RECURSIVE GAME CREATOR: AGENTIC PRODUCT-LEVEL EXPERIENCE-ORIENTED GAME HARNESS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents can quickly produce runnable games, but these prototypes can still fall far short of product-ready quality. Closing this gap requires engaging gameplay, balanced difficulty, clear feedback, coherent presentation, and reliable execution. We present Recursive Game Creator, a game development harness designed from the question of how to evaluate player experience. Different games have different rules and goals, so one fixed set of tests cannot assess them all. We therefore use two levels of evaluation. A meta evaluation protocol uses success rates and play time as shared signals. The Reviewer interprets them with a game-specific rubric: a set of checks suited to each game. It compares two game versions using play evidence and suggests changes. To support these checks, the Player writes proxy policies, or programs that play the game, to collect play records and screenshots. These programs act without a language-model call at each step. The Designer and Builder turn the feedback into changes, and the next round checks the result. We have built a harness that combines game generation, playtesting, and repeated review in one development loop. This provides a concrete step toward producing complete games that people enjoy and want to keep playing.

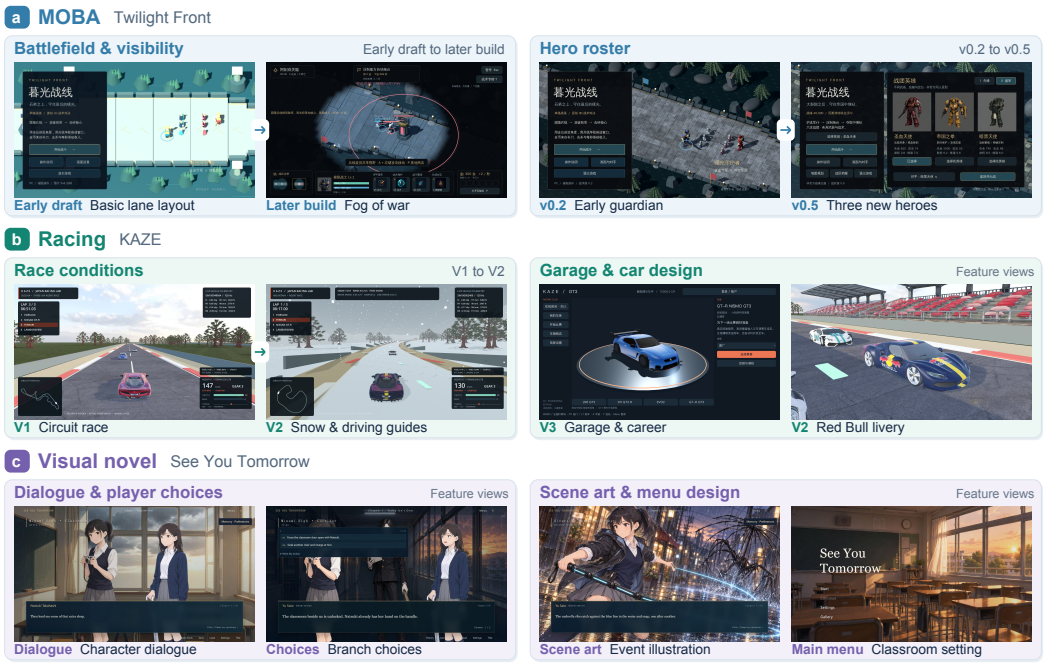

Figure 1: Game examples from Recursive Game Creator. Rows show MOBA, racing, and visual-novel games, each with two paired views of gameplay, characters, or presentation. Arrows mark documented earlier-to-later versions.

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.

1 INTRODUCTION

Coding agents can quickly build games that run. But a runnable game is not necessarily an attractive and popular game. There are aspects crucial to game experience: engaging gameplay, appropriate difficulty, clear feedback, etc. At the same time, these aspects are hard to evaluate quantitatively. Existing agentic game creation workflows (Yan et al., 2026; Hu et al., 2026) focus on correctness but overlook the human experience.

In this work, we aim to bridge a gap in *agent-native game creation*: using an agent harness to turn draft games into games of outstanding quality that people can play and enjoy. We build Recursive Game Creator with four roles: Designer, Builder, Player, and Reviewer (Figure 2), mirroring a real game studio. Our key insight is that accurate and informative evaluation signals are crucial to the evolution of game development. We view a Reviewer as core to Recursive Game Creator, combining general standard and game-specific rubrics. The evaluation requires a large amount of data to draw unbiased conclusions. Thus, we design a Coding Native Player that can sample a large amount of trajectories for comprehensive evaluation. Unlike Gui-Agents (Huang et al., 2026), which uses agents to play with screenshots, we find that agents that write policy programs to sample trajectories are much more effective and efficient.

