GUI Agents for Continual Game Generation

Anonymous ACL submission

Abstract

Generating a game is not the same as making one that can be played. Despite advances in code generation, existing approaches treat game generation as one-shot translation from prompt to artifact, leaving interaction-level failures undetected. We argue that evaluating and improving game generation requires a player, and study two roles for GUI agents in this process: (1) as an objective evaluator, for which we introduce PlaytestArena, a new evaluation environment that pairs 200 browser-based game generation tasks across eight genres with rubrics of expected in-play behaviors, adjudicated by a GUI agent that loads each build in a browser and plays it; and (2) as a subjective playtester, for which we propose Play2Code, where a game agent and a GUI agent operate in a sustained loop with shared memory, turning game generation into a dialogue between coding and playing. Our experiments show that even frontier models struggle to generate playable games directly, while Play2Code achieves a 66.8% rubric pass-rate, improving over single-pass and agentic-coding baselines by 37.1 and 14.6 points respectively. Further analysis shows that GUI playtester feedback is more traceable than a human report, yet idiosyncratic in ways reminiscent of human testers, establishing game playtesting as a critical testbed for interactive code generation.

1 Introduction

"A game is a series of interesting choices."

— Sid Meier

A musical score, by itself, is silent. Every note in place, every dynamic marked, every tempo specified. Yet whether the music is moving, dull, or merely correct cannot be known until someone plays it. A game, like a score, must be played. It can compile, run, and pass every test, yet be broken in ways no static analysis can reveal. This is why the game industry has long relied on human playtesters, people who sit down with a build, play for hours, and report back what no inspection can see, such as a button press that gets no response, a sprite that fails to render, or a win condition that never triggers.

Recent LLM-based code generators have begun to produce increasingly capable programs from natural-language prompts (Zheng et al., 2023; Zhu et al., 2024; Dong et al., 2025), including interactive web game artifacts (Hu et al., 2024; Gallotta et al., 2024; Sweetser, 2024). Yet this progress has outpaced playtesting itself: generators produce candidate games faster than any human can play them, while what makes a game good remains as inaccessible to inspection as it ever was. In most code generation tasks, quality can be checked against tests or specifications. In game generation, it cannot. Under this constraint, how to evaluate and improve game generation remains an open question (Hu et al., 2024; Mu et al., 2026).

We argue that evaluating and refining game generation requires a player. Among the candidate surrogates one might imagine, only GUI agents meet the game on the same presentational surface a human playtester does: they perceive the rendered interface, decide on actions grounded in that perception, and act through the same input channels a human would. We therefore study two roles for GUI agents in game generation.

First, as an objective evaluator, we introduce PlaytestArena, a new environment for studying game generation when quality lives in play rather than code (§3). Our arena consists of a curated set of generation tasks, each pairing a natural-language prompt with a rubric of expected in-play behaviors that defines what the game should look like when correctly implemented. A GUI agent serves as the evaluator: it loads each generated game in a browser, plays it through clicks and keys, and adjudicates each rubric criterion based on what

Figure 1: Three paradigms for AI-driven game generation. Unlike prior one-shot or runtime-probing pipelines, Play2Code makes generation a sustained dialogue between coding and playing.

it observes during play. The tasks span common game genres, including puzzle, strategy, and card games, each challenging generators with different demands on stateful gameplay, input handling, and visual feedback. PlaytestArena is useful for studying how generation methods perform when the unit of evaluation shifts from code to play.

Second, as a **subjective playtester**, we introduce **Play2Code**, a system that bridges game generation and playtesting, and sustains their interaction to refine the game (§4). As specified in Figure 1, prior work treats game generation as end-to-end synthesis, where a single prompt yields a single artifact (Hu et al., 2024; Jiang et al., 2026b). Play2Code treats it as continual evolution, with each build shaped by what the previous one was like to play (Wang et al., 2025b; Huang et al., 2026a). A game agent and a GUI agent operate in a loop around a shared runtime and a shared memory. The GUI agent surfaces observations from each build, and the game agent uses them to refine the next. Generation becomes not a one-shot translation but a sustained dialogue between coding and playing (Shinn et al., 2023; Wang et al., 2023; Tao et al., 2024; Gao et al., 2025).

In this paper, we study the following research questions:

RQ1: How does the nature of games reshape what counts as "better"?

RQ2: What must a "player" look like, and where should it sit?

RQ3: How does such a "player" drive game generation, and does it drive it where a human would?

To study these questions, we make three complementary contributions. **First**, we build PlaytestArena, a testbed of 200 browser-based game generation tasks spanning eight genres, each paired with a rubric of expected in-play behaviors and adjudicated by a GUI agent that loads the build in a browser and plays it. **On top of this**, we develop Play2Code, in which a game agent and a GUI agent operate in a sustained loop with shared memory, letting experience accumulate across rounds and tasks. Across three frontier backbones, Play2Code achieves a 66.8% rubric pass-rate, improving over single-pass and agentic-coding baselines by 37.1 and 14.6 points respectively, with rubric scores rising monotonically across rounds. **Lastly**, we take a closer look at what the GUI playtester contributes and find that its feedback is traceable in ways a human report is not, and idiosyncratic in ways reminiscent of human testers themselves.

2 Can GUI Agents Playtest?

Consider how a human playtester works. They open the build, sit at the screen, and engage with the game in real time. Their access is mediated entirely by its presentational surface. What they produce is a record of what the game was like to play, including which mechanics worked, where the experience broke down, and what felt missing.

This is also where GUI agents live. A GUI agent perceives the rendered interface, decides on actions grounded in that perception, and acts through the same input channels a human would: clicks, key presses, scrolls. Among the candidate surrogates one might imagine, code-reading critics (Yadavally et al., 2025; Sistla and Kleiman-Weiner, 2026), scripted API drivers (Shin et al., 2020; Balla et al., 2024), or GUI agents themselves (Wang et al., 2024; Ouyang et al., 2026), only the last meets the

Figure 2: An example GUI agent session playing Snake. At each grid step, the agent observes the current screen, reasons about the next move, and issues a keyboard input. Across eight steps, the agent tracks snake and apple positions, updating its plan after each state transition.

game on the surface where playtesting takes place. We therefore take GUI agents as the natural surrogate for human playtesters.

Surface compatibility, however, is only a necessary condition. Whether GUI agents can actually playtest remains to be shown. We verify this directly by assembling a small testbed of 20 browser-based games, curated from both LLM-generated games and public web games, spanning puzzle, strategy, card, platformer, and management games. Because games vary widely in length and structure, each game is decomposed into several discrete levels, yielding approximately 120 levels in total. An agent is considered to have passed a level if it satisfies the level's completion condition within an episode. We connect three GUI agent backbones (GPT-5.4 (OpenAI, 2025), Claude Sonnet 4.6 (Anthropic, 2026), and Kimi K2.5 (Moonshot AI, 2026)) to each game and let them play independently. As a reference point, we recruit three human players to play the same levels under matched conditions. See more details in Appendix §A.

Table 1 reports pass@k for each backbone and the human reference at $k \in \{5, 10, 20\}$. All three GUI agents clear the majority of levels, with GPT-5.4 approaching the human reference. Beyond the aggregate numbers, GUI agents exhibit the three capabilities a playtester needs: (1) **They observe**: their state descriptions reflect the rendered game screen, including object locations, interface changes, and failure states. (2) **They reason**: their intermediate decisions connect the current observation to the next intended goal, such as approaching a target, avoiding an obstacle, or retrying after a failed interaction. (3) **They act**: their keyboard and mouse inputs follow from this reasoning and produce observable state transitions in the game. Figure 2 walks through one such session, showing the agent's observations, reasoning, and actions over a playing trajectory. This illustrates that the agent is not merely judging a static artifact, but genuinely interacting with the game and, when needed, producing signals for evaluation and refinement based on the trajectory.

Table 1: Pass@k for GUI agent backbones and the human reference.

