CODING4GAMES: CAN MODERN CODING AGENTS PLAY MODERN VIDEO GAMES?

Anonymous authors
Paper under double-blind review

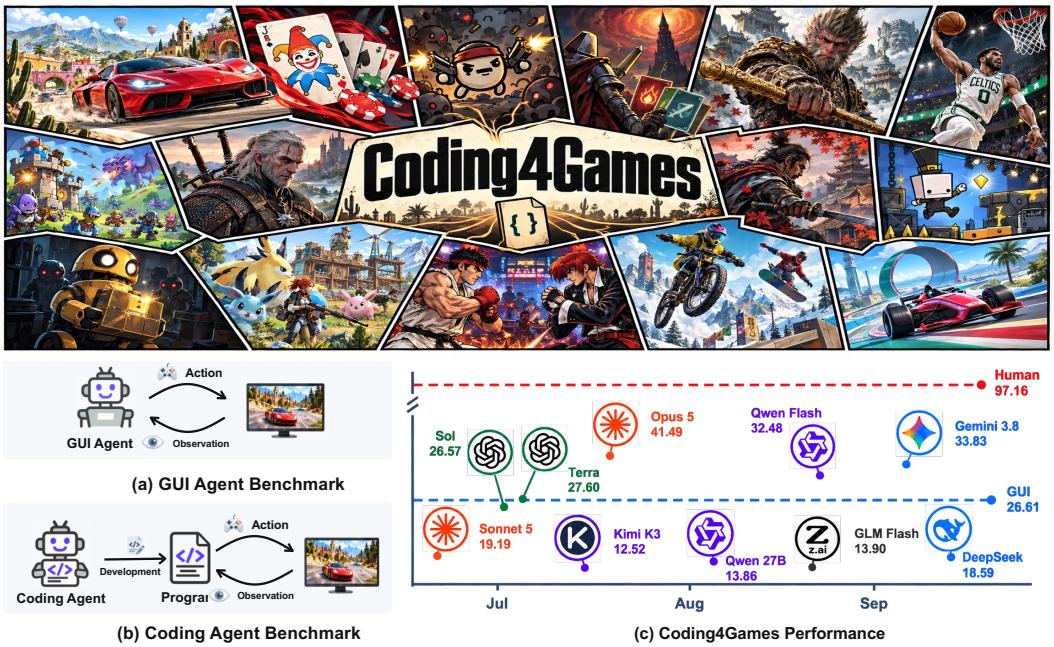

Figure 1: **From playing games directly to building standalone game-playing programs.** (a) In prior computer-use benchmarks, agents play directly through the GUI, querying a language model to determine each action. (b) Coding4Games introduces a different paradigm: a coding agent interacts with the game GUI to develop a game-playing program. Once selected, the program is frozen and plays autonomously without language-model inference. (c) Average normalized performance of ten coding agents by release date, compared with human and GUI-agent baselines across 15 games.

ABSTRACT

Games, especially modern AAA games, challenge language models to perceive complex scenes, control actions, and make decisions over long horizons. Existing game benchmarks primarily test language models as computer-use agents that query a language model throughout play, incurring repeated inference cost and, in real-time games, acting while the environment continues to evolve. We consider the abilities of the large language model via the aspect of modern coding agents: they can utilize tools and skills, develop programs that subsequently play without online model inference. We thus introduce *Coding4Games*, a benchmark for evaluating whether coding agents can develop standalone programs that play modern games through visual observations and standard game inputs. It has three defining features: *1) Diverse Modern Games.* spanning visually and behaviorally varied tasks; compared with classic games, these games offer richer visuals, more complex states, and broader action spaces, placing greater demands on perception, control, and long-horizon decision-making. *2) Development–Test Separation*, in which a coding agent uses a sandbox and generic GUI interaction to iteratively develop a program, after which the selected program is frozen and independently tested without language-model inference; and *3) Outcome and Process Evaluation*, which measures not only game performance and reliability but also how agents use feedback, revise programs, and resolve problems during development. The resulting programs outperform computer-

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.

use agents that query a model at every step in some games, yet remain far below human performance. We additionally examine the effects of development budgets. Coding4Games evaluates whether coding agents can turn their understanding of complex interactive environments into programs that work reliably on their own. Project page: `https://anonymous.4open.science/w/Coding4Games-Page-Review-DE42/`

1 INTRODUCTION

Language-model agents are increasingly used across a wide range of tasks, and coding agents have emerged as capable autonomous developers (Jimenez et al., 2024; Yang et al., 2024; Merrill et al., 2026; OpenAI, 2025; Anthropic, 2025). Yet existing evaluations largely focus on conventional software tasks whose outputs can be verified by unit tests or final environment states. Can coding agents move beyond these settings and develop programs that play modern video games? With rich visual interfaces, complex state spaces, large action spaces, and real-time dynamics, modern games demand stronger capabilities in rule understanding, perception, control, long-horizon reasoning, and iterative debugging. These properties make modern games a challenging and discriminative testbed for whether coding agents can produce robust programs for complex, dynamic environments.

Existing game benchmarks have primarily studied game playing through GUI agents that remain in the loop throughout gameplay: the model observes each frame or state description and selects the next action (Figure 1(a)) (Paglieri et al., 2025; Zhang et al., 2025; Park et al., 2026; Ouyang et al., 2026; Zhang et al., 2026b). This setting evaluates an agent's ability to play games directly through interaction, but requires model inference for every action; in real-time games, the environment may continue to evolve while the model deliberates (Zhang et al., 2025; Ouyang et al., 2026). A complementary line of work studies API-based game-playing programs, which receive structured game states and issue actions through programmatic interfaces. CodeClash (Yang et al., 2026) and ProxyWar (Peng et al., 2026) ask coding agents to write and iteratively revise bots that compete in game arenas, and related work optimizes game policies or world models represented as code (Kuang et al., 2025; Bachrach et al., 2025; Lehrach et al., 2026). However, game-specific APIs expose structured state and action abstractions that bypass the perception and low-level control required in real games, limiting applicability to and transfer across modern titles. (Table 1).

Benchmark	Target	Setting	N	Exec.	Vis.	Actions	Mode	Reset	Test
BALROG (Paglieri et al., 2025)	Online play	Text / grid	6	LM	✓	Semantic	Stepped	✓	✗
VideoGameBench (Zhang et al., 2025)	Online play	Retro	10	LM	✓	GUI	Both	NR	✗
Orak (Park et al., 2026)	Online play	Mixed	12	LM	✓	Game API	NR	✓	✗
GameVerse (Zhang et al., 2026b)	Video reflection	Mixed	15	LM	✓	GUI / Sem.	Subset	NR	✗
GameWorld (Ouyang et al., 2026)	Online play	Web	34	LM	✓	GUI / Sem.	Both	✓	✗
CodeClash (Yang et al., 2026)	Code evolution	Arenas	6	Program	✗	Program API	N/A	✓	✗
ProxyWar (Peng et al., 2026)	Code generation	Arenas	9	Program	✗	Program API	N/A	✓	✓
Coding4Games (ours)	**Coding agents**	**Modern**	**15**	**Program**	✓	**GUI**	**Stepped**	✓	✓

Table 1: **Comparison of game benchmarks.** Coding4Games evaluates coding agents that develop standalone programs for modern video games, with low-level GUI inputs and stepped execution throughout development and evaluation. Target: evaluation target; Setting: game setting; N: number of games; Exec.: gameplay executor; Vis.: visual input; Actions: action interface; Mode: execution mode; Reset: episode reset; Test: separate test.