There is no single set of rules that can fully evaluate every game. Games come in many forms: a racing game needs realistic visuals and responsive controls, while a visual novel needs an attractive story and dialogue. These differences make one-for-all automatic evaluation difficult. Therefore, we propose an evaluation protocol that combines general standards with game-specific rubrics. The general metrics are success rate and playtime. A suitable success rate can encourage players to keep trying to win. The time players spend in a game reflects the overall experience of playing it. The game-specific rubric examines the game's content and observed behavior to define checks for its mechanics, controls, story, and presentation. It extends the development goal beyond program correctness to the quality of play and player experience.

Estimating success rates and studying playtime require sufficient gameplay records spanning repeated attempts and different behaviors. Visual evidence already supports game review: AVR-Agent uses audio-visual recordings for revision feedback, and GameCraft-Bench judges replayed gameplay (Jolicoeur-Martineau, 2025; Luo et al., 2026) at 2FPS. Collecting this evidence through direct visual interaction can be costly. In screenshot-based play, the model repeatedly reads the screen and decides how to act. VideoGameBench identifies model response time as a major limit in real-time play (Zhang et al., 2026a). Such delays can prevent timely control in racing games and first-person shooters (FPS). Repeated image processing also uses time and model calls, limiting the number of playthroughs within a fixed budget. A small set of sampled views may miss brief events, while a small set of playthroughs may miss different strategies and failures.

We propose *Coding-Native Player Policies*. The agent writes and refines programs that play the game, using different strategies and skill settings. Once written, each policy can be reused to collect trajectories without intervention of coding agent. It supports fast interaction and provides the Reviewer with a wider range of gameplay records, alongside sampled visual evidence. The policies serve as a group of proxy players for repeated testing. Their success rates and run times reflect the tested behaviors. We also support human feedback during development. The Designer and Builder close the loop by applying the review feedback, with harness constraints on tool use and checks on the resulting changes.

We make three contributions:

- **Human-Oriented Reviewer.** We turn the goal of human enjoyment into feedback suited to each game. The Reviewer combines shared signals of success and playtime with game-specific rubrics to assess whether changes improve challenge, interest, and the overall play experience.
- **Coordinated Player.** We connect the Reviewer's need for gameplay evidence with a Player that writes and refines reusable policies. Different policies collect varied play records without per-action language model calls, supporting frequent control and repeated testing at a lower inference cost.

- **Self-Evolution Game Design Workflow.** We build a workflow that turns review feedback into new designs, game changes, and further playtests. This repeated process supports the design of games that people enjoy and want to keep playing.

2 Related Work

Applications of coding agents. Code generation allows agents to work beyond software development, including video creation and embodied tasks. For educational videos, Code2Video combines planning, Python code generation, and visual review to improve the rendered layout (Chen et al., 2025b). VideoAgent uses generated code to create animations and combines them with slides and narration for scientific videos (Liang et al., 2026). VideoCoCo uses a coding agent to write Blender programs that produce dynamic scene drafts. A video generation model then turns these drafts into realistic videos (Li et al., 2026). In embodied control, Code as Policies turns language instructions into robot policy code that connects perception with control APIs (Liang et al., 2023). ProgPrompt uses program-like descriptions of available actions and objects to generate robot task plans (Singh et al., 2022). Voyager builds a library of reusable Minecraft skills and revises their code using environment feedback and execution errors (Wang et al., 2023). Eureka extends code generation to reward design, using feedback from policy training to revise reward functions (Ma et al., 2024). Across these applications, code gives agents a way to express, execute, and reuse structured behavior. These works motivate our use of executable policies for gameplay testing. Our focus is on collecting varied play evidence and using it to guide changes to the game through experience-based review.

Game development with coding agents. Coding agents generate and edit game programs within an explicit runtime. General software-agent systems such as MetaGPT and ChatDev organize planning, implementation, and testing (Hong et al., 2024; Qian et al., 2024); GameGPT applies role-based collaboration to game development (Chen et al., 2025a). AutoUE targets Unreal Engine workflows, while OpenGame combines reusable project templates with debugging knowledge (Yin et al., 2026; Jiang et al., 2026). Beyond initial generation, AVR-Agent iterates JavaScript content using audio-visual recordings and multimodal comparisons, and Harness-of-Harness supports persistent development with independent assessment (Jolicoeur-Martineau, 2025; Yan et al., 2026). MuKV uses multi-grained KV-cache compression and retrieval for efficient streaming video QA (Xiao et al., 2026), supporting timely visual understanding, which is crucial for conventional GUI-based players. GameDevBench and GameCraft-Bench evaluate engine-grounded development and interactive artifacts; GameXpert-Bench extends evaluation across generation, repair, and cumulative optimization (Chi et al., 2026; Luo et al., 2026; Chen et al., 2026). Thus, iteration and gameplay-based assessment are established components. Recursive Game Creator focuses on a policy-generating Player that supplies behavioral trajectories and sampled replays to a separate Reviewer; the value of this interface requires comparison with existing iterative systems.