Agent	pass@5	pass@10	pass@20
GPT-5.4	0.67	0.75	0.82
Sonnet 4.6	0.63	0.76	0.79
Kimi K2.5	0.52	0.68	0.72
Human Reference	0.83	0.92	0.92

3 What's "Better": PlaytestArena

Having established that GUI agents can carry out playtesting (§2), we now build the environment where they can judge game generation quality. Existing benchmarks for AI-generated

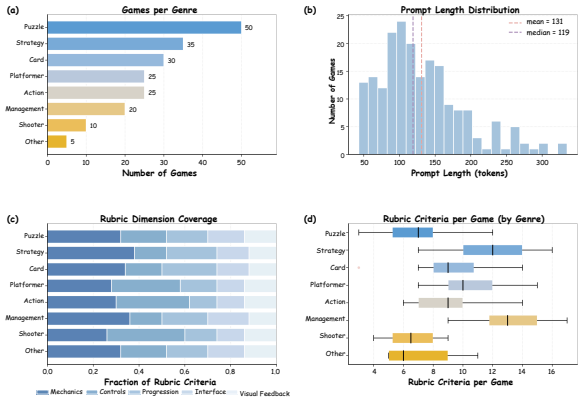

Figure 3: Statistics of PlaytestArena: (a) distribution of 200 games across eight genres, (b) prompt length distribution, (c) per-genre coverage of five rubric dimensions, and (d) rubric criterion count per game by genre.

games remain scarce, and the few that exist (e.g., OpenGame-Bench (Jiang et al., 2026b) and WebGameBench (Zhang et al., 2026)) are not publicly released, leaving the community without a shared substrate for comparison. PlaytestArena fills this gap. Games are instantiated as self-contained HTML/CSS/JS bundles that run in any modern browser, placing game playtesting in the same substrate as the web GUIs that contemporary GUI agents already operate on. PlaytestArena requires two components: a set of target games, each paired with a generation prompt and a rubric defining what correct play should look like; and a playtester that inspects whether a generated game satisfies its rubric by playing it. The first defines what "better" means; the second turns that definition into a score.

Game Prompts Construction. Each entry in PlaytestArena consists of a generation prompt and a corresponding rubric. Game topics are collected from common game genres (Wikipedia contributors, 2026) and public web-game platforms (itch.io, 2026; Poki, 2026), covering puzzle games, strategy games, card games, and others (Figure 3a), totaling 200. For each topic, we recruited human experts, as described in Appendix §B, to author prompts sketching the intended game, specifying stateful gameplay, win and loss conditions, player input, and feedback behavior. Prompts have a mean length of 131 tokens (median 119; Figure 3b), concise enough to remain self-contained while specifying gameplay loops, win/loss conditions, and feedback behavior. Vague or non-self-contained prompts are removed in a final manual pass.

Rubrics Construction. To build rubrics paired with prompts, we follow recent rubric-based evaluation practice (Arora et al., 2025; Li et al., 2026): human experts write criteria covering the intended mechanics, controls, progression, and visual feedback; they then refine these into testable items, each stating an expected behavior such as "the spawned enemies move toward the player's position". A final human pass ensures every item is observable and faithful to the prompt. The resulting rubric corpus contains 1,548 criteria across the 200 games, with a mean of 7.7 criteria per game; coverage across the five dimensions is balanced across all eight genres, with Mechanics and Controls accounting for the largest share, consistent with the prompt-level emphasis on stateful gameplay (Figure 3c,d). The result is a per-game specification of what playing the generated artifact should reveal, against which any generation attempt can be checked.

Evaluation. Given a generation prompt, a generated game, and the game's rubric, the GUI agent loads the game in a browser and plays it, observing the rendered interface, taking actions through clicks and keys, and tracking which rubric criteria its observations support, aligning with human playtesters. For each criterion, the agent issues a pass-fail judgment grounded in the gameplay it has experienced, paired with the expected behavior that defines what evidence justifies a pass. The final score for a game is the fraction of its rubric criteria that the GUI playtester adjudicates as passed.

Is the GUI Evaluator Reliable? Because every PlaytestArena score is adjudicated by a GUI agent, the benchmark inherits whatever bias that judge carries. To validate it, we collect per-criterion judgments from blind human annotators on a 32-game stratified sample spanning all eight genres and the full quality range of the three methods in §5. At the per-criterion level, the GUI judge matches humans on 84.2% of items ($\kappa = 0.64$), within the human–human band ($\kappa = 0.66$). At the game level, GUI- and human-derived scores rank the 32 games near-identically (Spearman's $\rho = 0.87$, Pearson's $r = 0.88$), confirming that method comparisons in §5 are robust to the evaluator's residual noise. Full setup, per-pair breakdowns, and disagreement case studies are in Appendix C.

4 Play2Code

So far we've positioned the GUI agent as an evaluator (§3). But evaluation is only one of the things a playtester does. Here, we repurpose the GUI

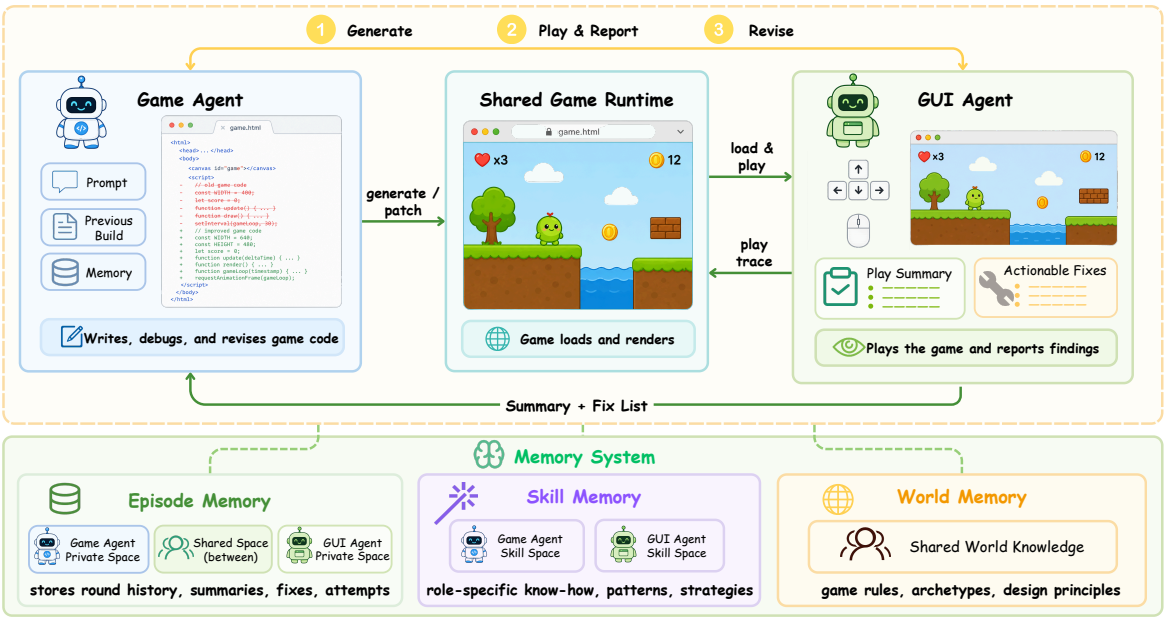

Figure 4: Overview of Play2Code. A game agent and GUI agent operate in a loop: the game agent generates or patches an HTML game, the GUI agent plays the build and produces a play summary and fix list, which guide the next revision. A three-layer memory system (episode, skill, and world memory) accumulates experience across rounds and tasks, turning GUI playtesting into a continual refinement signal.

Metric	GUI–Human	Human–Human
Raw agreement (%)	84.2	90.7
Cohen's κ	0.64	0.66
Spearman's ρ	0.87	0.91
Pearson's r	0.88	0.91

Table 2: Agreement between GUI and human evaluators on 32 PlaytestArena games, averaged over three pairs for both GUI–Human and Human–Human conditions.

agent for this second role, as a source of refinement signal that drives game generation itself.

To put the GUI agent in this role, we introduce **Play2Code**, a system that bridges game generation and playtesting and sustains their interaction across rounds (Figure 4). At its core, Play2Code couples two agents around a shared game runtime. A *Game Agent* writes, debugs, and patches game code. A *GUI Agent* plays the game build and reports what it experienced back to the game agent through shared memory. Each generation task unfolds as a sequence of rounds, each consisting of a *generate or revise* step by the game agent and a *play-and-report* step by the GUI agent.