We bring these two lines of work together and ask: *Can a general coding agent understand a modern game through visual interaction and write a program that plays it using only standard game controls, without access to game-specific APIs?* We define a game-playing program as an executable program built with general-purpose libraries, whose logic maps visual observations to control actions without language-model inference during play. Once developed, such a program can operate in real time, while its behavior can be inspected, rerun, and systematically tested. Developing such a program requires constructing perception from screenshots, translating strategies into precise inputs, and debugging behavior from delayed, partial, and noisy gameplay feedback. Performance in this setting therefore measures not only whether a model can understand a game, but also whether a coding agent can translate that understanding into a reliable standalone program.

To answer this question, we introduce *Coding4Games*, a benchmark with three defining features. *1) Diverse Modern Games.* Coding4Games evaluates 15 games spanning diverse visual, strategic, and control demands, rather than restricting evaluation to classic games with comparatively simple visual interfaces and control schemes. The suite covers games across five categories: Combat & Action Timing, 3D Sports & Precision Control, Movement & Timing, Strategy & Resource Planning, and Multi-step Task Execution. *2) Development–Test Separation.* During the *Development stage*, coding agents explore games, develop programs, and self-test in a persistent sandbox, interacting with games through a shared, gym-like interface provided by the game server. During the *Test stage*, the selected program is frozen and repeatedly evaluated in a fresh environment without language-model inference, testing whether it can play reliably on its own. *3) Outcome and Process Evaluation.* Beyond game performance, we assess how programs use game feedback, how substantially agents revise them, and whether revisions are tested and resolve targeted problems. We further compare frozen programs with GUI agents under stepped interaction.

Our experiments with 10 coding agents demonstrate the potential to turn game interaction into standalone programs. The best programs outperform GUI agents on 13 of the 15 games and even slightly surpass human performance on two games. However, a substantial gap to human performance remains across much of the benchmark. Process analysis further reveals that self-testing often fails to translate feedback into effective strategy revisions. Together, these findings highlight the promise of code as a reusable representation of interactive behavior, while identifying strategy discovery and reliable improvement through interaction as central challenges for coding agents.

In summary, our contributions are as follows:

- We formulate game playing by coding agents as the development of standalone programs through GUI interaction, with a protocol that separates autonomous development from independent, model-free testing.
- We build Coding4Games, an extensible, gym-like environment that brings 15 modern games with diverse perception, control, and planning demands under a common interaction interface, with standardized task initialization and game-specific scoring.
- We evaluate 10 coding agents through game performance and process measures of program grounding, modification depth, and problem resolution, revealing a persistent gap between identifying failures through self-testing and translating feedback into effective strategy.

2 Related Work

Game Agents. Game-playing agents have moved from learned control policies. Deep reinforcement learning and self-play in Atari, board games, and real-time strategy (Mnih et al., 2013; Silver et al., 2018; Vinyals et al., 2019), and policies learned from gameplay demonstrations (Baker et al., 2022; SIMA Team et al., 2024; Magne et al., 2026). Language-model agents that interleave reasoning and action, replan after execution failures, and preserve experience in memory (Yao et al., 2023; Wang et al., 2023; 2025), and further to agents that compose and reuse behavior as executable skills (Tan et al., 2025; Wang et al., 2024). Representing behavior as a program itself has a longer history (Verma et al., 2018; Liang et al., 2023; Xu et al., 2025), and the closest approaches use language models only during construction. Revising Atari policies from execution traces, generating best-response programs against Checkers bots, or synthesizing executable simulators for classical planning (Kuang et al., 2025; Bachrach et al., 2025; Lehrach et al., 2026). So that the resulting artifact runs without per-action LLM decisions. Building on this precedent for separating construction from execution, Coding4Games asks what game-playing behavior a general coding agent can leave in a program artifact after autonomous development, and how that artifact performs during independent testing without LLM inference.

Benchmarks for Coding and Game Agents. Coding benchmarks differ in the scope of the artifact and the environment in which it is evaluated: programs satisfying input–output requirements (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Liu et al., 2023; Jain et al., 2025), repository-level issue resolution and the interfaces through which agents edit and execute code (Jimenez et al., 2024; Yang et al., 2024; Merrill et al., 2026), and interaction in heterogeneous environments, websites, desktops, and embodied control (Liu et al., 2024; Zhou et al., 2024; Xie et al., 2024; Fu et al., 2026). For games, From Code to Play, ProxyWar, and CodeClash are the closest precedents, evaluating program search, testing and repair, and repeated codebase editing under competition (Eberhardinger et al., 2025; Peng et al., 2026; Yang et al., 2026). Game-agent evaluation instead builds

on established platforms (Bellemare et al., 2013; Kempka et al., 2016; Guss et al., 2019; Côté et al., 2019; Hausknecht et al., 2020) and recent language-model suites (Paglieri et al., 2025; Hu et al., 2026; Park et al., 2026), whose conditions are essential to interpreting reported performance: real-time versus paused execution (Zhang et al., 2025), state-based verification (Ouyang et al., 2026; Zhang et al., 2026c) versus model-judged milestones (Zhang et al., 2026b), and improvement across attempts versus transfer to held-out variants (Lin et al., 2026). Coding4Games brings the two perspectives together through autonomous development, explicit selection of a frozen program, and separately invoked independent testing, scoring program performance and reliability alongside development and deployment cost while retaining the action and timing conditions of online baselines.

Self-Improving Agents. Self-improvement methods differ in what changes and what persists across tasks: program libraries learned together with a search-guiding model (Ellis et al., 2023), parameters updated from accepted self-generated rationales (Zelikman et al., 2022), outputs or verbal feedback refined in context (Madaan et al., 2023; Shinn et al., 2023), and reusable tools and skills accumulated with fixed language-model weights (Schick et al., 2023; Cai et al., 2024; Zhou et al., 2026). Closer to our setting, one line searches over executable policies and agent implementations (Kumar et al., 2026; Hu et al., 2025), and recursive variants additionally modify the improvement machinery itself (Zelikman et al., 2024; Yin et al., 2025; Zhang et al., 2026a)—program-level modification that should be distinguished from changing the underlying model. Coding4Games evaluates the game-playing program produced during development without prescribing a self-improvement algorithm; whether such a process yields reliable behavior, and at what development and deployment cost, remains an empirical question.

3 Coding4Games Benchmark

Coding4Games investigates whether modern coding agents can play video games by developing standalone executable programs. For each evaluation task, we specify a game, an objective, a starting scenario, and an interaction budget. Evaluating this capability requires both an environment for developing programs through game interaction and a protocol for testing the resulting programs independently. We therefore introduce a framework that connects a sandboxed development environment to a gym-like game server across a diverse set of games while keeping program development separate from program evaluation. Below, we present the evaluation framework (Sec. 3.1), the games and their tasks (Sec. 3.2), and the measures used to score them (Sec. 3.3).

3.1 Evaluation Framework

As shown in Fig. 2, the framework makes heterogeneous games accessible to coding agents and evaluates the programs they produce under independent, consistent conditions. Given the input prompt of task description, API documentation, and setting description, the coding agent combines a game server for standardized interaction, a Development stage that supports autonomous software development, and an isolated Test stage for evaluating frozen programs. Below, we introduce the main components of the evaluation framework.

Game server. Games differ in how they start, expose observations, and accept inputs, making them difficult to access consistently from a coding environment. The game server wraps each task with a shared, gym-like interface for starting episodes, returning game screenshots, and applying gameplay inputs. It also tracks elapsed game tics and computes task-specific scores from gameplay evidence independently of the coding agent. Keeping initialization, game-tic accounting, and scoring under host control separates gameplay from measurement and supports fair, consistent evaluation across submitted programs.

