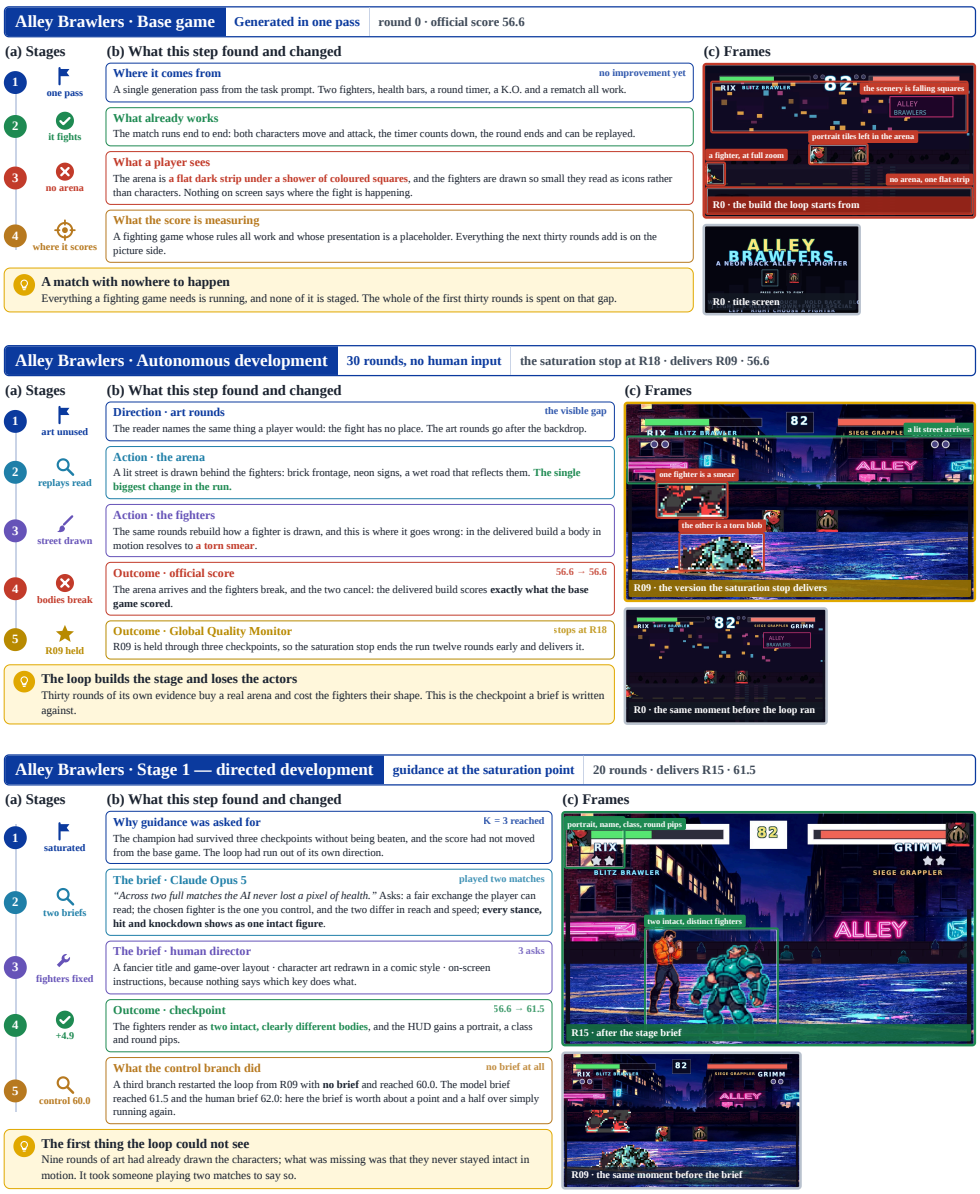

Figure 14: *Alley Brawlers* from the generated build to the first brief. Cards, boxes and script as in Figure 11. **Base game:** a working match on a flat strip under falling squares. **Autonomous:** the arena arrives and the fighters break, and the delivered build scores exactly what the base game scored. **Stage 1:** a director who played two matches asks for a fair exchange and for a fighter that stays one intact figure; the score moves for the first time in the run.