The following section focuses on the conceptual description of the essential components in Play2Code. Further implementation details, including the workflows of the game agent and GUI agent and the management of the memory system, are provided in Appendix §D.

Game Agent. The game agent produces or revises code conditioned on the most recent build, the episode memory, and the cross-task skill and world memories. The output is written into a shared runtime, packaged as a self-contained HTML-based artifact that the GUI agent can immediately load.

GUI Agent. The GUI agent loads the runtime in a browser and plays through it without access to the code or evaluation rubric, relying only on memory and a game guide. Once the GUI agent decides that the episode should end, it compiles its action and observation log into two artifacts. The first is a summary describing events and interaction behaviors during play. The second is a list of actionable fixes that map observed failures to concrete code-level changes, for example, enemies remain stationary when they should patrol → verify enemy patrol trigger conditions. Both are written into the shared subspace of the episode memory.

Between Rounds. The game agent reads the summary and the fix list and decides, on its own, which fixes are worth acting on and which to defer or discard, treating the GUI agent's report as advice rather than instruction. A new round begins with the game agent revising the build accordingly. The loop terminates when the GUI agent judges that the build meets the intent of the prompt, or when a maximum number of rounds is reached.

	Direct LLM			OpenGame			Play2Code (Ours)		
Genre	GPT-5.4	Sonnet 4.6	Kimi K2.5	GPT-5.4	Sonnet 4.6	Kimi K2.5	GPT-5.4	Sonnet 4.6	Kimi K2.5
Puzzle	35.2	33.4	31.8	60.5	59.7	50.2	77.8	74.6	52.8
Strategy	30.4	28.6	26.9	54.3	55.1	44.1	69.5	70.2	52.0
Card	36.8	34.7	32.5	61.2	60.6	51.4	76.8	75.9	67.3
Action	28.3	26.5	24.8	51.7	51.2	41.3	69.4	67.8	58.6
Platformer	31.5	29.8	27.4	55.6	55.1	45.2	74.7	70.9	52.4
Management	33.6	31.5	29.7	58.4	57.9	48.5	73.2	73.7	54.5
Shooter	27.4	25.6	23.9	50.8	50.3	40.5	67.9	66.8	57.4
Other	29.5	27.7	25.6	53.2	52.6	42.7	69.3	68.5	59.8
Avg.	31.6	29.7	27.8	55.7	55.3	45.5	72.3	71.1	56.9

Table 3: Per-genre rubric scores on PlaytestArena across three methods and three backbones. Each cell reports the rubric pass-rate (%). Cells shaded from yellow to grey mark the best, second-best, and third-best score per row.

Memory System. Across many generation tasks, the same kinds of failures recur, such as incorrect jump arcs in platformers, yet each is often rediscovered from scratch. To preserve and reuse this experience, Play2Code organizes memory along two dimensions, *lifecycle* (in-task or cross-task) and *scope* (private, shared, or common), yielding three layers. **Episode Memory** is in-task, accumulating summaries, fixes, and attempts across rounds in a shared subspace between agents. **Skill Memory** is cross-task and agent-isolated, holding each agent's accumulated know-how, recurring code patterns for the game agent and interaction strategies for the GUI agent. **World Memory** is cross-task and shared across all agents, recording general game rules, common archetypes, and universal design principles. Together, these layers turn Play2Code into a system whose competence grows with experience.

5 Does Play Drive Code Generation?

To test whether positioning a GUI agent inside the generation process in Play2Code matters, we compare Play2Code against two baselines representing prior paradigms for AI-driven game generation (Figure 1). Direct LLM prompts the model to emit the entire game in a single pass. OpenGame runs an end-to-end agentic pipeline that iteratively builds, runs, and inspects the game at syntactic and compiling level. Play2Code adds the missing component: a GUI agent that plays each generated build and feeds prescriptive feedback back into the next round of generation. We instantiate each method with three backbone models used as both game agent and GUI agent: Claude Sonnet 4.6, GPT-5.4, and Kimi K2.5.

Playtesting exposes what code-level signals miss. Table 3 reports two comparisons that together isolate the value of playtesting. Across three backbones, Play2Code improves over Direct LLM from 29.7 to 66.8, reflecting the cost of committing to an artifact without ever observing play: any mismatch between intended and realized behavior goes uncorrected. The more revealing comparison is with OpenGame (52.2 → 66.8), an agentic system that already builds, runs, and inspects games iteratively, but does not play them. That Play2Code consistently outperforms OpenGame isolates the playtesting signal itself: failures such as unresponsive controls, missing state transitions, or win conditions that never trigger are invisible to code- and compilation-level inspection, and only surface during play. Direct ablations (Appendix F) confirm that both the GUI Agent and the layered memory system are essential components of Play2Code.

Continual playtesting drives round-wise game evolution. We further examine how Play2Code improves within a single evolving trajectory. As shown in Figure 6a, for each generated game, we record the rubric score after each play–code round. The scores increase monotonically from Round 1 to Round 3, showing that GUI feedback is actionable: failures observed are translated into concrete fixes that improve the next build. Early rounds often expose basic interaction failures, such as missing responses to player input, while later rounds address finer-grained issues like visual effects. This round-wise improvement suggests that playtesting provides a usable optimization signal for code generation. In this sense, Play2Code turns gameplay observations into an iterative refinement process.

Game complexity modulates the effectiveness of GUI-driven refinement. Figure 6a reveals substantial variance in Play2Code's gains across complexity tiers (see Appendix G for tier definitions).

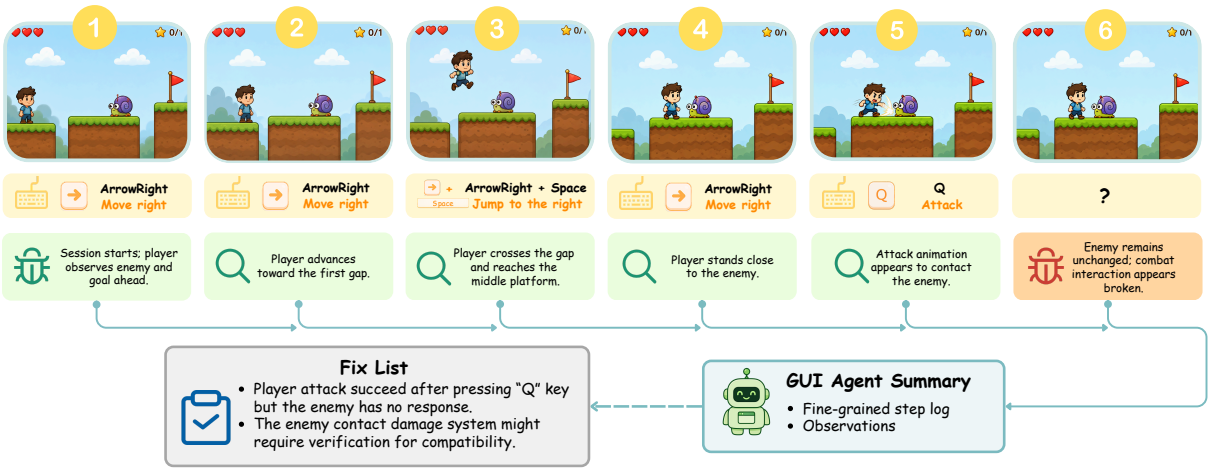

Figure 5: Example playtest trace produced by the GUI Agent. Per-step inputs and observations are distilled into a summary and an actionable fix list that the Game Agent consumes in the next revision round.