Development stage. The Development stage gives a coding agent one persistent sandbox for each development run, in which it can use a real coding CLI to write, execute, and debug programs (OpenAI, 2025; Anthropic, 2025). Keeping this sandbox across iterations preserves the agent's tools and intermediate work and reduces variation from changes in the development environment. The sandbox is configured to block direct access to game implementation files and the host-side scorer; game observation and control instead pass through the game server's MCP interface (Anthropic, 2024). The agent can repeatedly interact with the game, test and revise candidates, and select a program artifact, P_{sel}; it may influence game outcomes through normal gameplay inputs, not by editing game files or scores. The host separately records model-token use and game tics, enforcing a token budget, a cumulative game-tic budget, and a per-rollout game-tic limit that bound further interaction and determine when development must stop. Agent and game interactions, candidate tests, and resource use are logged as an auditable development trajectory. Together, the stable sandbox

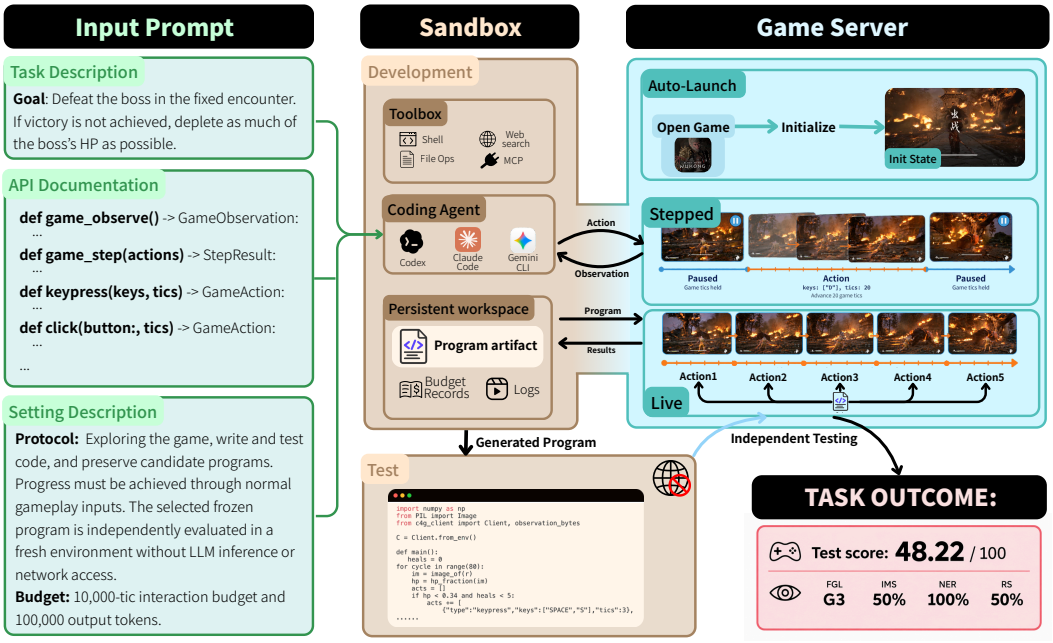

Figure 2: **Evaluation framework.** The game server provides a common interface to heterogeneous games. Coding agents develop and select programs in a sandbox; the frozen programs are then evaluated through independent rollouts with host-controlled initialization, budgets, and scoring.

and recorded trajectory support a reproducible experimental procedure without requiring identical gameplay outcomes.

Test stage. The Test stage evaluates the selected frozen program in a fresh environment, separate from development (Jimenez et al., 2024; Peng et al., 2026). The program runs without model inference or public network access; its code stays fixed, although its internal state can change during play. Each test rollout follows the same per-rollout game-tic limit as development. Before each rollout, the host prepares the designated scenario using recorded initialization actions. After the rollout, the host-side scorer computes a task-specific score from gameplay evidence. Repeated rollouts estimate the program's performance and variability under the same task conditions.

3.2 Games and Tasks

Table 2 lists the games and their associated capability requirements for game-play program. We select games to examine whether coding agents can use interaction to develop standalone game-playing programs under diverse control and planning demands. Our proposed suite comprises 15 tasks, one per game, across five categories: *Combat & Action Timing*, *Strategy & Resource Planning*, *3D Sports & Precision Control*, *Movement & Timing*, and *Multi-step Task Execution*.

Category	Games
Combat & Action Timing	Black Myth: Wukong (BMW); Sekiro: Shadows Die Twice (SEK); The King of Fighters '98 (KOF98)
Strategy & Resource Planning	Balatro (BAL); Slay the Spire (StS); Kingdom Rush 5: Alliance (KR5)
3D Sports & Precision Control	Forza Horizon 5 (FH5); Riders Republic (RR); NBA 2K25 (NBA25)
Movement & Timing	TrackMania Nations Forever (TM); Brotato (BRO); BattleBlock Theater (BBT)
Multi-step Task Execution	Overcooked! 2 (OC2); Aimlab (AIM); Palworld (PAL)

Table 2: **Game categories.** The working suite groups 15 games by their principal interaction demands. Task objectives and scoring criteria appear in Appendix B.

These tasks require different combinations of responding to game feedback, timing precise inputs, tracking resources and persistent state, and coordinating actions across stages. Each task defines a starting scenario, an objective, and an interaction budget to support repeated evaluation under consistent conditions. Each task also defines a quantitative episode score that the host computes from recorded gameplay evidence, independently of the agent's self-assessment. Following prior work (Ouyang et al., 2026; Zhang et al., 2026b), these task-specific scores capture partial progress beyond binary success or failure. Depending on the objective, they measure boss-health depletion, waves or floors reached, distance traveled, or orders completed, with bonuses and penalties for outcomes such as victory or wasted resources. During development, the host returns the score after each episode to guide program comparison and revision. During Test, the same scoring rule measures the performance of the selected frozen program across repeated episodes.

3.3 Evaluation Measures

We design these measures to understand what coding agents produce from game interaction, how they develop it, and at what cost. Independent game scores show how well the selected program plays; the other measures examine whether it uses game state, how deeply the agent revises it, whether revisions address observed problems, and the resources required for development and deployment. Together, they distinguish programs with similar game scores but different decision mechanisms, development trajectories, and costs.

Game performance. For each model–game pair, we evaluate the program selected during development in N independent test episodes and use the mean episode score as its game performance:

$$\bar{s} = \frac{1}{N} \sum_{k=1}^{N} s_k, \tag{1}$$

where s_k is the score in test episode k. Repeated testing reduces sensitivity to randomness in the game and variation in the program's behavior, either of which can change the outcome of a single episode. We also report variation across episodes.

Dependence on game state. We propose *Final Grounding Level* (FGL) to describe how the selected program uses game information to choose actions: **G0**, fixed actions; **G1**, actions based on time or counters; **G2**, reactions to simple observed signals; **G3**, decisions based on task-relevant game state; and **G4**, state-based decisions with memory, planning, and failure recovery. We use these levels to study how the agent encodes its understanding of the game in the program and turns that understanding into gameplay control.

Modification depth. We propose *Iterative Modification Score* (IMS) to measure the depth of changes between successive self-tested program versions. Each change receives a rating d_i from 0 to 4: no change, parameter adjustment, local patch, component change, or strategy shift. For N changes, IMS is the average rating scaled to 0–100:

$$\text{IMS} = \frac{100}{4N} \sum_{i=1}^{N} d_i. \tag{2}$$

A higher value means deeper changes, not necessarily better results. We compare each tested version's score and resource use to see how performance evolves during development (Lin et al., 2026).

Problem resolution. For each problem identified before a modification, we check whether the next self-test reaches a situation where that problem can be assessed. Across M problem–modification pairs, let $e_j = 1$ if pair j receives such a test and 0 otherwise. Let r_j be 0 if the problem remains, 0.5 if partly resolved, and 1 if resolved. We measure how often modifications can be assessed (NER) and how fully the assessed problems are resolved (RS):

$$\text{NER} = \frac{\sum_{j=1}^{M} e_j}{M}, \qquad \text{RS} = \frac{\sum_{j=1}^{M} e_j r_j}{\sum_{j=1}^{M} e_j}. \tag{3}$$

NER is undefined when $M = 0$; RS is undefined if no pair receives a relevant test. Together they distinguish untested changes from observed problem resolution.