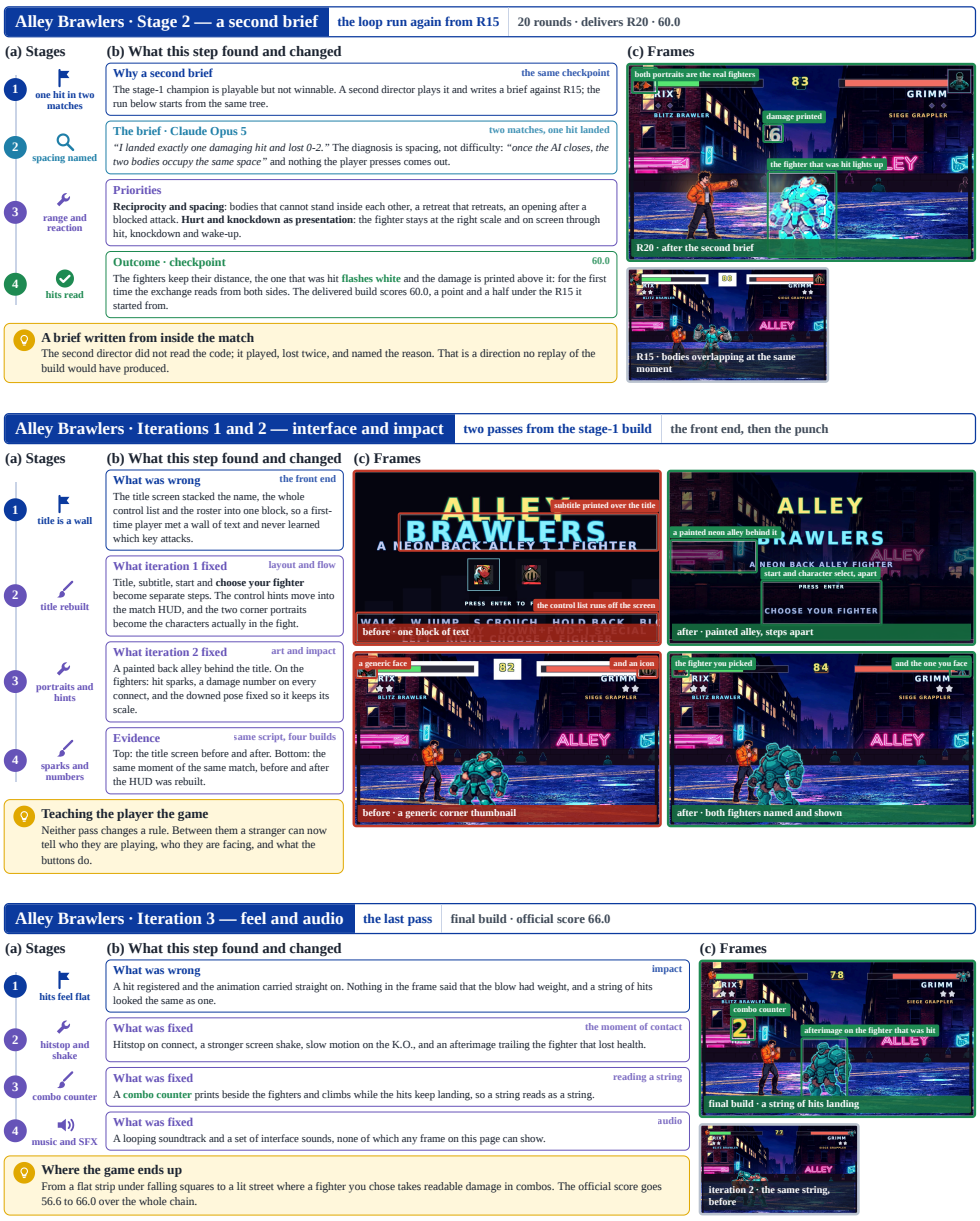

Figure 15: The second brief and the development passes, both continuing from the stage-1 champion. **Stage 2:** a second director names spacing as the reason a player cannot act, and the exchange becomes legible from both sides. **Iterations 1–2** rebuild the front end — the title becomes steps rather than a wall of text, and the HUD names and shows both fighters — and add hit sparks and damage numbers. **Iteration 3** is the feel of contact: hitstop, screen shake, K.O. slow motion, a combo counter and a soundtrack.

F.2.3 A Game the Loop Cannot Improve

Block Cascade is the opposite starting point. Seven-bag randomiser, next-three preview, ghost piece, hold swap, wall kicks, line clears, top-out and a best score all work in the generated build, and thirty rounds of evidence-led improvement find nothing worth swapping to. The loop is not stuck on a defect; it has run out of anything to call one.