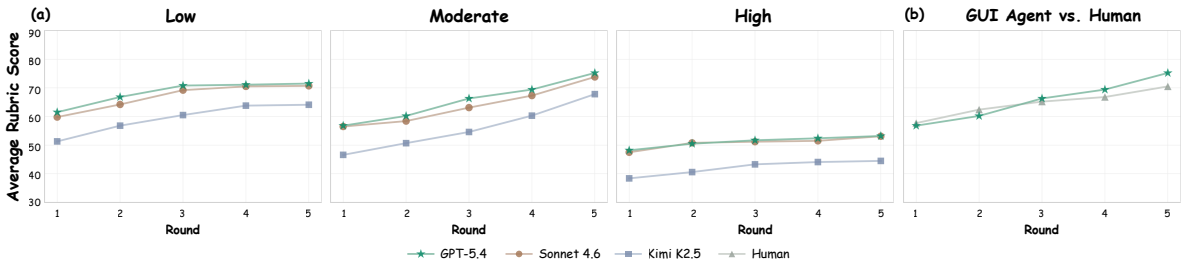

Figure 6: Round-wise evolution of average rubric scores. (a) Games are partitioned into Low-, Moderate-, and High-complexity tiers based on score trajectories under our Play2Code protocol. (b) GUI-agent vs. human-driven refinement on Moderate-complexity games. See Appendix §H for more experimental details.

On *Low-complexity* games, the GUI agent surfaces nearly all latent defects within the first few rounds, with scores converging by rounds 3–5. *Moderate-complexity* games benefit most from the iterative loop, exhibiting steady, monotonic improvement across all five rounds and the largest absolute gain end-to-end. *High-complexity* games, by contrast, see markedly limited improvement: scores plateau early and remain low across all three code models, as intricate mechanics are difficult for the GUI agent to trigger reliably, leaving little actionable signal for refinement. This suggests the bottleneck on harder games lies in GUI-agent capability rather than the refinement loop itself, pointing to stronger GUI agents as a direct lever for further gains.

6 How Far Are GUI Agents from Human Playtesters?

Human playtesters remain irreplaceable for what they uniquely provide: a felt sense of difficulty, frustration, boredom, and surprise, alongside the tacit aesthetic judgments that no agent currently approximates. The question we ask here is therefore how far GUI agents are from filling that role: what they can deliver even when they fall short of a full human playtester, and where their behavior mirrors the idiosyncrasies of human playtesters themselves.

GUI agents produce more traceable feedback than humans. A human playtester, even an attentive one, produces a report that is fundamentally a summary: salient moments remembered, causal chains reconstructed under uncertainty, and the raw trace of what actually happened, every key press, every observation that preceded a failure, lost by the time the report reaches the developer. The GUI agent's feedback is structured differently. Every action, observation, and diagnosis is logged (Figure 5), so when the agent reports that "enemies remain stationary when they should patrol," the failure is anchored in a recoverable sequence of frames rather than a playtester's reconstruction. This traceability translates into measurable gains: on games well-suited to GUI play, the agent approaches human playtesters in driving refinement (Figure 6b), suggesting that GUI-driven feedback can serve as a scalable and competitive refinement signal.

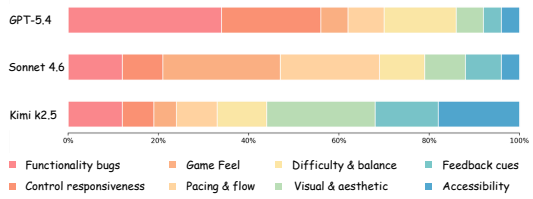

Figure 7: Distribution of feedback categories across playtested games for three backbones.

Different agents notice different things. Different GUI agents playing the same game produce feedback with different emphases, much as different human playtesters do. Figure 7 shows this divergence across three playtester backbones during game generation. GPT-5.4 concentrates the bulk of its feedback on functionality bugs and control responsiveness, flagging broken mechanics, missing state transitions, and unresponsive controls. Sonnet 4.6 spreads its attention more evenly, devoting a comparable share to experiential issues such as pacing, game feel, and difficulty balance. Kimi k2.5 is the most striking outlier: nearly half its feedback concerns visual and aesthetic dimensions, where the others barely linger. We do not read this divergence as a failure of consistency. It suggests that GUI agents, mechanical though at the level of execution, carry something like a *taste* at the level of what they notice, and that taste, like a human playtester's, depends on who is doing the playing.

7 Related Work

7.1 Game Generation and Evaluation

Large language models have shown strong code-generation capabilities (Zhu et al., 2024; Hui et al., 2024; Dong et al., 2025), including synthesizing complete games (Hu et al., 2024; Gallotta et al., 2024). Existing evaluations, however, rely on functional correctness (Zheng et al., 2023; Zhang et al., 2025a), which cannot capture whether a game is playable, responsive, or faithful to its intended interaction design (Peng et al., 2026). Because game quality is grounded in real-time visual interaction, evaluating and improving games requires feedback derived from gameplay trajectories (Wen et al., 2025). We adopt this interaction-centered view, with the GUI agent serving as a playtester whose observations drive subsequent revisions.

7.2 GUI Agents

GUI agents, typically built on vision-language models (VLMs), have demonstrated strong visual understanding and action planning (Hong et al., 2024; You et al., 2024; Qin et al., 2025; Wang et al., 2025a; Huang et al., 2026b), interpreting screen-based observations and taking actions conditioned on the environment state and task goals (Deng et al., 2023; Wu et al., 2024; Xie et al., 2024; Liu et al., 2024; Nguyen et al., 2025). In game-playing settings, they further show the ability to interact with dynamic environments to achieve task goals. (Ouyang et al., 2026; Plaat et al., 2024). These properties make GUI agents a natural candidate for scalable proxies of human feedback in real-time environments.

7.3 Memory-Augmented Agent Evolution

Agent evolution enables continual improvement through iterative feedback loops over execution, tool use, and multi-agent interaction (Tao et al., 2024; Gao et al., 2025; Wang et al., 2025c; Zhang et al., 2025b; Ma et al., 2024; Wang et al., 2025b; Huang et al., 2026a), though most existing approaches focus on static settings rather than interactive environments (Dong et al., 2026; Wang et al., 2023). In parallel, agent memory retains interaction histories, preferences, and task-specific knowledge for reuse across various tasks (Park et al., 2023; Shinn et al., 2023; Zhong et al., 2024; Chhikara et al., 2025; Yu et al., 2026), and can further support reusable skills that transfer successful strategies to new tasks (Jiang et al., 2026a; Ma et al., 2026). Our setting connects these two directions: a coding system and a GUI agent co-evolve with a shared memory pool, so that interaction history and skills accumulate across tasks on both sides.

8 Conclusions

We studied game generation under the constraint that quality lives in play, not in code. PlaytestArena scores generated games via a GUI agent that loads each build, plays it, and adjudicates per-game rubrics from observation. Play2Code then places this GUI agent inside the generation loop, where a game agent and a GUI agent co-evolve through shared memory across games. Across three backbones, Play2Code outperforms single-shot and agentic-coding baselines on PlaytestArena, with rubric scores rising monotonically within each refinement trajectory, evidence that playtesting yields not just a richer evaluation signal but also a usable optimization signal for code generation.

Limitations

We note several limitations of this work. First, our current implementation focuses on HTML-based games; extending the framework to broader game formats, such as native-engine or 3D titles, is left to future work. Second, our benchmark covers a curated set of games selected for quality control, and does not exhaust the full diversity of real-world game scenarios. Third, the memory schema is shared across all games in our current design, while game-specific or genre-aware memory structures may yield further gains.

References

Anthropic. 2026. Claude sonnet 4.6. Large language model. Accessed: 2026-05-22.

Rahul K Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, and 1 others. 2025. Healthbench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*.

Martin Balla, George EM Long, James Goodman, Raluca D Gaina, and Diego Perez-Liebana. 2024. Pytag: Tabletop games for multiagent reinforcement learning. *IEEE Transactions on Games*, 16(4):993–1002.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*.

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

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*.

Zihan Dong, Zirou Liu, Zekun Wang, Yunqing Li, and Zhiyuan Ma. 2026. The evaluation challenge of agency: Reliability, contamination, and evolution in llm agents.

Roberto Gallotta, Graham Todd, Marvin Zammit, Sam Earle, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. 2024. Large language models and games: A survey and roadmap. *IEEE Transactions on Games*.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 1.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and 1 others. 2024. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14281–14290.

Chengpeng Hu, Yunlong Zhao, and Jialin Liu. 2024. Game generation via large language models. In *2024 IEEE Conference on Games (CoG)*, pages 1–4. IEEE.

Yixu Huang, Xinglei Yu, and Zhongyu Wei. 2026a. Ace: Self-evolving llm coding framework via adversarial unit test generation and preference optimization. *arXiv preprint arXiv:2605.16299*.