Feedback refinement and preference alignment. Self-Refine and Reflexion use feedback to revise outputs or subsequent attempts (Madaan et al., 2023; Shinn et al., 2023); AFlow, EvoMAC, and the Darwin Gödel Machine instead optimize workflows, collaboration, or agent code (Zhang et al., 2025; Hu et al., 2025; Zhang et al., 2026b). A related line makes the optimization objective sensitive to human preferences: trajectory comparisons can train reward models, while RLHF and direct preference optimization align language-model behavior with preferred outputs (Christiano et al., 2017; Ouyang et al., 2022; Rafailov et al., 2023). VideoDPO extends preference optimization to video diffusion using automatically scored pairs, which should be distinguished from direct human judgments (Liu et al., 2025). In games, experience-driven procedural content generation connects player modeling to content adaptation, and experience-driven RL generates racetracks toward target affective patterns (Yannakakis & Togelius, 2011; Barthet et al., 2024). These works motivate feedback tied to the intended experience, while differing in what is optimized and who supplies the feedback. Our Reviewer uses game-specific evidence to select and revise artifacts with fixed model parameters; it is neither a learned human-preference reward model nor a substitute for evaluating enjoyment with human players.

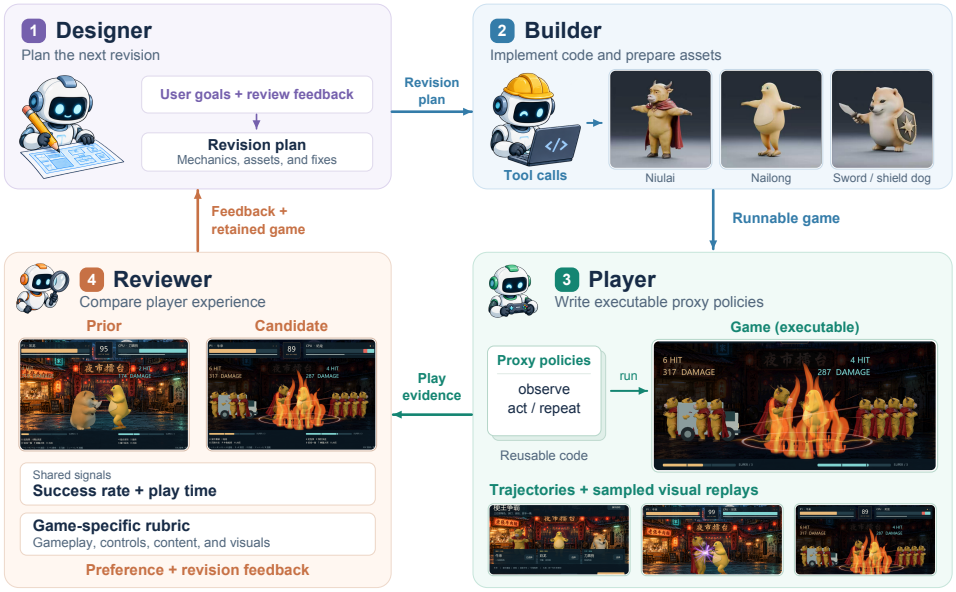

Figure 2: Recursive Game Creator's refinement workflow. The Designer and Builder revise the game. The Player collects play evidence using executable policies. The Reviewer combines success rate and play time with a game-specific rubric to compare versions and return feedback. The harness selects the version to retain. Author-provided character renders illustrate the Builder's assets. Meme Arena screenshots illustrate the Reviewer's version comparison and the Player's visual inputs. Prior and candidate labels explain the workflow; actual review uses anonymous labels. The displayed frames are illustrative inputs, not a matched test or a recorded trajectory.

3 Recursive Game Creator: Closed-Loop Game Refinement

Recursive Game Creator uses self-evolution to develop an attractive game through successive rounds of implementation, playtesting, and revision. The intended experience combines engaging mechanics, a reason to continue playing, appropriately demanding challenges, clear feedback, and reliable execution. Refinement therefore targets mechanics and difficulty alongside implementation defects, narrative coherence, and visual presentation. To evaluate these dimensions, we separate evidence obtainable through programmatic interaction, such as completion, collisions, and progression, from evidence that requires rendered views, such as visual coherence, text readability, and the clarity of on-screen feedback. This distinction motivates a coding-native Player: executable proxy policies collect interaction trajectories, while sampled visual replays support appearance and presentation assessment. Decoupling these channels allows frequent gameplay interaction without equally frequent visual-model calls. A human-oriented Reviewer combines both channels to identify experience-relevant problems and guide the next revision.