Model	Harness	BAL	FH5	StS	BMW	BRO	NBA25	RR	KOF98	TM	KR5	OC2	SEK	AIM	PAL	BBT	Rank
Budget (all)		10k	10k	10k	10k	10k	10k	10k	10k	10k	20k	10k	10k	1k	10k	10k	
Budget (rollout)		1k	1k	1k	1k	1k	1k	1k	1k	1k	10k	1k	5k	320	1.2k	3k	
Budget (token)		20M	20M	20M	20M	20M	20M	20M	10M	1M	10M	10M	50M	5M	10M	10M	
Human Player	—	100.0	99.5	100.0	100.0	100.0	100.0	77.7	100.0	100.0	100.0	100.0	100.0	80.2	100.0	100.0	—
GUI Agent																	
GPT-5.6-sol	Codex	4.8	70.7	8.3	29.5	13.7	80.7	5.4	32.8	16.3	1.6	0.0	11.2	22.0	85.1	14.0	—
Claude Sonnet 5	CC	4.0	100.0	66.8	12.3	93.3	38.7	18.6	8.2	0.0	1.8	0.0	0.0	24.9	30.0	3.7	—
Coding Agent																	
Closed																	
GPT-5.6-sol	Codex	16.3	14.7	0.0	39.6	5.3	58.1	47.3	60.1	0.9	74.2	3.0	13.7	43.4	15.0	7.0	5
GPT-5.6-terra	Codex	17.0	72.2	0.0	73.2	37.1	0.0	53.1	54.6	1.9	28.9	0.0	7.2	55.4	6.5	7.0	4
Claude Opus 5	CC	32.5	47.7	91.8	38.1	89.1	58.1	11.1	16.4	0.8	28.3	0.0	2.9	100.0	68.0	37.7	1
Claude Sonnet 5	CC	9.9	52.9	0.0	12.1	0.7	38.7	0.0	62.8	13.5	15.4	0.0	2.7	47.8	24.3	7.0	6
Gemini-3.8-flash	GC	20.7	50.3	66.8	3.7	30.3	22.6	17.9	98.4	18.1	27.3	1.5	10.0	43.9	89.0	7.0	2
Qwen3.8-flash	Codex	0.0	30.6	0.0	65.4	95.1	32.3	100.0	19.1	20.6	19.6	0.0	5.5	54.1	41.1	3.7	3
Open																	
Qwen3.8-27B	Codex	18.5	14.7	33.3	0.0	1.2	58.1	14.1	5.5	4.6	28.9	0.0	0.0	23.9	5.1	0.0	9
GLM-5.3-Flash	Codex	16.0	28.0	0.0	5.1	1.3	71.0	1.3	0.0	34.0	0.0	7.4	1.6	42.9	0.0	0.0	8
deepseek-v4.1-flash	Codex	0.0	0.0	58.3	10.3	28.5	6.4	16.7	10.9	33.9	30.4	1.5	0.7	34.1	40.1	7.0	7
Kimi-K3	Codex	0.0	22.1	0.0	47.2	0.0	54.8	32.4	16.4	1.2	1.8	1.5	9.7	0.0	0.8	0.0	10

Table 3: **Main results on Coding4Games.** We compare coding agents, GUI agents, and a human player using mean game scores, min–max normalized to 0–100 within each game across all available entries. The top 3 rows report each game's cumulative game-tic budget, per-rollout game-tic limit, and token budget. Game names follow Tab. 2; CC and GC denote Claude Code and Gemini CLI.

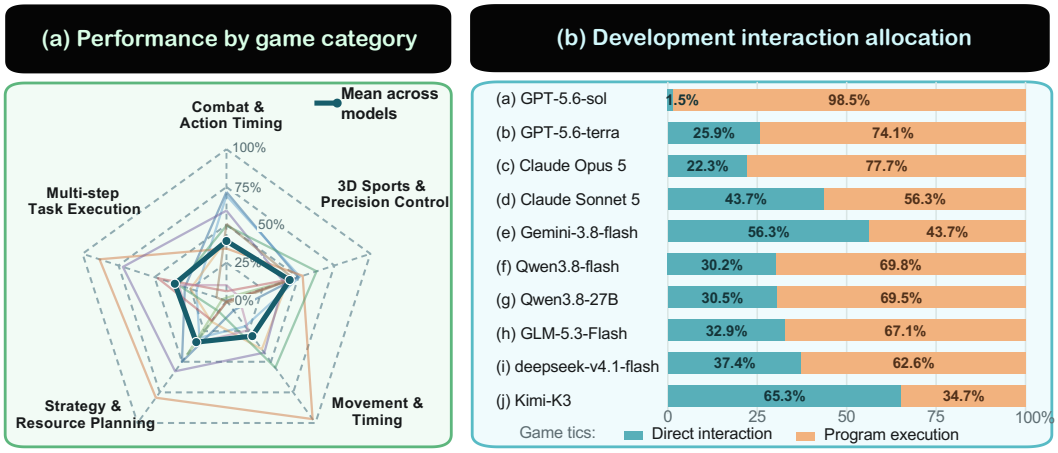

Figure 3: **Game-category performance and development interaction modes.** (a) All 10 coding models overlaid on one radar, with game-category axes and a shared 0–1 scale. (b) Direct and Program game tics as fractions of their sum within one illustrative development run per model.

4 Experiments

Our evaluation includes 10 coding-agent models. The closed-model group comprises GPT-5.6-sol and GPT-5.6-terra (OpenAI, 2026), Claude Opus 5 (Anthropic, 2026a), Claude Sonnet 5 (Anthropic, 2026b), Gemini-3.8-flash (Google DeepMind, 2026), and Qwen3.8-flash (Alibaba Cloud, 2026). The open-model group comprises Qwen3.8-27B (Qwen Team, 2026), GLM-5.3-Flash (AutoClaw Team, 2026), deepseek-v4.1-flash (DeepSeek-AI, 2026), and Kimi-K3 (Kimi Team, 2026). We use each model's associated coding harness where available and Codex as a common harness for the remaining open models. The human reference consists of a single player who is briefed on the basic rules and controls of each game but has not previously developed proficiency in playing it. The GUI baselines use GPT-5.6-sol and Claude Sonnet 5, with the game paused during agent inference. We set token and game-tic budgets tailored to each game, with the specific values reported in Table 3. We evaluate each method on each game over 3 test episodes.

4.1 Main Results

Overall performance. Table 3 shows substantial variation across models and games. Six coding models achieve the highest coding score on at least one game: Claude Opus 5 leads on four, while

Model	G0	G1	G2	G3	G4	IMS	NER	RS
Closed								
GPT-5.6-sol	9.1	63.6	27.3	0.0	0.0	64.3	100.0	41.7
GPT-5.6-terra	18.2	63.6	9.1	9.1	0.0	62.5	100.0	62.5
Claude Opus 5	0.0	10.0	20.0	20.0	50.0	53.6	93.0	42.5
Claude Sonnet 5	0.0	18.2	36.4	27.3	18.2	50.0	100.0	83.3
Gemini-3.8-flash	9.1	18.2	18.2	36.4	18.2	67.5	94.4	58.8
Qwen3.8-flash	20.0	0.0	10.0	20.0	50.0	56.4	85.3	44.8
Open								
Qwen3.8-27B	0.0	0.0	20.0	50.0	30.0	50.7	82.9	50.0
GLM-5.3-Flash	30.0	30.0	10.0	30.0	0.0	56.8	100.0	35.7
deepseek-v4.1-flash	0.0	0.0	10.0	50.0	30.0	67.0	80.0	56.2
Kimi-K3	9.1	9.1	27.3	18.2	36.4	57.6	95.0	42.1

Table 4: **Process evaluation of coding agents.** We report final-program grounding distributions (G0–G4) and development metrics (IMS, NER, and RS), all on a 0–100 scale. Metric definitions follow Section 3.3.