Yixu Huang, Tinghui Zhu, and Muhao Chen. 2026b. Learning adaptive reasoning paths for efficient visual reasoning. *arXiv preprint arXiv:2604.14568*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

itch.io. 2026. itch.io – download the latest indie games.

Guanyu Jiang, Zhaochen Su, Xiaoye Qu, and Yi R Fung. 2026a. Xskill: Continual learning from experience and skills in multimodal agents. *arXiv preprint arXiv:2603.12056*.

Yilei Jiang, Jinyuan Hu, Qianyin Xiao, Yaozhi Zheng, Ruize Ma, Kaituo Feng, Jiaming Han, Tianshuo Peng, Kaixuan Fan, Manyuan Zhang, and 1 others. 2026b. Opengame: Open agentic coding for games. *arXiv preprint arXiv:2604.18394*.

Bo Li, Yida Yin, Wenhao Chai, Xingyu Fu, and Zhuang Liu. 2026. Ueval: A benchmark for unified multimodal generation. *arXiv preprint arXiv:2601.22155*.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, and 1 others. 2024. Autoglm: Autonomous foundation agents for guis. *arXiv preprint arXiv:2411.00820*.

Hao Ma, Tianyi Hu, Zhiqiang Pu, Boyin Liu, Xiaolin Ai, Yanyan Liang, and Min Chen. 2024. Coevolving with the other you: Fine-tuning llm with sequential cooperative multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 37:15497–15525.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. 2026. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv:2604.08377*.

Moonshot AI. 2026. Kimi K2.5: Visual agentic intelligence. Accessed: 2026-05-22.

Enhong Mu, Jinyu Cai, Yijun Lu, Mingyue Zhang, Kenji Tei, and Jialong Li. 2026. Knowledge graph-enhanced large language model for incremental game playtesting. *IEICE Transactions on Information and Systems*, page 2025KBP0004.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, and 1 others. 2025. Gui agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22522–22538.

OpenAI. 2025. Introducing GPT-5. Blog post, August 7, 2025.

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

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Zhiyuan Peng, Wei Tao, Xin Yin, Chenhao Ying, Yuan Luo, and Yiwen Guo. 2026. Playcoder: Making llm-generated gui code playable. *arXiv preprint arXiv:2604.19742*.

Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki Van Stein, and Thomas Back. 2024. Reasoning with large language models, a survey. *arXiv preprint arXiv:2407.11511*, 1(2):3.

Poki. 2026. Poki – free online games.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, and 1 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*.

Yuchul Shin, Jaewon Kim, Kyohoon Jin, and Young Bin Kim. 2020. Playtesting in match 3 game using strategic plays via reinforcement learning. *IEEE Access*, 8:51593–51600.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Swadesh Sistla and Max Kleiman-Weiner. 2026. Evaluating llms in open-source games. *Advances in Neural Information Processing Systems*, 38:104032–104063.

Penny Sweetser. 2024. Large language models and video games: A preliminary scoping review. In *Proceedings of the 6th ACM Conference on Conversational User Interfaces*, pages 1–8.

Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. 2024. A survey on self-evolution of large language models. *arXiv preprint arXiv:2404.14387*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, and 1 others. 2025a. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*.

Shuai Wang, Weiwen Liu, Jingxuan Chen, Yuqi Zhou, Weinan Gan, Xingshan Zeng, Yuhan Che, Shuai Yu, Xinlong Hao, Kun Shao, and 1 others. 2024. Gui agents with foundation models: A comprehensive survey. *arXiv preprint arXiv:2411.04890*.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025b. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. 2025c. Mobile-agent-e: Self-evolving mobile assistant for complex tasks. *arXiv preprint arXiv:2501.11733*.

Yule Wen, Yixin Ye, Yanzhe Zhang, Diyi Yang, and Hao Zhu. 2025. Real-time reasoning agents in evolving environments. *arXiv preprint arXiv:2511.04898*.

Wikipedia contributors. 2026. Video game genres.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Aashish Yadavally, Hoan Nguyen, Laurent Callot, and Gauthier Guinet. 2025. Large language model critics for execution-free evaluation of code changes. *arXiv preprint arXiv:2501.16655*.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pages 240–255. Springer.

Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. 2026. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885*.

Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Changzhi Zhou, Ken Deng, Dengpeng Wu, Guanhua Huang, Kejiao Li, and 1 others. 2025a. Artifactsbench: Bridging the visual-interactive gap in llm code generation evaluation. *arXiv preprint arXiv:2507.04952*.

Ding-Chu Zhang, Yida Zhao, Jialong Wu, Liwen Zhang, Baixuan Li, Wenbiao Yin, Yong Jiang, Yu-Feng Li, Kewei Tu, Pengjun Xie, and 1 others. 2025b. Evolvesearch: An iterative self-evolving search agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 13134–13147.

Wenyu Zhang, Guoliang You, Haotian Zhao, Tianshu Zhu, Haoran Wang, Xiaoxuan Tang, Mingyang Dai, Jingnan Gu, Daxiang Dong, Jianmin Wu, and 1 others. 2026. Webgamebench: Requirement-to-application evaluation for coding agents via browser-native games. *arXiv preprint arXiv:2605.17637*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*, pages 5673–5684.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Appendix

A GUI Playtester Feasibility Study: Details

This appendix expands on the small-scale feasibility study summarized in §2, which establishes that GUI agents are operationally capable of playing games before we deploy them as playtesters in PlaytestArena (§3) and as refinement drivers in Play2Code (§4). We describe the testbed construction, level decomposition, agent and human protocols, and per-game results.

A.1 Testbed Construction

The feasibility testbed contains 20 games, intentionally kept small so that the same levels can be played end-to-end by both GUI agents and human references under matched conditions. Games are drawn from two sources:

- **LLM-assisted specifications (10 games):** Prompts authored by the same expert pool described in Appendix §B, but disjoint from the 200 PlaytestArena tasks. Each prompt is realized as a self-contained HTML/CSS/JS bundle by a frontier code model and lightly debugged by a human until it runs without crashes. These games are kept simple by design, since the purpose is to test whether agents can play, not whether they can play hard games.
- **Public web games (10 games):** Lightweight browser titles drawn from `itch.io` (itch.io, 2026) and `Poki` (Poki, 2026), selected to span the five genres listed below. We use only games that run as static HTML bundles with no login, payment, or external API dependency.

The 20 games cover five genres: puzzle (5), strategy (4), card (4), platformer (4), and management (3). Compared to PlaytestArena, this testbed omits action and shooter genres because their fast-paced real-time dynamics introduce additional confounds (agent inference latency vs. game tick rate) that are orthogonal to the question of whether GUI agents can play at all.

A.2 Level Decomposition

Games vary substantially in length: a Sokoban-style puzzle may admit a clean per-puzzle decomposition, while a tower defense or management game runs as a single long session. To make *pass* a meaningful unit of measurement, each game is decomposed into discrete **levels**, each with a single, unambiguous completion condition (e.g., "reach the goal tile", "clear all blocks", "survive wave 5", "reduce opponent HP to zero"). Decomposition is performed by the same expert who authored or selected the game, and reviewed by a second expert for unambiguity. Across the 20 games, this yields 118 levels in total (mean 5.9 per game, min 3, max 10), which we round to approximately 120 in the main text.

A level is counted as **passed** if its completion condition is satisfied within a single episode, where an episode terminates on (i) completion, (ii) an explicit game-over state, or (iii) a per-level timeout (5 minutes wall-clock for GUI agents, matched for humans). pass@k is computed as the fraction of levels for which at least one of k independent episodes satisfies this condition.

A.3 Agent Protocol

We connect three GUI agent backbones to each game: GPT-5.4, Claude Sonnet 4.6, and Kimi K2.5. Each backbone is run within the same GUI-agent scaffold used later by the GUI Playtester in Play2Code, with three differences appropriate to a feasibility study:

- **No rubric, no fix list.** The agent is given only the game guide (controls, objective, success condition) and the rendered screen. It is not asked to produce a fix list or pass–fail rubric judgments; its sole task is to attempt the level.
- **No memory.** Each level is played from a clean slate, so pass@k reflects raw playing capability rather than the benefit of accumulated experience.
- **Fixed observation rate.** Screenshots are captured at 1280×720 resolution. The agent decides when to issue the next action; we do not impose a fixed tick.