3.1 Closed-Loop Refinement

The Designer interprets the user specification and the previous review to propose changes to the retained game. The Builder implements a candidate, and the Player collects behavioral and visual evidence from that version. The Reviewer compares the candidate and retained game under anonymous labels using the same game-specific experience criteria, then returns a preference judgment and concrete revision objectives. The harness maps this judgment back to the versions for retention decisions. An accepted candidate becomes the starting point for the next round; otherwise, the prior version is preserved. This loop evolves the game artifact while keeping the underlying language-model parameters fixed. Figure 1 shows selected game views, including changes in battlefield detail, race conditions, and game interfaces.

3.2 CODING-NATIVE PLAYER

GUI-based agents play by reading screenshots and acting through the user interface. Reading each image and choosing an action takes time. This limits how often the agent can act. This can make fast-paced games difficult to test and restrict the amount of play data collected. Yet evaluating success rate and playtime requires many trajectories from different behaviors. We therefore use a coding agent's strength in writing code: it builds policies that play through the game's programmatic interface. We call this approach *coding-native* because the agent expresses play as executable code, rather than reading an image to choose every action.

Playing through the command-line interface. The game exposes its state and valid actions through a command-line interface. A policy reads the state values, chooses the next action, and sends it back through this interface. This control loop requires no UI operation, image interpretation, or language-model call. The policy code is lightweight and can run at the rate supported by the game. Separate game instances can run in parallel. The agent checks the resulting trajectories and refines a policy when its behavior does not match the intended strategy.

Diverse policies and parallel sampling. Players differ in skill and preferences, so one policy cannot represent all ways of playing a game. The Player builds policies with different strategies and skill settings to sample a wider range of behavior. For example, one policy may favor fast progress, while another explores more options or takes fewer risks. Running these policies in parallel produces trajectories with different actions, outcomes, and run times. The Player sends these records and replay data to the Reviewer. The policies serve as a varied group of proxy players; their settings do not establish that they match the full human player population.

Advantages of coding-native play. This design supports frequent actions without waiting for a model response at every step. It can therefore test games that require fast control, such as racing and first-person shooters. Lightweight execution and parallel sampling also make it practical to collect many trajectories. Different policies explore different situations and can reveal bugs that one strategy misses. These records give the Reviewer evidence for assessing success rate, playtime, and difficulty. The Player and Reviewer thus work together: the Player supplies varied behavior, and the Reviewer uses it to identify changes that may improve the game. Total testing cost still includes policy generation, repair, execution, and review.

3.3 HUMAN-ORIENTED REVIEWER

There is no single fixed rubric that can fully evaluate every game. Our Reviewer is *human-oriented*: it judges a game in terms of the intended player experience and preferences. It checks more than whether the program runs. It also considers whether bugs disrupt play, controls respond as expected, visuals support the game, and the content gives players a reason to continue. The review combines shared signals of success rate and playtime with a game-specific rubric. Together, they guide revisions toward the experience that the game aims to provide.

Shared signals: success rate and playtime. The coding-native Player supplies many trajectories from policies with different skills and strategies. The Reviewer uses this evidence to estimate success rates and examine where play ends or gets stuck. Success that comes too easily may offer little challenge, while repeated failure may cause frustration. The goal is therefore a suitable level of difficulty for the intended players, not the highest possible success rate. Comparing outcomes across policies helps assess whether progress is achievable and whether stronger play is rewarded. For version comparisons, policy settings and observation access must remain consistent.

The same trajectories record playtime. The Reviewer examines run times together with progress, explored content, and reasons for stopping. This helps distinguish a long run with varied activity from one spent stuck at the same point. Success rate and playtime therefore support a broader assessment of difficulty and playability. Both remain measures of the tested policies; proxy playtime alone does not show human interest or enjoyment.

Game-specific rubric. The Reviewer examines the game's content and play evidence to define checks suited to its goals. A racing game may need clear track guidance, responsive steering, and

visible collision feedback. A visual novel may need a coherent story, readable dialogue, and choices with clear effects. Other checks cover varied content, pacing, visual style, and reasons to explore or replay. The Reviewer also checks whether the rules and visual cues agree with what happens during play. It explains the strengths, weaknesses, and tradeoffs behind its judgment, rather than simply averaging unrelated criteria.

Visual evidence and diagnosis. The Reviewer examines play reports, sampled screenshots, and available recordings without seeing source code or version order. Trajectories show where a policy succeeds, fails, or stops progressing. Rendered views show what a player could see at those moments. The Reviewer uses them to assess text readability, visual style, and action feedback. Visual review is separate from the fast control loop, so collecting a trajectory does not require interpreting an image at every step. Brief effects or missing feedback may still require a recording rather than a few frames.