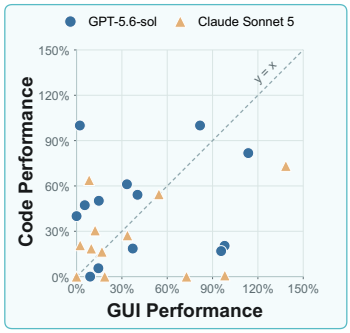

Figure 4: **GUI agents *v.s.* frozen programs.** Points above the dashed line favor programs; points below favor GUI agents.

GLM-5.3-Flash leads on NBA 2K25, TrackMania, and Overcooked! 2. Figure 3(a) also shows different category profiles, with GPT-5.6-sol strongest in Combat & Action Timing and Qwen3.8-flash in 3D Sports & Precision Control. Even within a category, no model consistently leads: Claude Opus 5 scores highest among coding agents on Balatro and Slay the Spire, whereas GPT-5.6-sol leads on Kingdom Rush 5. The human player outperforms all agents on 12 of 15 games; GUI agents exceed this reference on Forza Horizon 5, and coding agents do so on Riders Republic and Aimlab. In the Computer Use Comparison, GPT-5.6-sol's frozen programs outperform its GUI baseline on eight games and underperform on seven; Claude Sonnet 5's programs win on seven, lose on six, and tie on two. Together, these results show that strong performance remains specific to particular models and games, and that frozen programs offer an alternative to online GUI control whose relative effectiveness varies across tasks.

4.2 Understanding the Performance

1. What strategies do agents implement? *Models differ in how they use game information to control actions.* Table 4 shows that GPT-5.6-sol mainly produces G1 controllers, which select actions according to timing or counters rather than observed game conditions. In contrast, Qwen3.8-27B more often produces G3 and G4 programs that use game state to guide decisions, while G4 is the most common level for Claude Opus 5 and Qwen3.8-flash, indicating the use of memory, planning, and recovery alongside game state. These distributions distinguish programs that largely follow predefined schedules from those that adjust decisions using feedback and accumulated information. Coding agents therefore differ in the control logic they encode, although greater reliance on game state does not by itself establish better performance.

2. Do more complex strategies perform better? *Greater strategy complexity does not necessarily translate into higher game performance.* In Forza Horizon 5, Qwen3.8-flash's PID controller achieves a mean score of 443.7, compared with 645.7 for Claude Sonnet 5's simpler near-field route follower. Its development traces record road departures and false recovery triggers, suggesting unresolved control-calibration difficulties without isolating their contribution to the score gap. This implementation–execution gap offers a possible explanation for why stronger conventional programming performance may not transfer to standalone gameplay.

3. Do agents resolve failures through self-testing? *Agents can identify problems during self-testing without reliably repairing them.* Table 4 reports NER values ranging from 80.0% to 100%, but RS values spanning a wider range from 35.7% to 83.3% for identified problems targeted by modifications. GPT-5.6-sol and Gemini-3.8-flash achieve NER values of 100.0% and 94.4%, respectively, yet their RS values are 41.7% and 58.8%, respectively. Obtaining relevant test feedback therefore does not ensure successful repair. Moreover, local problem resolution still requires independent testing to establish improved game performance, as the case study below illustrates.

4. How do agents interact during debugging? *Most models rely primarily on script execution to obtain feedback during development and debugging.* Figure 3(b) shows that eight of ten models devote most observed game tics to script execution, with GPT-5.6-sol allocating 98.5%. Kimi-K3 is an exception, allocating 65.3% of its game tics to direct interaction through agent-issued commands. These shares characterize development interaction, but do not distinguish exploration

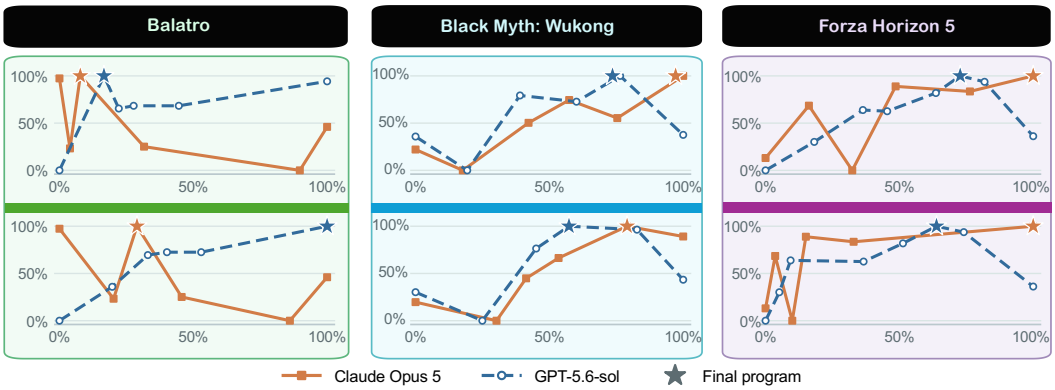

Figure 5: **Development progress *v.s.* test performance.** Development is capped at 15,000 game tics and 25 million tokens. Programs sampled at nonuniform intervals are independently tested. Columns correspond to games; the horizontal axis shows game tics (top) or tokens (bottom), and the vertical axis shows mean test score. Both axes are min–max normalized to 0–100% within each curve. Stars mark the final selected programs.

from targeted debugging or measure log-inspection time. Unequal run counts and budgets also limit their interpretation as stable model preferences.

5. Where do frozen programs fail? *Partial state representations limit how frozen programs adapt to feedback.* In Riders Republic, GPT-5.6-sol's selected program detects crashes using a red-screen threshold but executes tricks on a fixed counter-based schedule. Feedback thus supports recovery without adapting trick selection or timing to the gameplay state: detecting failures alone is insufficient to guide the actions that precede them.

6. How do frozen programs compare with GUI agents? *Frozen programs can match or exceed the same model's GUI agent on some games without model inference during testing.* Figure 4 compares paired results for GPT-5.6-sol and Claude Sonnet 5. GPT-5.6-sol's programs score higher on eight of 15 paired games and lower on seven; Claude Sonnet 5's programs score higher on seven, lower on six, and tie on two. Both models' programs outperform their GUI agents on Balatro, Kingdom Rush 5, and KOF98, spanning strategy and combat tasks. However, both GUI agents score higher on Forza Horizon 5, Slay the Spire, Brotato, and Palworld. These results show that programs produced during development can support comparable or better gameplay on a subset of tasks without continued model involvement, but they do not yet provide a general replacement for online GUI-agent decision-making.

7. How do development budgets affect frozen-program performance? *Both token and game-tic budgets influence performance, but their benefits vary across games.* We examine how game-tic and token budgets affect independent test performance for Claude Opus 5 and GPT-5.6-sol on Balatro, Black Myth: Wukong, and Forza Horizon 5 in Fig. 5. We vary the cumulative game-tic budget from 5k to 15k in increments of 2k while fixing the token budget at 10 million, and vary the token budget from 5 to 25 million in increments of 3 million while fixing the game-tic budget at 10k.

5 Conclusion

We introduced Coding4Games, a benchmark that asks coding agents to develop standalone programs that play 15 modern games from visual observations through keyboard-and-mouse inputs, separating autonomous development in a sandbox from independent, model-free testing of a frozen program, and evaluating the development process alongside game performance and cost. Across 10 frontier coding agents, the resulting programs match or exceed computer-use agents on 13/15 games, yet remain far below human play, and our process measures locate the gap less in strategy sophistication than in revision: agents expose modifications to self-testing but often fail to resolve the failures they target, leaving programs that cover part of the gameplay conditions or rest on visual assumptions that do not hold outside the observed scenario. Turning execution feedback into revisions that generalize is, on this evidence, the central open problem for coding agents in dynamic environments.