For each (backbone, level) pair we run 20 independent episodes, allowing us to report pass@5, pass@10, and pass@20 from the same set of rollouts.

A.4 Human Reference Protocol

We recruit three human players from the same pool as the expert annotators (Appendix §B), with no

overlap with the prompt/rubric authors or the GUI judge validation annotators. Each human is given the same game guide provided to the GUI agents and plays each level under matched conditions: the same browser, the same 1280×720 viewport, and the same 5-minute per-level timeout. Each human plays each level up to 20 times if they have not yet passed it, allowing a directly comparable pass@k to be computed. Humans are compensated at the same hourly rate as the expert annotators.

To prevent fatigue effects, no human plays more than 30 levels in a single session, and sessions are capped at 90 minutes with a mandatory 15-minute break.

A.5 Per-Genre Results

Table 4 breaks down pass@10 by genre. All three GUI backbones clear the majority of levels across every genre, with the gap to the human reference narrowest on puzzle and card games (which reward deliberation over reaction time) and widest on platformer games (where precise timing of jumps remains a bottleneck for current GUI agents). This pattern foreshadows the difficulty stratification later observed in Play2Code (Figure 6a): genres in which GUI agents play more reliably also benefit more from GUI-driven refinement.

Genre	GPT-5.4	Sonnet 4.6	Kimi K2.5	Human
Puzzle	0.81	0.83	0.74	0.95
Strategy	0.78	0.79	0.69	0.94
Card	0.84	0.82	0.76	0.96
Platformer	0.64	0.66	0.55	0.87
Management	0.72	0.71	0.62	0.89
Avg.	0.75	0.76	0.68	0.92

Table 4: Pass@10 by genre on the 20-game feasibility testbed. All three GUI agents clear the majority of levels across every genre, with the largest gap to humans on platformer games.

A.6 Qualitative Observations

Beyond aggregate pass rates, we observe three recurring patterns in agent rollouts that are relevant to the playtester role assumed in later sections:

- **Observation grounded in the rendered surface.** Across backbones, the per-step observation logs (e.g., "snake head is just below the apple", Figure 2) consistently reflect what is on screen rather than what the agent expects to be on screen, including in failure states such as "no apple visible". This is a prerequisite for using the same observation logs as evidence for rubric adjudication in §3.
- **Retry behavior under failure.** When an action does not produce the expected effect (e.g., a button click with no visible response), agents typically retry once or twice before exploring an alternative interaction, rather than either giving up immediately or repeating the same action indefinitely. This behavior is what makes the GUI agent's report distinguish between "mechanic missing" and "mechanic present but hard to trigger", a distinction that matters for fix-list generation in Play2Code.
- **Failure modes that explain the gap to humans.** The dominant failure modes are (i) imprecise timing on real-time games (especially platformers) and (ii) misreading small or low-contrast UI elements. They are mainly backbone-level limitations caused by inference latency.

These observations are consistent with the three capabilities identified in §2 (*observe*, *reason*, *act*) and support the use of GUI agents as a playtester surrogate in the rest of the paper.

B PlaytestArena Construction Details

This section documents the human-expert workforce behind PlaytestArena, summarized in §3, including recruitment, qualifications, the annotation protocol for prompts and rubrics, and quality control.

B.1 Expert Recruitment and Qualifications

We recruit five human experts to author the 200 generation prompts and the paired rubrics. Experts were recruited from a pool of graduate students and recent graduates in computer science, human–computer interaction, and game design at our institution. To qualify, each expert was required to meet at least two of the following three criteria:

- self-reported ≥10 hours per week of digital game play over the past year;
- prior coursework in game design, HCI, or interactive systems;
- prior experience implementing a playable game (course project, game jam entry, or hobby project) in any engine or framework.

The final five experts comprise three graduate students and two recent graduates; three have shipped at least one playable game prototype, and all five report regular play across multiple genres. None of the experts are authors of this paper, and none participated in the GUI judge validation study (Appendix C), ensuring those two annotation efforts are independent.

B.2 Genre and Topic Coverage

The 200 game topics are distributed across the eight genres reported in Figure 3. Within each genre, topics are seeded from two sources: (i) representative subgenres listed in Wikipedia contributors (2026), and (ii) titles browsable on the public web-game platforms itch.io and Poki (itch.io, 2026; Poki, 2026). We use these sources only to enumerate genre conventions and topic archetypes; no game code, asset, or rubric is copied from existing titles. Topics are reviewed by the lead authors before being assigned to experts to remove duplicates and ensure each topic is implementable as a self-contained HTML/CSS/JS bundle within a single generation pass by a frontier model.

B.3 Prompt Construction Protocol

Each topic is assigned to one of the five experts as primary author. The expert writes a generation prompt that specifies, at minimum:

- the stateful gameplay loop;
- win and loss conditions;
- the set of valid player inputs (keys, mouse actions);
- expected feedback behavior on input (visual, auditory, or state-change cues);
- any progression structure (levels, waves, rounds).

Each prompt is then reviewed by a second expert who flags ambiguity, missing constraints, or non-self-contained references (e.g., "make it feel like *[existing game]*" without further detail). Prompts flagged in this second pass are revised by the primary author or removed. The final 200 prompts are the result of an initial pool of 247 candidate prompts, of which 47 were discarded.

B.4 Rubric Construction Protocol

Once a prompt is finalized, the same primary expert drafts a paired rubric. Each rubric criterion is written to state a single expected in-play behavior in concrete, observable terms (e.g., “the game loads and renders without errors”, “the spawned enemies move toward the player’s position”). The expert is instructed to cover five dimensions: *mechanics*, *controls*, *progression*, *interface*, and *visual feedback*, with a target of 6–10 criteria per game. The final rubric corpus contains 1,548 criteria across the 200 games, with a mean of 7.7 criteria per game (min 5, max 11).

Each rubric is then reviewed by a second expert against three checks:

- **Concreteness**: every criterion describes a single observable behavior, not a quality judgment (e.g., “the spawn rate increases each wave” is accepted; “the difficulty curve feels right” is not);
- **Observability**: every criterion can in principle be adjudicated by a tester who only sees the rendered game, without access to source code or internal state;
- **Faithfulness**: every criterion follows from the paired prompt and is not introduced from the expert’s personal preferences.

Items failing any check are revised or removed. A final pass by one of the lead authors verifies all 200 prompt–rubric pairs against the same three checks.

B.5 Calibration and Inter-Expert Consistency

Before authoring the main set, all five experts complete a calibration round on three shared topics (one puzzle, one strategy, one platformer). Each expert independently writes a prompt and rubric for these three topics; the five drafts are then reviewed jointly, and disagreements on what counts as a concrete, observable criterion are resolved in discussion. The calibration prompts and rubrics are discarded and are not part of the released benchmark.

As a post-hoc check on consistency, we have each expert re-review a random 10% sample of criteria authored by their peers and mark each as ACCEPT or REVISE. The acceptance rate is 95.3%, with the most common reason for REVISE flags being a criterion that is concrete but underspecified

(e.g., omitting which input triggers a behavior). Flagged criteria were revised before benchmark release.

C GUI Evaluator Validation: Full Setup and Analysis

This appendix documents the validation study summarized in §2. We describe the sampling procedure, annotator protocol, full per-pair agreement breakdowns, and disagreement case studies.

Pair	Raw Agr. (%)	Cohen's κ	Spearman's ρ	Pearson's r
GUI vs. Human				
GUI – h_1	85.6	0.70	0.86	0.88
GUI – h_2	84.1	0.62	0.89	0.87
GUI – h_3	82.9	0.60	0.86	0.89
Average	*84.2*	*0.64*	*0.87*	*0.88*
Human vs. Human				
h_1 – h_2	91.6	0.67	0.93	0.88
h_1 – h_3	89.8	0.64	0.89	0.93
h_2 – h_3	90.7	0.67	0.91	0.94
Average	*85.7*	*0.66*	*0.91*	*0.91*

Table 5: Per-pair agreement between the GUI judge and human annotators, and among human annotators. GUI–human agreement is comparable to human–human agreement across raw agreement, Cohen's κ, Spearman rank correlation, and Pearson correlation.