Restarting from that build is what pays, and the briefs are not what makes it pay: the two brief-guided branches and the control branch with no brief all reach 50.0, and what the briefs buy is arriving there at round 6 rather than round 12. The second brief then asks for stakes — a named mode, a live clock, a reward ladder — and gets them, along with a collision between the new readouts and the ones already on screen, which the pass after it lays out.

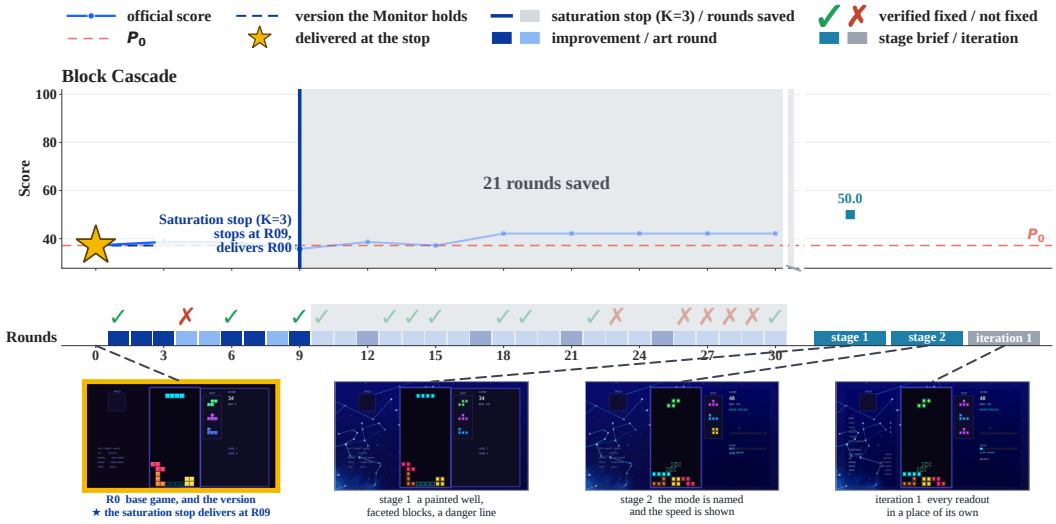

Figure 16: *Block Cascade*, from the generated game to the build we shipped. The generated game already implements every mechanic in the spec, so the improvement rounds have almost nothing to improve: through nine checkpoints the Global Quality Monitor never prefers a later build, the saturation stop ends the run at round 9, and what it delivers is round 0 itself. Everything past the axis break follows a restart from that build.