Each observation is tied to its game version, policy, and replay. The Reviewer separates what happened from why it may have happened. A failed turn could reflect poor controls, an unclear cue, or a weak policy. When the cause is uncertain, the feedback specifies what further evidence would help resolve it.

Revision feedback and selection. The Reviewer compares a candidate with the retained game using the same shared signals and game-specific criteria. It checks whether the change fixes the earlier problem and causes any new ones. The output is an A/B preference, a tie, or an unavailable judgment, with reasons and evidence paths. Feedback states the problem, its possible cause, its effect on play, and a revision goal with a follow-up check. The harness maps the anonymous judgment back to the game versions and selects which one to retain. The Designer and Builder then apply the feedback. Human input can also guide this loop, while automated review remains distinct from measured human enjoyment.

4 EXPERIMENTS

We evaluate Recursive Game Creator on GameCraft-Bench and GameASG-Bench. These benchmarks test game quality and whether generated games meet their requirements. We then examine Player exploration, game examples, and expert scores across refinement rounds.

4.1 MAIN RESULTS

4.1.1 GAMECRAFT-BENCH

GameCraft-Bench evaluates complete Godot games using replayed gameplay and game-specific rubrics (Luo et al., 2026). It checks mechanics, content depth, functional visuals, and art. We score games after each refinement round using the same weighted scoring formula as Harness-of-Harness (HoH) (Yan et al., 2026). Table 1 reports category and overall scores. It also includes published HoH results and a GPT-6 Astra reference. The table notes the different task sets used by these configurations.

Overall quality improves with each refinement round, and the gains appear in every reported game category. Action games show the largest improvement. This pattern supports repeated playtesting and revision as a way to improve the generated games under the benchmark rubric. It does not isolate the contribution of each role; that requires a comparison with the same development budget.

4.1.2 GAMEASG-BENCH

GameASG-Bench tests whether generated browser games meet their source and behavior requirements (Zhang et al., 2026c). L1 checks the source code, while L2 runs the game in a browser. P0 covers startup and the test interface, P1 covers required gameplay, and P2 covers extended features. Task success requires valid delivery, completed evaluation, and passing every L1 check and every applicable L2 P0/P1 check. We report this measure alongside check pass rates in Table 2.

Recursive Game Creator achieves the highest reported mean L2 pass rate and core-gameplay pass rate, while its task success rate is close to the strongest published configuration. This shows that

Table 1: GameCraft-Bench results (Luo et al., 2026). Scores use the HoH reporting formula (0–100; higher is better). Published HoH results (Yan et al., 2026) appear above our results. @k denotes refinement round k.

Method	Action	Timing	Strategy*	Simulation	Adventure	Overall*
Codex + GPT-5.5 (high) (HoH paper; 45 tasks)						
Vanilla	48.74	48.80	44.06	53.63	52.68	49.58
HoH@1	59.73	53.79	57.01	66.75	61.24	59.71
HoH@2	64.34	62.03	59.97	71.77	66.11	64.84
HoH@3	71.02	70.26	66.13	78.42	71.76	71.52
OpenCode + DeepSeek-V4-Pro (HoH paper; 45 tasks)						
Vanilla	26.21	24.05	21.27	37.12	25.84	26.90
HoH@1	27.75	27.40	21.73	43.75	22.44	28.61
HoH@2	43.22	36.56	33.51	52.26	36.05	40.32
HoH@3	49.00	45.05	43.34	55.86	51.64	48.98
Pi + MiniMax-M3 (HoH paper; 45 tasks)						
Vanilla	45.64	38.20	34.53	48.19	44.26	42.16
HoH@1	50.59	52.23	38.60	54.00	49.89	49.06
HoH@2	54.70	56.52	42.33	63.20	58.47	55.04
HoH@3	58.24	62.10	44.86	64.25	64.44	58.78
This work						
GPT-6 Astra (high)	73.33	64.26	74.83	–	–	70.31†
Recursive Game Creator@1	67.96	68.76	72.10	77.72	76.95	72.70
Recursive Game Creator@2	70.33	73.19	72.96	80.57	79.69	75.35
Recursive Game Creator@3	77.62	74.41	74.46	82.94	80.00	77.89

* For this work, Strategy covers three Puzzle items; HoH covers nine Strategy, Card Game, and Puzzle tasks. Overall covers 45 tasks for HoH and 21 for Recursive Game Creator. † GPT-6 Astra Overall covers 24 games. Dashes mean not run. Published scores are not reruns with our harness.

the harness can produce games that meet a broad range of runtime requirements. High average pass rates still leave some tasks incomplete, so reliable end-to-end delivery remains a challenge. Our configuration uses medium reasoning effort; the published baselines use their maximum settings. The task sets also differ, as noted in the table, so these results do not establish a gain under matched settings.