C.1 Sample Construction

We sample 32 generated games from PlaytestArena using stratified sampling along two axes:

- **Genre:** 4 games per genre across the eight genres in PlaytestArena (Puzzle, Strategy, Card, Action, Platformer, Management, Shooter, Other), ensuring each genre is equally represented.
- **Generation method and quality:** Within each genre, we sample games produced by the three methods compared in §5 (Direct LLM, OpenGame, Play2Code). This prevents the agreement statistics from being inflated by sampling only easy cases (e.g., uniformly high- or uniformly low-scoring builds).

The 32 games yield 375 rubric criteria in total, with a mean of 11.7 criteria per game (min 9, max 15).

C.2 Annotator Protocol

Recruitment. We recruit three annotators in computer science with prior game-playing experience (self-reported $\geq$5 hours per week of digital game play). None of the annotators are authors of this paper or were involved in PlaytestArena's construction.

Blinding. Each annotator is blind to (i) which generation method produced each game, (ii) the GUI judge's per-criterion verdicts, and (iii) the GUI judge's play trace.

Task. For each game, annotators receive the generation prompt and the full rubric, play the build in a browser under a 10-minute time budget matching the GUI agent's session length, then adjudicate each rubric criterion as PASS or FAIL. We disallow an UNCLEAR option to keep the label space aligned with the GUI judge, which is also forced into a binary verdict.

Quality control. Before the main study, each annotator completes a calibration round on 3 games not included in the main sample, with feedback on borderline cases to align their interpretation of rubric items. The calibration data is discarded.

C.3 Agreement Metrics

We report metrics along two axes:

1. **Per-criterion agreement.** For each pair of judges (one GUI–human pair or one human–human pair), we compute (a) *raw agreement* as the percentage of rubric criteria on which the two judges return the same verdict, and (b) *Cohen's κ*, which corrects for chance agreement under the marginal pass-rates of each judge. We then average over the three GUI–human pairs and, separately, over the three human–human pairs.
2. **Per-game ranking consistency.** For each GUI–human or human–human pair, we score every game by the fraction of rubric criteria each judge marks PASS, then compute *Spearman's ρ* between the two resulting rankings of the 32 games. We also report *Pearson's r* on the same scores as a robustness check; the two metrics give consistent conclusions.

C.4 Results

Per-pair breakdown. Table 5 reports the four metrics for each of the six pairs individually. The GUI judge's agreement with each individual human is comparable to the agreement between any two humans, with no human annotator showing a substantially closer or more distant relationship to the GUI judge than to their peers. This argues against the concern that the GUI judge might be tracking the idiosyncrasies of any single annotator's calibration.

Aggregate agreement. Table 2 in the main text summarizes the agreement averaged within each pair type, reproducing the headline numbers from §3 and adding Pearson's r as a robustness check.

D Play2Code System Design Detail

This section documents the concrete instantiation of the Game Agent and the GUI Agent introduced in §4. The two agents share a runtime and a memory system, but operate over disjoint observation spaces and tool inventories, reflecting their respective roles as developer and playtester.

D.1 Game Agent

The Game Agent is responsible for translating the generation prompt into a runnable HTML-based game artifact and for revising that artifact between rounds based on the GUI Agent's report. It operates in a code-first mode, working autonomously through a fixed multi-phase workflow rather than producing the game in a single shot.

Inputs. At the start of each round, the Game Agent receives: (i) the original generation prompt, (ii) the current state of the codebase (or an empty scaffold in the first round), (iii) the GUI Agent's play summary and actionable fix list from the previous round (rounds ≥ 2 only), and (iv) the relevant slices of the shared memory system (§D.3).

Workflow. In the first round, the Game Agent proceeds through five phases after receiving the user prompt: game design, asset generation, code implementation, build-time verification, and memory capture. In subsequent rounds, once the GUI Agent provides a play summary and actionable fix list, the Game Agent skips the game design and asset generation phases by default. Instead, it reviews the GUI feedback, decides which suggested fixes are relevant and feasible, and resumes from code implementation, followed by build-time verification and memory capture.

1. **Game design.** A specifc game design document is generated from the user requirement, according to . This phase also includes game guide (`GAME_GUIDE.md`) generation, which is later consumed by the GUI Agent. (§D.2).

2. **Asset generation.** Sprites, tilemaps, and audio are generated according to game deisign.

3. **Code implementation.** The agent implements scenes, entities and game logics according game design document.

4. **Verification.** The agent runs `npm run build` and a headless test pass, fixing any TypeScript or runtime errors before handing the build to the GUI Agent.

5. **Memory capture.** On completion, the agent writes any non-obvious fix patterns, pitfalls, or design decisions to skill or world memory for use in subsequent tasks.

Tools. The Game Agent has access to file-system tools (`read_file`, `write_file`, `run_shell_command`), project-specific code generation tools (`generate_game_guide`, `generate-game-assets`, and the memory interface (`memory_query`, `memory_save`).

D.2 GUI Agent

The GUI Agent is responsible for playing each candidate build and producing structured feedback that the Game Agent consumes in the next round. It does not read source code or inspect internal game state; its judgment is grounded entirely in the rendered game surface.

Inputs. At the start of each playtest session, the GUI Agent receives: (i) screenshots of the current game frame, captured at 1280×720 resolution; (ii) the `GAME_GUIDE.md` produced by the Game Agent, which specifies the intended controls, objectives, mechanics, and success conditions; (iii) a fixed system prompt defining its role as a playtester, its interaction conventions, and its stopping criteria; and (iv) the relevant GUI-side skill memory and shared world memory (§D.3).

Workflow. The GUI Agent proceeds through five phases: initial observation, game start, interactive playtesting, assessment, and memory capture. Like the Game Agent, it follows an explicit step-wise workflow rather than producing a judgment from a single screenshot.

1. **Initial observation.** The agent observes the rendered game screen and uses short waits when necessary to distinguish loading screens from frozen or unresponsive builds.

2. **Game start.** The agent starts the game by clicking a visible start affordance, pressing

the instructed key, or proceeding directly if the game auto-starts.

3. **Interactive playtesting.** The agent plays toward the stated objective using only mouse and keyboard actions. It verifies the game guide through interaction rather than assuming it is correct. For movement-heavy games, it holds keys long enough to produce visible displacement and combines keys when chorded actions such as run-and-jump are required. The agent does not stop at the first failure state; for platformer-style games, it must attempt at least two materially different retries before declaring the level blocked.

4. **Assessment.** The agent summarizes whether the run was completed, reached an ending, blocked by a bug, or could not start. It evaluates the start flow, controls, core mechanics, progression, and ending, and records severity-tagged findings grounded in the observed trajectory.

5. **Memory capture.** At the end of the session, the agent writes reusable interaction patterns, false-positive cases, and game-archetype-specific testing heuristics to memory for future playtests.

Tools. The GUI Agent has access to browser-level interaction tools: `browser_screenshot`, `browser_click`, `browser_drag`, `browser_key`, `browser_type`, `browser_scroll`, and `browser_wait`, as well as the memory interface `memory_save`. These tools mirror the input channels available to a human playtester. The agent has no access to the DOM, source code, internal variables, or console logs, ensuring that its feedback reflects what playing the game is actually like rather than what code inspection reveals.

Output. At the end of each session, the GUI Agent emits a single Markdown report with a fixed structure: run outcome and confidence, probe signals considered, chronological interaction log, per-dimension gameplay assessment, severity-tagged findings, the most blocking issue, and a recommended fix direction. This report is parsed downstream into the play summary and actionable fix list provided to the Game Agent in the next round.

D.3 Memory System

Both agents read from and write to the shared memory system described in §4. Memory entries are tagged with `layer` (`episode-shared`, `skill`, or `world`), `owner` (`game-agent`, `gui-player`, or `shared`), `kind` (`pitfall`, `fix_pattern`, `decision`, `interaction_pattern`, `false_positive`, `observation`), and `archetype`. Agents query memory at the start of each task scoped by archetype and role, and write new entries selectively at the end of each session, retaining only insights with cross-task reuse value rather than one-off task details.