AI USE STATEMENT

The research questions, hypotheses, and conceptual framework were developed by the authors without generative AI assistance. We used generative AI tools, primarily Codex, to discuss and refine the methodology and to carry out most of the code implementation. AI tools also assisted with recording and analyzing experimental logs, analyzing and interpreting experimental results, conducting the literature survey, polishing the manuscript, and generating visual materials, including background illustrations. The authors reviewed the entire manuscript and the outputs produced by the code, and rigorously tested the implementation against the intended requirements. The authors take responsibility for the final text, claims, results, and research artifacts, including those produced with AI assistance. The use of models as experimental subjects is described separately in the experimental protocol.

REPRODUCIBILITY STATEMENT

The main text and appendix describe the benchmark protocol, game tasks, and evaluation procedures. Our anonymous project website provides selected frozen program artifacts and recorded gameplay for inspection. We will release the complete benchmark code, execution and evaluation scripts, and experimental logs when the paper is made publicly available. Reproducing the experiments requires access to the relevant games, a Windows machine, and the model APIs used in our study. A consolidated archive of experimental configurations and setup instructions is not yet available.

REFERENCES

Alibaba Cloud. Qwen3.8-Flash Model Information, 2026. URL https://help.aliyun.com/en/model-studio/qwen3-8-flash. Accessed: 2026-09-22.

Anthropic. Introducing the Model Context Protocol, 2024. URL https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-09-14.

Anthropic. Claude 3.7 Sonnet and Claude Code, 2025. URL https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2026-09-14.

Anthropic. Claude Opus 5 System Card, 2026a. URL https://anthropic.com/claude-opus-5-system-card. Accessed: 2026-09-22.

Anthropic. Claude Sonnet 5 System Card, 2026b. URL https://anthropic.com/claude-sonnet-5-system-card. Accessed: 2026-09-22.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021.

AutoClaw Team. GLM-5.3-Flash: More Intelligence with Less Compute, 2026. URL https://autoclaw.z.ai/blog/model/glm-5.3-flash/. Published: 2026-09-26. Accessed: 2026-09-26.

Yoram Bachrach, Edan Toledo, Karen Hambardzumyan, Despoina Magka, Martin Josifoski, Minqi Jiang, Jakob Foerster, Roberta Raileanu, Tatiana Shavrina, Nicola Cancedda, Avraham Ruderman, Katie Millican, Andrei Lupu, and Rishi Hazra. Combining Code Generating Large Language Models and Self-Play to Iteratively Refine Strategies in Games. In *International Joint Conference on Artificial Intelligence (IJCAI), Demonstrations Track*, pp. 10999–11003, 2025. doi: 10.24963/ijcai.2025/1249.

Bowen Baker, Ilge Akkaya, Peter Zhokhov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video PreTraining (VPT): Learning to Act by Watching Unlabeled Online Videos. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. doi: 10.1613/jair.3912.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large Language Models as Tool Makers. In *International Conference on Learning Representations (ICLR)*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. TextWorld: A Learning Environment for Text-Based Games. In *Computer Games Workshop (CGW), Revised Selected Papers*, volume 1017 of *Communications in Computer and Information Science*, pp. 41–75. Springer, 2019. doi: 10.1007/978-3-030-24337-1_3.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the Limits of KV Cache Compression, 2026. URL `https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash`. Accessed: 2026-09-22.

Manuel Eberhardinger, James Goodman, Alexander Dockhorn, Diego Perez-Liebana, Raluca D. Gaina, Duygu Çakmak, Setareh Maghsudi, and Simon Lucas. From Code to Play: Benchmarking Program Search for Games Using Large Language Models. *IEEE Transactions on Games*, 2025. doi: 10.1109/TG.2025.3614499. Early access.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing Generalizable, Interpretable Knowledge with Wake–Sleep Bayesian Program Learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023. doi: 10.1098/rsta.2022.0050.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. CaP-X: A Framework for Benchmarking and Improving Coding Agents for Robot Manipulation. In *International Conference on Machine Learning (ICML)*, 2026.

Google DeepMind. Gemini 3.8 Flash Model Card, 2026. URL `https://deepmind.google/models/model-cards/gemini-3-8-flash/`. Accessed: 2026-09-22.

William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. MineRL: A Large-Scale Dataset of Minecraft Demonstrations. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2442–2448, 2019. doi: 10.24963/ijcai.2019/339.

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. Interactive Fiction Games: A Colossal Adventure. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 34, pp. 7903–7910, 2020. doi: 10.1609/aaai.v34i05.6297.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence with APPS. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2021.

Lanxiang Hu, Mingjia Huo, Yuxuan Zhang, Haoyang Yu, Eric P. Xing, Ion Stoica, Tajana Rosing, Haojian Jin, and Hao Zhang. lmgame-Bench: How Good Are LLMs at Playing Games? In *International Conference on Learning Representations (ICLR)*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *International Conference on Learning Representations (ICLR)*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *International Conference on Learning Representations (ICLR)*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-Based AI Research Platform for Visual Reinforcement Learning. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, 2016. doi: 10.1109/CIG.2016.7860433.

Kimi Team. Kimi K3: Open Frontier Intelligence. *arXiv preprint arXiv:2607.24653*, 2026. arXiv version 1.

Zhiyi Kuang, Ryan Rong, YuCheng Yuan, and Allen Nie. Learning Game-Playing Agents with Generative Code Optimization. *arXiv preprint arXiv:2508.19506*, 2025. ICML Workshop on Programmatic Representations for Agent Learning.

Akarsh Kumar, Ryan Bahlous-Boldi, Prafull Sharma, Phillip Isola, Sebastian Risi, Yujin Tang, and David Ha. Digital Red Queen: Adversarial Program Evolution in Core War with LLMs. In *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 691–700. ACM, 2026. doi: 10.1145/3795095.3805116.

Wolfgang Lehrach, Daniel Hennes, Miguel Lázaro-Gredilla, Xinghua Lou, Carter Wendelken, Zun Li, Antoine Dedieu, Marc Lanctot, Atil Iscen, John Schultz, Marcus Chiam, Ian Gemp, Piotr Zielinski, Satinder Singh, and Kevin P. Murphy. Code World Models for General Game Playing. In *International Conference on Learning Representations (ICLR)*, 2026.

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. In *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, 2023. doi: 10.1109/ICRA48891.2023.10160591.

Mingxian Lin, Shengju Qian, Yuqi Liu, Yi-Hua Huang, Yiyu Wang, Wei Huang, Yitang Li, Fan Zhang, Zeyu Hu, Lingting Zhu, Xin Wang, and Xiaojuan Qi. OmniGameArena: A Unified UE5 Benchmark for VLM Game Agents with Improvement Dynamics. *arXiv preprint arXiv:2606.09826*, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as Agents. In *International Conference on Learning Representations (ICLR)*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Loïc Magne, Anas Awadalla, Guanzhi Wang, Yinzhen Xu, Joshua Belofsky, Fengyuan Hu, Joohwan Kim, Ludwig Schmidt, Georgia Gkioxari, Jan Kautz, Yisong Yue, Yejin Choi, Yuke Zhu, and Linxi Fan. NitroGen: An Open Foundation Model for Generalist Gaming Agents. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 21511–21521, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *International Conference on Learning Representations (ICLR)*, 2026.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013. NIPS Deep Learning Workshop.

OpenAI. Introducing Codex, 2025. URL https://openai.com/index/introducing-codex/. Accessed: 2026-09-14.

OpenAI. GPT-5.6 System Card, 2026. URL https://deploymentsafety.openai.com/gpt-5-6. Accessed: 2026-09-22.