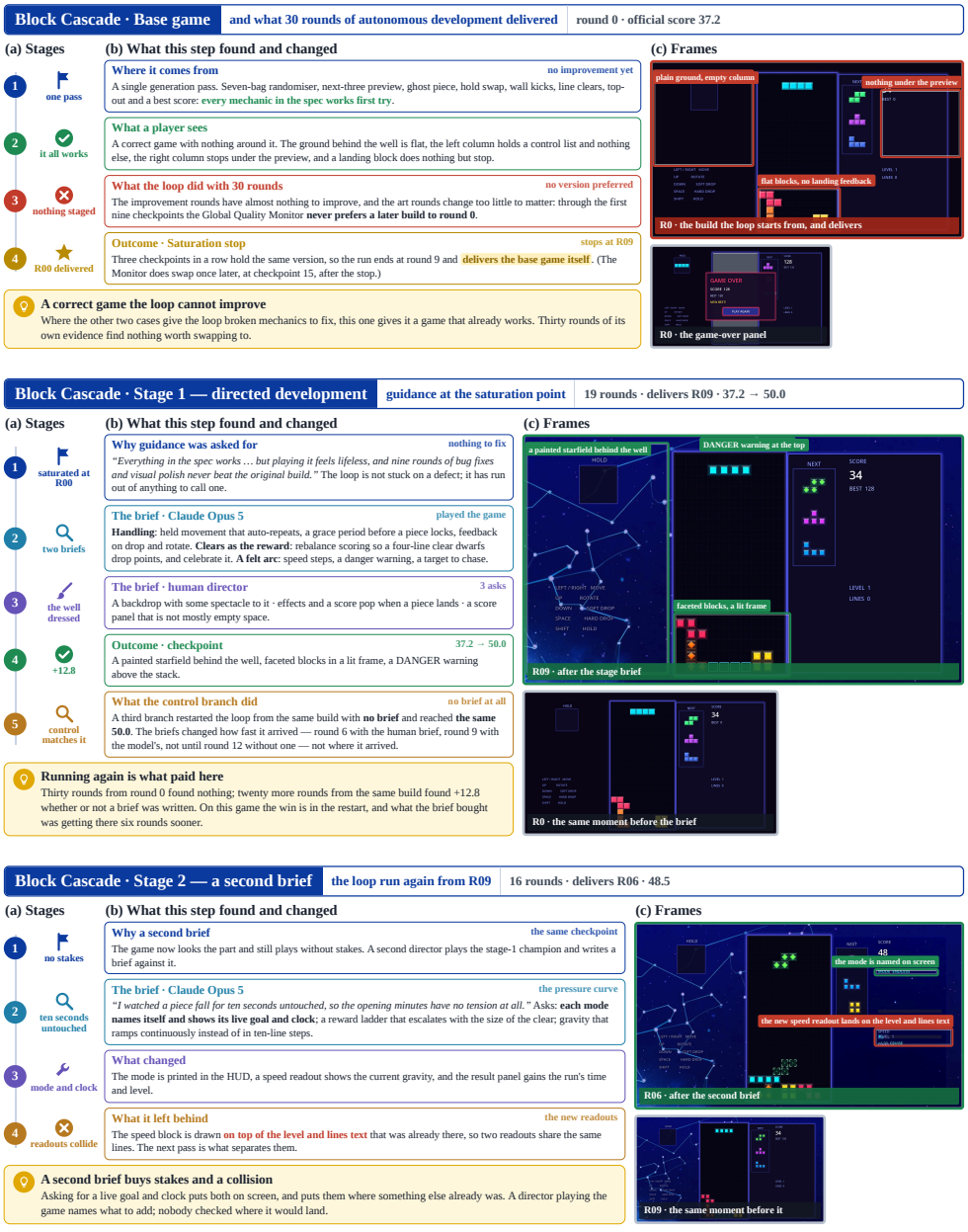

Figure 17: *Block Cascade* from the generated build through both briefs. Cards, boxes and script as in Figure 11. **Base game:** every mechanic in the spec works first try, and nine rounds later the loop delivers this same build. **Stage 1:** the well gets a painted ground, faceted blocks and a danger line — and a control branch with no brief reaches the same score. **Stage 2:** a named mode and a live speed readout arrive, printed on top of the readouts already there.