Together, the two benchmarks provide evidence of improving game quality across rounds and strong behavioral compliance. They assess the games produced by the full workflow; they do not separately measure the effects of the Reviewer and Player or establish human enjoyment.

4.2 Player Exploration Analysis

Figure 3 shows exploration in an FPS scene. The recorded policy rollouts reach areas across the map, including building interiors. A reusable policy collects this evidence without a model call for every action. The GUI routes are illustrative, so the figure shows coverage rather than a measured speedup. A controlled comparison must match budgets and observation access, including the cost of policy construction, repair, execution, and review.

4.3 Qualitative Analysis

Figure 1 shows three game examples. Twilight Front adds a fog-of-war display and expands its hero roster. KAZE includes changing race conditions, a career garage, and car liveries. See You Tomorrow combines character dialogue, player choices, event art, and a classroom menu. The examples show the range of content and interfaces in the games. The screenshots are not matched play sessions and do not measure the effect of each change on player experience.

Figure 2 uses Meme Arena screenshots to illustrate version comparison and visual inputs. The frames show changes in scene detail and combat effects. They are workflow examples, not matched tests of player experience.

Table 2: GameASG-Bench results. Baselines are from Zhang et al. (2026c), Table 3. All scores are percentages; task success also reports the count. Bold marks column maxima.

Model	Harness	Task success	L1	L2 Mean	L2 P0	L2 P1	L2 P2
GPT-6-Astra ultra	Codex CLI	**26/47 (55.3%)**	98.0	93.2	99.0	92.7	**92.4**
Claude-Opus-5	Claude Code	24/47 (51.1%)	**99.6**	90.4	91.2	89.9	91.1
GPT-5.6-Sol	Codex CLI	21/47 (44.7%)	98.9	91.3	**100.0**	89.7	91.9
DeepSeek-V4-Flash	Claude Code	18/47 (38.3%)	99.4	89.3	98.0	86.7	90.2
DeepSeek-V4-Pro	Claude Code	15/47 (31.9%)	**99.6**	85.2	85.3	83.9	88.9
Kimi-K3	Claude Code	15/47 (31.9%)	97.7	88.5	98.0	86.3	89.7
GLM-5.2	Claude Code	11/47 (23.4%)	98.8	84.6	91.2	83.3	83.0
Hunyuan-3	Claude Code	10/47 (21.3%)	98.3	81.7	96.1	78.2	82.3
MiniMax-M3	Claude Code	7/47 (14.9%)	98.3	70.6	91.2	64.5	75.4
GPT-6-Astra medium	**Recursive Game Creator**	25/46 (54.3%)	98.3	**93.4**	99.0	**92.8**	92.2

Our model uses medium reasoning effort; baselines use their maximum settings. L1 and L2 Mean average per-task check pass rates; P0–P2 pool applicable checks within each priority. Task success uses the benchmark's strict definition.

4.4 Expert Scoring

Table 3 reports ratings for controls, playability, depth, and art on a 1–10 scale, alongside average playtime. All four ratings increase across the three rounds. Average playtime also increases across rounds. Longer playtime alone does not establish greater enjoyment.

Table 3: Expert scores and average playtime across refinement rounds. Controls, playability, depth, and art are rated on a 1–10 scale. Average playtime is reported in minutes.

Round	Controls	Playability	Depth	Art	Avg. playtime (min)
Round 1	2	1	3	2	1.50
Round 2	5	4	4	6	4.87
Round 3	8	7	6	8	12.41

4.5 Implementation and Reporting Scope

The draft identifies Codex with GPT-6 Astra at medium reasoning effort, Godot and HTML-based games, and the Hunyuan3D API for asset generation. These describe the stated configuration, not independently verified execution records. Exact model and engine versions, prompts, task selection, iteration limits, API configuration, token accounting, and runtime measurements remain to be documented. Godot benchmark results should be reported separately from HTML case studies.

5 Discussion and Limitations

Reusing policies makes gameplay evidence a practical input to development, but more trajectories and rounds do not guarantee better games. Similar policies can miss the same failures, and revisions can satisfy the Reviewer without improving human experience. Proxy policies may use privileged state or miss natural human behavior. Sampled visuals can miss brief defects, while a Reviewer and Builder based on the same model may share blind spots.

The results show higher game-quality scores across rounds and strong runtime compliance. Matched tasks and budgets are still needed to measure each role's contribution, testing efficiency, and development time. Human studies are needed to assess enjoyment and retention. Broader engine support and product readiness also require tests of stability, accessibility, asset rights, and deployment.