Mingyu Ouyang, Siyuan Hu, Kevin Qinghong Lin, Hwee Tou Ng, and Mike Zheng Shou. GameWorld: Towards Standardized and Verifiable Evaluation of Multimodal Game Agents. In *European Conference on Computer Vision (ECCV)*, volume 17053 of *Lecture Notes in Computer Science*, pp. 41–59. Springer, 2026. doi: 10.1007/978-3-032-37255-0_3.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. BALROG: Benchmarking Agentic LLM and VLM Reasoning On Games. In *International Conference on Learning Representations (ICLR)*, 2025.

Dongmin Park, Minkyu Kim, Beongjun Choi, Junhyuck Kim, Keon Lee, Jonghyun Lee, Inkyu Park, Byeong-Uk Lee, Jaeyoung Hwang, Jaewoo Ahn, Ameya S. Mahabaleshwarkar, Bilal Kartal, Pritam Biswas, Yoshi Suhara, Kangwook Lee, and Jaewoong Cho. Orak: A Foundational Benchmark for Training and Evaluating LLM Agents on Diverse Video Games. In *International Conference on Learning Representations (ICLR)*, 2026.

Wenjun Peng, Xinyu Wang, and Qi Wu. ProxyWar: Dynamic Assessment of LLM Code Generation in Game Arenas. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026. doi: 10.1145/3744916.3773220.

Qwen Team. Qwen3.8-27B Model Card, 2026. URL https://huggingface.co/Qwen/Qwen3.8-27B. Accessed: 2026-09-22.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023. doi: 10.52202/075280-0377.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

SIMA Team, Maria Abi Raad, Arun Ahuja, Catarina Barros, Frederic Besse, Andrew Bolt, Adrian Bolton, Bethanie Brownfield, Gavin Buttimore, Max Cant, Sarah Chakera, Stephanie C. Y. Chan, Jeff Clune, Adrian Collister, Vikki Copeman, Alex Cullum, Ishita Dasgupta, Dario de Cesare, Julia Di Trapani, Yani Donchev, Emma Dunleavy, Martin Engelcke, Ryan Faulkner, Frankie Garcia, Charles Gbadamosi, Zhitao Gong, Lucy Gonzales, Karol Gregor, Kshitij Gupta, Arne Olav Hallingstad, Tim Harley, Sam Haves, Felix Hill, Ed Hirst, Drew A. Hudson, Jony Hudson, Steph Hughes-Fitt, Danilo J. Rezende, Mimi Jasarevic, Laura Kampis, Rosemary Ke, Thomas Keck, Junkyung Kim, Oscar Knagg, Kavya Kopparapu, Rory Lawton, Andrew Lampinen, Shane Legg, Alexander Lerchner, Marjorie Limont, Yulan Liu, Maria Loks-Thompson, Joseph Marino, Kathryn Martin Cussons, Loic Matthey, Siobhan Mcloughlin, Piermaria Mendolicchio, Hamza Merzic, Anna Mitenkova, Alexandre Moufarek, Valeria Oliveira, Yanko Oliveira, Hannah Openshaw, Renke Pan, Aneesh Pappu, Alex Platonov, Ollie Purkiss, David Reichert, John Reid, Pierre Harvey Richemond, Tyson Roberts, Giles Ruscoe, Jaume Sanchez Elias, Tasha Sandars, Daniel P. Sawyer, Tim Scholtes, Guy Simmons, Daniel Slater, Hubert Soyer, Heiko Strathmann, Peter Stys, Allison C. Tam, Denis Teplyashin, Tayfun Terzi, Davide Vercelli, Bojan Vujatovic, Marcus Wainwright, Jane X. Wang, Zhengdong Wang, Daan Wierstra, Duncan Williams, Nathaniel Wong, Sarah York, and Nick Young. Scaling Instructable Agents Across Many Simulated Worlds. *arXiv preprint arXiv:2404.10179*, 2024.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, Ruyi An, Molei Qin, Chuqiao Zong, Longtao Zheng, Yujie Wu, Xiaoqiang Chai, Yifei Bi, Tianbao Xie, Pengjie Gu, Xiyun Li, Ceyao Zhang, Long Tian, Chaojie Wang, Xinrun Wang, Börje F. Karlsson, Bo An, Shuicheng Yan, and Zongqing Lu. Cradle: Empowering Foundation Agents towards General Computer Control. In *International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 58658–58725. PMLR, 2025.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically Interpretable Reinforcement Learning. In *International Conference on Machine Learning (ICML)*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5045–5054. PMLR, 2018.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Çağlar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning. *Nature*, 575 (7782):350–354, 2019. doi: 10.1038/s41586-019-1724-z.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, Explain, Plan and Select: Interactive Planning with Large Language Models Enables Open-World Multi-Task Agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, Xiaojian Ma, and Yitao Liang. JARVIS-1: Open-World Multi-Task Agents with Memory-Augmented Multimodal Language Models. *IEEE Trans-*

actions on Pattern Analysis and Machine Intelligence, 47(3):1894–1907, 2025. doi: 10.1109/TPAMI.2024.3511593.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024.

Zhongwen Xu, Xianliang Wang, Siyi Li, Tao Yu, Liang Wang, Qiang Fu, and Wei Yang. Agents Play Thousands of 3D Video Games. *arXiv preprint arXiv:2503.13356*, 2025.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024.

John Yang, Kilian Lieret, Joyce Yang, Carlos E. Jimenez, Muhtasham Oblokulov, Aryan Siddiqui, Ofir Press, Ludwig Schmidt, and Diyi Yang. CodeClash: Benchmarking Goal-Oriented Software Engineering. In *International Conference on Machine Learning (ICML)*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel Agent: A Self-Referential Agent Framework for Recursively Self-Improvement. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 27890–27913. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1354.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping Reasoning with Reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation. In *Conference on Language Modeling (COLM)*, 2024.

Alex L. Zhang, Thomas L. Griffiths, Karthik R. Narasimhan, and Ofir Press. VideoGameBench: Can Vision-Language Models Complete Popular Video Games? *arXiv preprint arXiv:2505.18134*, 2025.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. In *International Conference on Learning Representations (ICLR)*, 2026a.

Kuan Zhang, Dongchen Liu, Qiyue Zhao, Jinkun Hou, Xinran Zhang, Qinlei Xie, Miao Liu, and Yiming Li. GameVerse: Can Vision-Language Models Learn from Video-based Reflection? *arXiv preprint arXiv:2603.06656*, 2026b.

Ruizhi Zhang, Ye Huang, Yuangang Pan, Chuanfu Shen, Zhilin Liu, Ting Xie, Wen Li, and Lixin Duan. PokeGym: A Visually-Driven Long-Horizon Benchmark for Vision-Language Models. *arXiv preprint arXiv:2604.08340*, 2026c. arXiv version 1.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, Runyu Yang, Qiangbin Liu, Xinlei Yu, Jianmin Zhou, Na Wang, Chunyang Sun, and Jun Wang. Memento-Skills: Let Agents Design Agents. *arXiv preprint arXiv:2603.18743*, 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *International Conference on Learning Representations (ICLR)*, 2024.

A Implementation Details

This appendix describes the implemented game-tool interface, recorded initialization procedure, and server-side scoring pipeline.

A.1 Game and Submission APIs

The coding agent accesses server-controlled game tools through an MCP adapter. Both the coding agent and the generated program observe gameplay through screenshots only. Generated programs can invoke the corresponding tools through a supplied client connected to the authenticated local bridge. These are benchmark interaction and lifecycle APIs, rather than game-engine APIs exposing semantic actions such as selecting a card or buying an item. General-purpose shell and filesystem operations are supplied by the coding CLI separately.

Table 5 lists the registered tools. Development exposes all nine tools; independent testing exposes only `game.observe`, `game.step`, `game.stop`, and `bench.status`. In independent testing, the server starts the episode and launches the selected artifact, so the program cannot create additional episodes or change the submitted artifact through these tools.

Table 5: **Registered benchmark APIs.** D: Development stage; T: Test stage. Optional arguments are described in the text; signatures summarize the principal inputs.