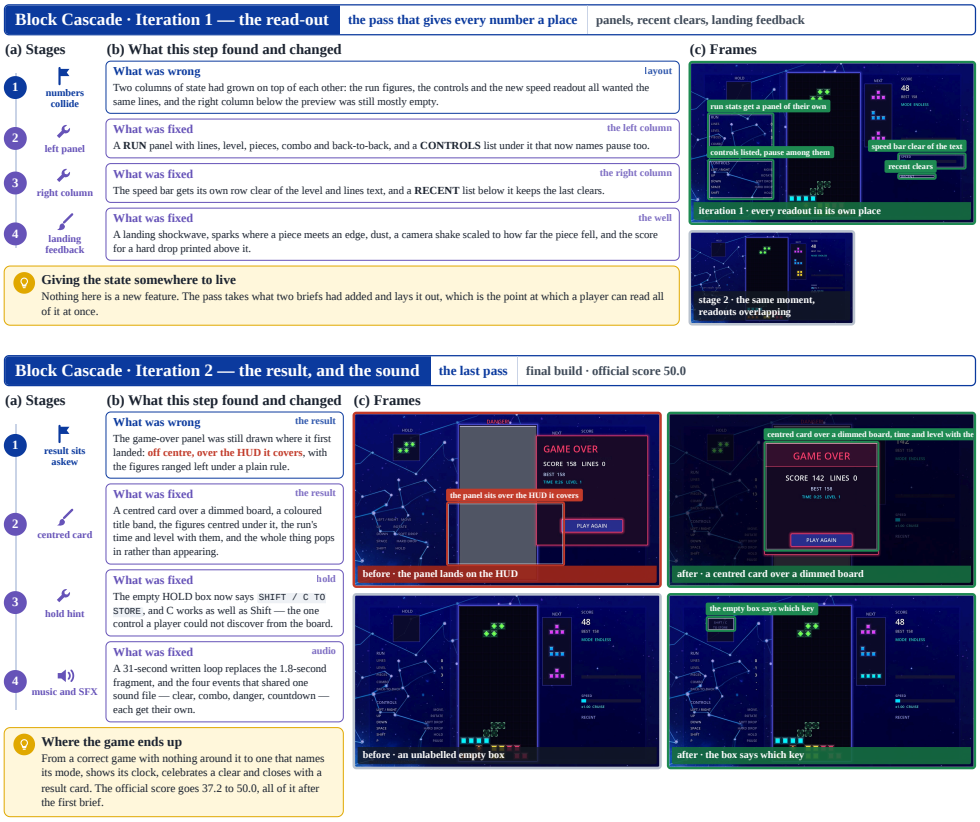

Figure 18: The two development passes that close the run. **Iteration 1** gives every number a place of its own and a landing its feedback. **Iteration 2** adds a centred result card over a dimmed board, a HOLD box that says which key fills it, and a written soundtrack in place of a 1.8-second fragment.

F.3 Per-Family Results

Table 20: Per-family breakdown on Godot for the Codex + GPT-5.5 (high) generator (Table 1, first row group), by mean Overall (↑), Qwen3.8-27B judge. N is the number of tasks in the family (scored/total while scoring is in progress); families are ordered by base score; a task with no generated project scores 0. Abbreviations: **+P2C** Play2Code; –: not yet available on all tasks.

Family	N	Base ↑	+P2C	+RSIGame
idle	4	81.9	86.8	87.5
sports	4	72.8	70.5	81.2
horror	5	67.2	71.4	75.1
openworld	15	59.2	57.9	62.2
tycoon	16	53.3	54.4	73.2
visualnovel	11	52.0	55.8	71.5
roguelike	14	46.1	38.5	56.8
shooter	7	46.0	49.7	64.1
strategy	17	46.0	46.3	61.5
rhythm	5	45.4	44.7	67.9
cardgame	5	44.2	44.3	71.3
puzzle	8	43.4	45.1	53.9
platformer	19	43.3	44.4	58.4
racing	4	43.3	53.4	68.4
simulation	6	38.1	37.7	48.8
All	140	50.26	50.74	64.53

Table 21: Per-family breakdown on Godot for the Kimi-K2.6 generator (Table 1, second row group).

Family	N	Base ↑	+P2C	+RSIGAME
idle	4	53.5	50.1	60.0
sports	4	45.9	57.3	64.4
horror	5	43.1	56.6	63.4
tycoon	16	36.1	40.7	46.4
rhythm	5	33.7	37.1	45.4
shooter	7	33.4	35.9	39.1
simulation	6	30.7	29.8	34.9
visualnovel	11	30.1	36.8	45.3
roguelike	14	29.4	35.1	44.5
racing	4	29.1	39.4	41.8
openworld	15	27.0	33.6	42.5
cardgame	5	26.2	23.6	48.8
strategy	17	24.1	33.1	47.7
platformer	19	23.3	29.2	41.4
puzzle	8	15.3	19.4	29.2
All	140	29.63	35.19	44.77

Table 22: Per-family breakdown on Godot for the GLM-5.3-Flash generator (Table 1, third row group).

Family	N	Base ↑	+P2C	+RSIGAME
sports	4	49.6	61.5	72.6
idle	4	39.0	59.2	68.6
tycoon	16	38.8	43.1	56.1
racing	4	34.9	36.7	53.0
horror	5	34.1	43.6	49.9
visualnovel	11	32.7	38.8	59.0
shooter	7	32.6	42.5	54.9
strategy	17	32.2	39.2	42.5
simulation	6	32.0	38.6	46.5
rhythm	5	26.3	37.2	52.9
cardgame	5	25.5	33.1	52.7
platformer	19	25.3	35.7	48.7
roguelike	14	24.9	37.5	40.9
openworld	15	19.2	28.1	43.2
puzzle	8	16.9	33.3	40.8
All	140	29.46	38.59	49.72

Table 23: Per-family breakdown on Godot for the Qwen3.8-27B generator (Table 1, fourth row group).