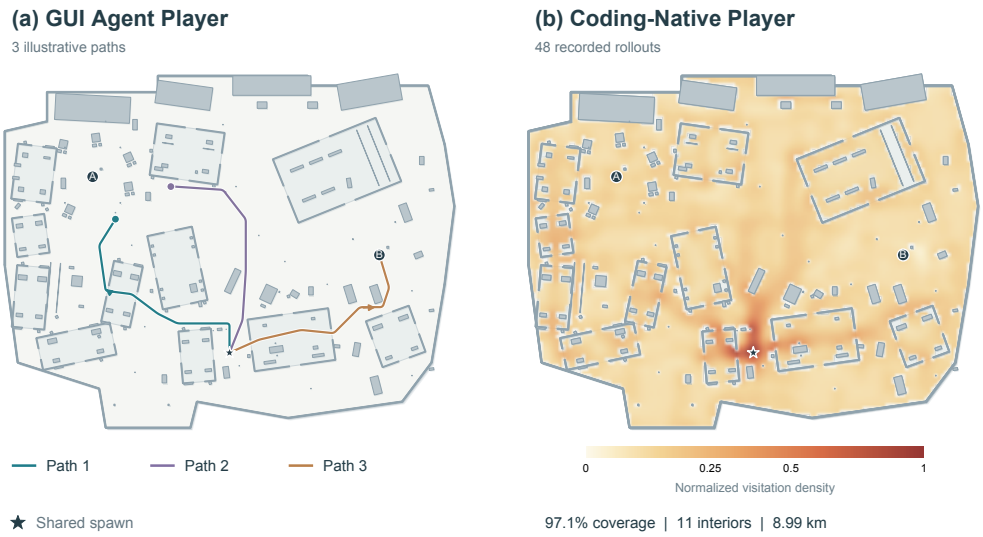

Figure 3: Spatial exploration in an FPS test scene. Left: three illustrative GUI-style routes from a shared spawn. Right: visitation from 48 recorded coding-native policy rollouts. Color shows normalized visitation density. The source reports 97.1% grid coverage, including 11 interiors. The GUI paths are illustrative, so this is not a matched-budget comparison.

5.1 Humility: Self-Improvement of the Harness

During early development, we also let the agent improve the harness itself. We observe problems in the workflow and describe them to the agent, together with the goal that the generated games should be playable. The agent then inspects and revises the harness. We use the revised workflow to generate games and check whether it addresses the earlier problems. Further observations guide the next changes.

This process differs from refining one game. Game refinement changes a game artifact; harness self-improvement changes the procedure used to build and evaluate games. Human developers provide observations and the final goal, while the agent makes the harness changes. The underlying model remains fixed. In our early development experience, this process helped address workflow problems and improve playability. This is a qualitative observation, not a controlled comparison of harness versions.

6 Conclusion

Recursive Game Creator organizes game refinement around four roles and an explicit retain-or-revert decision. Its central design is a coding-native Player that generates proxy policies, collects trajectories, and supplies sampled visual evidence to a game-specific Reviewer. This formulation connects automated testing to design revision while keeping human experience as a separate evaluation target. GameCraft-Bench scores improve across refinement rounds, while GameASG-Bench results show strong compliance with runtime requirements. Matched comparisons and complete cost accounting are still needed to measure each component's effect and testing efficiency. The current results do not establish human enjoyment.

AI USE STATEMENT

Generative AI tools were used for translation and language editing to improve readability, sentence ordering, and logical flow, as well as to assist in preparing schematic diagrams. They were not used to construct datasets, generate research ideas, or design the proposed method. All AI-assisted text and figures were reviewed by the authors, who take full responsibility for their accuracy and integrity.

ETHICS STATEMENT

The gameplay data were collected by the authors for non-commercial research. The underlying game content remains subject to the rights of its respective owners; we do not claim ownership of these assets or intend to release them for commercial use. All robotic experiments were conducted in simulation, without physical robot deployment.

REPRODUCIBILITY STATEMENT

The main text and appendix describe the data collection and preprocessing procedures, model architecture, training configurations, and evaluation protocols to support reproduction of our experiments. Additional implementation details are provided in the appendix.

REFERENCES

Matthew Barthet, Diogo Branco, Roberto Gallotta, Ahmed Khalifa, and Georgios N. Yannakakis. Closing the Affective Loop via Experience-Driven Reinforcement Learning Designers, 2024. URL https://arxiv.org/abs/2408.06346.

Dake Chen, Haoyang Zhang, Hanbin Wang, Yunhao Huo, Yuzhao Li, and Junjie Wang. GameGPT: Multi-agent Collaborative Framework for Game Development, 2025a. URL https://arxiv.org/abs/2310.08067.

Kun Chen, Haorong Hong, Peizhong Gao, Jianfeng Lin, Tongxu Luo, Yuxuan Xie, Chenxu Liu, Jieling He, Zhongyuan Liu, and Zeno Zeng. GameXpert-Bench: How Far Are Coding Agents from Expert Game Development?, 2026. URL https://arxiv.org/abs/2608.21833.