E Experimental Details

All three methods compared in Table **??** (Direct LLM, OpenGame, and Play2Code) are run on the same set of 200 PlaytestArena tasks, but their effective budgets differ in structure rather than in raw model calls.

Direct LLM. The model emits the entire game in a single pass from the generation prompt, with no runtime inspection and no opportunity for revision. Each task therefore consists of exactly one round.

OpenGame. We follow the released configuration of Jiang et al. (2026b), which iteratively builds, runs, and inspects the game at the syntactic and compilation level. The loop terminates once the game compiles cleanly and hangs normally—i.e., it enters a stable running state without crashing or exiting—or when the maximum of five inspection rounds is reached. We do not modify this stopping criterion to keep the baseline faithful to its original design.

Play2Code. We set the maximum number of evolving rounds to $R_{\max}=5$, matching OpenGame for fairness. However, the loop is not required to run for all five rounds: at the end of each play-and-report step, the GUI Agent decides whether the current build already meets the intent of the generation prompt. Concretely, the GUI Agent is prompted to emit an explicit `Run Outcome` field with values in {`completed`, `reached-ending`, `blocked-by-bug`, `could-not-start`}, together with a confidence level. When the GUI Agent reports `completed` or `reached-ending` with `high` confidence, the loop terminates early and the current build is taken as the final artifact; otherwise, the loop continues until

$R_{\max}$ is reached. Across the 200 tasks, the average effective round count is 3.24 (median 3.05, std 0.72), with 93.5% of tasks terminating before $R_{\max}$.

F Ablation Studies of Play2Code

To isolate where Play2Code's gains come from, we ablate both the layered memory system (Episode, Skill, World) and the role of the GUI Agent. All ablations are conducted on PlaytestArena using GPT-5.4 as the backbone for both the Game Agent and the GUI Agent, and GPT-5.5 as the judge, with the same settings as in the main text. We report the same rubric-pass score as in the main text.

Memory Ablations. Play2Code organizes memory along two dimensions, *lifecycle* (in-task vs. cross-task) and *scope* (private, shared, or common), yielding three layers: Episode Memory, Skill Memory, and World Memory (§4). To test whether each layer carries its own weight, we evaluate four variants:

- **No Memory.** The Game Agent sees only the current prompt, the previous build, and the GUI Agent's report from the immediately preceding round. Nothing accumulates across rounds or across tasks.
- **Episode Only.** The shared in-task workspace is retained (round history, summaries, fix lists, prior attempts), but cross-task Skill and World memories are disabled.
- **Episode + Skill.** Each agent additionally carries its own cross-task Skill Memory (role-specific patterns and strategies), but the cross-agent World Memory is disabled.
- **Full (Episode + Skill + World).** The configuration used in the main experiments.

Results are reported in Table 6. Removing memory entirely causes the largest drop, confirming that even in-task accumulation is non-trivially useful: without it, the Game Agent often re-introduces fixes it has already tried and discarded in earlier rounds. Adding Skill Memory yields a further gain, most pronounced on genres with recurring implementation pitfalls (e.g., platformer jump arcs, tower-defense wave scheduling), where the Game Agent benefits from patterns crystallized on earlier tasks. World Memory contributes a smaller but consistent improvement, with the clearest gains on genres underrepresented in the agent's prior experience, suggesting that shared design-level knowledge transfers across agents and across tasks.

Variant	Rubric Score
No Memory	64.1
Episode Only	69.3
Episode + Skill	71.8
Full (Episode + Skill + World)	**72.3**

Table 6: Ablation of the layered memory system on PlaytestArena. Each layer contributes additively, with Episode Memory accounting for the largest share and World Memory providing genre-transfer benefits.

Variant	Rubric Score
No GUI Agent (self-verify)	58.7
Full	**72.3**

Table 7: Ablation of the GUI Agent on PlaytestArena (Full memory configuration). Replacing the GUI Agent with Game Agent self-verification causes a larger drop than removing any single memory layer, highlighting the necessity of grounded, execution-based feedback.

GUI Agent Ablation. Beyond memory, we ablate the GUI Agent itself to quantify its contribution to the iterative refinement loop. In the **No GUI Agent** variant, the GUI Agent is removed entirely: the Game Agent receives no external playtest feedback and instead performs self-verification by re-reading its own generated code and reasoning about potential failures. All other settings, including the Full memory configuration, remain unchanged.

Results are shown in Table 7. Removing the GUI Agent causes a substantial drop in rubric score, falling below even the *No Memory* baseline from Table 6. This underscores that the GUI Agent's role is not merely one of logging: its grounded, execution-based feedback surfaces runtime failures and interaction-layer bugs that the Game Agent systematically misses during self-verification—particularly rendering glitches, input-handling errors, and timing-sensitive mechanics that only manifest during actual play. The gap is widest on action and physics-heavy genres, where behavioral correctness cannot be inferred from static code inspection alone.

Taken together, the ablations support that each memory layer is independently useful, with Episode Memory carrying the largest share, Skill Memory accumulating role-specific operational know-how across tasks, and World Memory transferring design-level knowledge across agents and

genres. Furthermore, the GUI Agent ablation demonstrates that grounded playtest feedback is the single most impactful component of Play2Code, contributing more than any individual memory layer.

G Game Complexity Categorization

We partition the 200 games in PlaytestArena into three complexity tiers based on observed refinement behavior. Specifically, we compute for each game the average rubric score gain across five Play2Code rounds, averaged over three backbone models (GPT-5.4, Sonnet 4.6, and Kimi K2.5). Games are then ranked by this gain and split into terciles, yielding *High-complexity* ($\Delta \leq 10$), *Moderate-complexity* ($10 < \Delta \leq 15$), and *Low-complexity* ($\Delta > 15$) tiers, where Δ denotes the absolute rubric score gain over five rounds. We emphasize that this categorization is descriptive rather than causal: the labels reflect observed refinement behavior, not an independent measure of game difficulty. Qualitative inspection nonetheless confirms alignment with intuitive notions of complexity—*Low-complexity* games tend to involve simple, self-contained mechanics (e.g., single-screen puzzles) whose defects the GUI agent can surface and resolve within a few rounds, while *High-complexity* games involve intricate conditional logic, multi-phase state transitions, or emergent interactions that are difficult for the GUI agent to trigger reliably, leaving little actionable signal for the code model.

H GUI-Agent vs. Human Refinement

To situate GUI-agent-driven refinement relative to a human upper bound, we recruit human annotators to replace the GUI Agent in the Play2Code loop and compare the two on the 50 Moderate-complexity tasks from PlaytestArena.

Annotator recruitment and qualifications. We recruited 5 graduate students in computer science, all with prior experience in software development and casual game playing. Annotators were informed that the games were AI-generated but were not told the specific experimental hypothesis being tested, nor were they shown the rubric criteria, to avoid bias toward rubric-optimizing reports. Each task was assigned to exactly one annotator, who completed all rounds for that task independently.

Protocol. The human condition follows the Play2Code protocol exactly, with one substitution: the GUI Agent is replaced by the human annotator. At each round, the annotator plays the current build and submits a free-form fix report in place of the GUI Agent's structured output. Annotators were given the same generation prompt as the Game Agent and instructed to report (i) what they could and could not do, (ii) any bugs or unexpected behaviors encountered, and (iii) concrete suggestions for the next revision. No explicit memory entries are written on the human side; each annotator's observations exist only within the fix report for that round. The Game Agent side is unchanged: Episode, Skill, and World memories are retained and updated exactly as in the standard Play2Code loop, and the same $R_{\max}$=5 round budget applies.

Results. As shown in Figure 6b, human-driven refinement achieves higher rubric scores than GUI-agent-driven refinement across all rounds, confirming that the GUI Agent has not yet closed the gap with human play coverage. The difference is most pronounced in later rounds, where human annotators continue to surface subtle interaction failures that the GUI Agent tends to miss. Nevertheless, the GUI Agent tracks the human trajectory closely in early rounds and requires no human effort, suggesting that further improvements to GUI-agent capability could close the remaining gap.