Family	N	Base ↑	+P2C	+RSIGAME
horror	5	58.7	56.6	57.0
idle	4	48.8	53.2	73.9
cardgame	5	48.3	53.1	59.3
sports	4	44.8	51.2	55.6
shooter	7	42.3	44.7	61.5
visualnovel	11	42.0	46.0	44.9
tycoon	16	41.0	52.0	62.6
strategy	17	39.5	35.2	44.9
openworld	15	34.5	39.7	36.4
rhythm	5	33.6	35.0	54.7
simulation	6	32.1	36.4	33.3
platformer	19	31.6	31.7	43.5
roguelike	14	31.0	36.7	42.4
puzzle	8	23.5	32.5	43.5
racing	4	23.3	22.7	27.9
All	140	37.07	40.53	47.77

Table 24: Per-family breakdown on Godot for the Qwen3.8-27B (SFT) generator (Table 1, fifth row group).

Family	N	Base ↑	Play2Code ↑	RSIGAME ↑
idle	4	80.4	78.6	84.1
sports	4	73.3	68.7	76.1
horror	5	60.6	62.4	71.5
tycoon	16	54.7	56.4	66.3
strategy	17	52.1	51.0	62.0
visualnovel	11	50.2	48.9	58.4
openworld	15	48.1	46.7	58.2
roguelike	14	46.4	47.8	60.2
platformer	19	42.3	46.9	56.0
racing	4	41.3	38.5	52.1
puzzle	8	40.6	44.8	60.3
simulation	6	40.4	45.1	61.2
rhythm	5	37.4	43.9	63.8
cardgame	5	36.1	41.8	58.9
shooter	7	35.3	39.8	55.2
All	140	48.22	49.71	61.38

Table 25: Per-family breakdown on Phaser for the OpenGame + GPT-5.5 generator (Table 1, Phaser block, first row group).

Family	N	Base ↑	+P2C	+RSIGAME
idle	4	72.7	73.8	78.2
cardgame	5	68.9	68.9	61.8
visualnovel	11	68.0	68.6	73.5
tycoon	16	59.9	62.9	74.3
sports	4	57.2	59.8	64.8
roguelike	14	56.5	58.2	66.7
horror	5	53.5	52.7	57.6
simulation	6	51.8	57.7	55.4
rhythm	5	50.8	52.5	58.2
strategy	17	45.7	56.8	54.8
shooter	7	45.2	58.6	60.4
racing	4	39.8	60.4	52.8
platformer	19	38.1	43.7	45.0
openworld	15	32.8	41.0	39.5
puzzle	8	32.5	40.6	51.6
All	140	49.44	55.10	58.21

Table 26: Per-family breakdown on Phaser for the Qwen3.8-27B generator (Table 1, Phaser block, second row group).

Family	N	Base ↑	+P2C	+RSIGAME
sports	4	56.1	68.8	65.4
horror	5	55.6	52.9	57.6
simulation	6	55.3	53.3	67.8
visualnovel	11	53.3	54.2	52.4
rhythm	5	50.9	58.4	61.4
tycoon	16	47.5	58.9	56.7
cardgame	5	47.0	72.3	61.4
puzzle	8	42.6	44.2	39.5
idle	4	41.3	61.5	65.2
racing	4	40.9	45.3	51.6
strategy	17	37.3	50.0	50.6
platformer	19	37.1	40.2	46.2
roguelike	14	29.6	40.7	48.8
shooter	7	27.7	46.0	48.1
openworld	15	21.1	32.0	29.3
All	140	40.03	48.70	50.24

Table 27: Per-family breakdown on Phaser for the Qwen3.8-27B (SFT) generator (Table 1, Phaser block, third row group).

Family	N	Base ↑	Play2Code ↑	RSIGAME ↑
idle	4	73.0	75.2	78.5
horror	5	56.0	60.7	67.5
tycoon	16	53.3	62.0	66.5
sports	4	51.6	49.9	56.8
shooter	7	41.8	51.4	55.4
rhythm	5	46.8	61.3	66.2
simulation	6	46.0	50.7	60.1
roguelike	14	45.5	56.8	60.2
openworld	15	42.2	46.0	50.7
puzzle	8	43.8	58.7	64.8
strategy	17	42.9	51.3	54.9
cardgame	5	41.6	45.1	58.7
visualnovel	11	40.0	46.2	54.3
platformer	19	39.9	50.4	54.1
racing	4	32.8	39.1	49.6
All	140	45.14	53.15	58.53