Yanzhe Chen, Kevin Qinghong Lin, and Mike Zheng Shou. Code2Video: A Code-centric Paradigm for Educational Video Generation, 2025b. URL https://arxiv.org/abs/2510.01174.

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, 2026. URL https://arxiv.org/abs/2602.11103.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/d5e2c0adad503c91f91df240d0cd4e49-Paper.pdf.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, zili wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/6507b115562bb0a305f1958ccc87355a-Paper-Conference.pdf.

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. URL https://github.com/tettethu/VibeGame/blob/main/technical_report.pdf.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-Evolving Multi-Agent Collaboration Networks for Software Development. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 23007–23039, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/39af4f2f9399122a14ccf95e2d2e7122-Paper-Conference.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. URL https://arxiv.org/abs/2605.28258.

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. URL https://arxiv.org/abs/2604.18394.

Alexia Jolicoeur-Martineau. Multi-Agent Game Generation and Evaluation via Audio-Visual Recordings, 2025. URL https://arxiv.org/abs/2508.00632.

Haodong Li, Tianfei Ren, Xiaoxiao Ma, Chunmei Qing, Zhen Fang, Sipeng He, Ziyu Guo, Haoyu Wu, Juanxi Tian, Yihang Zou, Ruichuan An, Dongzhi Jiang, Boxue Yang, Ji Xie, Xu Huang, Wenhao Yan, Jialv Zou, Zhengrong Yue, Yaxin Luo, Xiaotong Li, Yuzhu Wang, Junyan Ye, Jinjing Zhao, Zehui Chen, Lin Chen, Renye Yan, Feng Zhao, and Pheng-Ann Heng. VideoCoCo: Code-as-CoT for Physically-Consistent Video Generation via an Agentic Dual-Engine System, 2026. URL https://arxiv.org/abs/2607.27380.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as Policies: Language Model Programs for Embodied Control, 2023. URL https://arxiv.org/abs/2209.07753.

Xiao Liang, Bangxin Li, Zixuan Chen, Hanyue Zheng, Zhi Ma, Di Wang, Cong Tian, and Quan Wang. VideoAgent: Personalized Synthesis of Scientific Videos, 2026. URL https://arxiv.org/abs/2509.11253.

Runtao Liu, Haoyu Wu, Ziqiang Zheng, Chen Wei, Yingqing He, Renjie Pi, and Qifeng Chen. Videodpo: Omni-preference alignment for video diffusion generation. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 8009–8019, 2025.

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. URL https://arxiv.org/abs/2606.17861.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models, 2024. URL https://arxiv.org/abs/2310.12931.

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 A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 27730–27744. Curran Associates, Inc., 2022. doi: 10.52202/068431-2011. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 53728–53741. Curran Associates, Inc., 2023. doi: 10.52202/075280-2338. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a85b405ed65c6477a4fe8302b5e06ce7-Paper-Conference.pdf.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating Situated Robot Task Plans using Large Language Models, 2022. URL https://arxiv.org/abs/2209.11302.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, 2023. URL https://arxiv.org/abs/2305.16291.

Junbin Xiao, Jiajun Chen, Tianxiang Sun, Xun Yang, and Angela Yao. Mukv: Multi-grained kv cache compression for long streaming video question-answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11381–11391, June 2026.

Haoyang Yan, Min le Su, Hangfan Zhang, Zhanhao Li, Chen Zhang, Shao Zhang, Yang Chen, Lei Bai, and Shuyue Hu. Harness-of-Harness: Multi-Day Autonomous Software Development with Continual Improvement, 2026. URL https://arxiv.org/abs/2609.01481.

G. N. Yannakakis and J. Togelius. Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, July 2011. ISSN 1949-3045. doi: 10.1109/t-affc.2011.6. URL http://dx.doi.org/10.1109/T-AFFC.2011.6.

Lei Yin, Wentao Cheng, Zhida Qin, Tianyu Huang, Yidong Li, and Gangyi Ding. AutoUE: Automated Generation of 3D Games in Unreal Engine via Multi-Agent Systems, 2026. URL https://arxiv.org/abs/2603.07106.

Alex L. Zhang, Thomas L. Griffiths, Karthik R. Narasimhan, and Ofir Press. Videogamebench: Can vision-language models complete popular video games?, 2026a. URL https://arxiv.org/abs/2505.18134.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents, 2026b. URL https://arxiv.org/abs/2505.22954.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Xiuhui Zhang, Yi Chen, Shusheng Xu, Fan Li, Huan Wang, Tongkai Yang, and Binhang Yuan. GameASG-Bench: Benchmarking autonomous software generation for game development, 2026c. URL https://arxiv.org/abs/2609.21293.