Tool	Phase	Inputs and effect
`game.start`	D	`mode` (set to `stepped` in the current protocol), optional `max_tics`: initialize a fresh episode and return its identifier and initial observation.
`game.observe`	D, T	Optional `episode_id`: capture an observation, or return the settled result of an ended episode.
`game.step`	D, T	Structured `actions`, or `controls` and `tics`: execute physical inputs in a stepped episode and return an observation and timing receipt.
`game.stop`	D, T	Optional `episode_id`: stop the episode, settle its score, and return the result.
`bench.status`	D, T	No arguments: report budget consumption and session status. The final-test response excludes the development episode history.
`bench.replay`	D	`uid`, `start`, `end`, optional `every`: retrieve frames from a completed episode recording without advancing the game.
`submission.save`	D	`path`, `entrypoint`, optional description and dependencies: archive an immutable executable candidate and return its identifier.
`submission.run`	D	`id`, optional `max_tics`: self-test a saved candidate in a fresh episode, using the development sandbox and shared budget.
`submission.finish`	D	`id`: select a saved artifact and end development. Early submission is accepted; exhausting the development budget is not required.

Physical actions and timing. In keyboard/mouse mode, `game.step` accepts an ordered list of `click`, `double_click`, `move`, `drag`, `scroll`, `keypress`, and `wait` actions. Each action specifies a positive duration in game tics. Coordinates refer to a 1280×768 observation surface; scrolling uses wheel steps, and dragging specifies a two-point path. In gamepad mode, a call instead specifies physical control tokens and a duration. The two formats cannot be mixed. These controls do not encode game-specific strategies.

The server validates the action batch before execution. Each stepped action resumes the game, delivers the input, releases controls, and pauses the game before returning observations. Receipts distinguish requested and actual tics, executed actions, and truncation due to termination or budget limits. Tics are computed from measured running time at the scenario's configured rate, rather than read from an internal engine clock.

Returned evidence and artifact identity. Tool results associate observations and outcomes with episode identifiers, resource use, and available evidence references. During gameplay, `score` is

null rather than zero. After settlement, `done`, `complete`, `outcome_kind`, and `end_reason` distinguish an ended episode from a valid scored result. A successful execution is not itself a submission: saving produces a candidate archive, while finishing explicitly selects its identifier. Independent tests use that selected archive rather than the mutable development directory. If the agent has not yet submitted when it has consumed 70% of its development budget, it receives a reminder to wrap up development and prepare its final submission. This reminder does not require immediate termination; the agent may continue working within the remaining budget or choose to submit earlier.

A.2 Recorded Initialization and Fixed Task Starts

Every episode follows one server-controlled initialization procedure: prepare the registered task state, cold-start the game, replay a recorded startup sequence, and perform the configured readiness checks before transferring control. This procedure places repeated evaluations at the same designated task entry, such as a fresh run, fixed match, boss encounter, or route. It does not require every game to have the same notion of a "level," nor does it guarantee identical random draws or trajectories.

Recording and replay. Menu navigation is prepared in advance as a recorded sequence of physical inputs and waits, stored in the game configuration as `menu_skip`. Entries preserve the input parameters, duration, and, where relevant, controller index. The server launches the executable, waits for and focuses the game window, then replays this sequence to enter the registered scenario. Thus, repeated menu navigation is automated by replaying recorded operations; it is not solved anew by the evaluated agent. The recording is an action sequence, not a video used to synthesize gameplay. Scenarios can request open-loop replay; the controller also supports screen-aware replay. Neither introduces a game-playing strategy after control is transferred.

Preparing the initial state. Before launch, the server stops the previous game process. When a scenario specifies a reference checkpoint, it restores the corresponding archive into the configured save location, replacing the covered files according to the restore configuration. Otherwise, the preparation hook clears a resumable run where the game has a registered cleanup routine.

Readiness and scope of reproducibility. Where configured, the server checks screen text and exact field values before handoff. It inspects the game while paused and advances short pulses when a transition needs to complete, avoiding an unbounded period of gameplay during recognition. Failed configured checks can trigger a cold-start retry and ultimately an initialization error. Window geometry is also configured where required by the scorer. These checks verify selected visible conditions, not a complete internal-state snapshot. Configured checks differ across tasks.

A.3 External Scoring and Episode Recordings

The server constructs a scorer from the scenario's registered implementation and calibration. Scorers operate on captured game images, using OCR for text and numbers or image rules for health bars and result screens. This is distinct from querying a game-engine state oracle, asking the coding agent to self-report progress, or using an agent to judge its own output.

Capture, stop, and settle. During gameplay, the environment queues scoring frames in a disk-backed stream using lossless PNG encoding. Episode video recordings provide a separate replay artifact; the scorer does not depend on decoding that compressed video. Once the episode stops, the server settles its score from the retained frames. For trajectory-based scorers, it resets the scorer and processes frames in temporal order:

$$S_g = \sum_{i=1}^{n} \Delta_g(f_i; z_{i-1}), \tag{4}$$

where f_i is a retained frame and z_{i-1} is the scorer's accumulated state, such as previous valid readings or confirmed milestones. The increments are task-specific and can be negative, as in health-difference or route-distance scoring. They are not uniformly rewards for each observed frame.

Snapshot-based scorers instead select a valid retained frame and apply their registered score function to that observation. OCR configurations specify regions, language, and confidence thresholds, while stateful rules reject implausible changes or require confirmation.

Result validity and audit records. Settlement occurs after gameplay is halted; its completion does not imply victory. The server records the outcome, consumed resources, termination reason, and available replay and scoring evidence. Scoring, sampling, or cleanup errors can invalidate a result: the public score is then null, with a partial value retained separately where available. Invalid or missing scores are excluded from score aggregation, and the corresponding episodes are not rerun. A missing score is not converted to a valid zero; mean scores are computed over the available valid scores. Online lifecycle checks and final score computation are separate; a visual terminal screen is not universally an automatic episode-stop condition. The retained logs and recordings provide an audit trail for inspecting execution failures and scoring readings.

B GAME TASKS AND SCORING

Table 6 provides the chapter overview: task objectives, scoring criteria, and interaction demands across the working game suite.

Game	Evaluation Task	Episode Scoring	Capabilities Evaluated
Combat & Action Timing			
Black Myth: Wukong (BMW)	Fight the Black Bear Guai boss and deplete its health	More boss health depleted earns a higher score	Close combat; dodge timing and boss adaptation
Sekiro: Shadows Die Twice (SEK)	Defeat Gyoubu Oniwa in a two-phase boss fight	Progress through both health phases earns points; full victory earns the maximum	Deflection and posture control; phase adaptation
The King of Fighters '98 (KOF98)	Fight a computer-controlled opponent and deplete its health	More opponent health depleted earns a higher score	Real-time fighting; spacing, defense, and attacks
Strategy & Resource Planning			
Balatro (BAL)	Build scoring card hands to advance through rounds	Stronger hands and later rounds earn more points	Hand construction; discard, economy, and deck planning
Slay the Spire (StS)	Play from the start of a run and climb as many floors as possible	Deeper floors earn higher scores	Turn-based combat; route, deck, and relic planning
Kingdom Rush 5: Alliance (KR5)	Defend a fixed level against successive enemy waves	Surviving more waves and preserving lives raise the score; victory adds a bonus	Tower placement and upgrades; wave planning
3D Sports & Precision Control			
Forza Horizon 5 (FH5)	Drive from a fixed start toward a designated destination	Getting closer to the destination earns more points	Road navigation; steering, throttle, and recovery
Riders Republic (RR)	Perform ski tricks to earn as many points as possible	Successful tricks increase the in-game score	Spatial control; timing and trick execution
NBA 2K25 (NBA25)	Score and defend in a 1,000-tic Blacktop one-on-one episode	+15 per own point; -5 per opponent point; no active victory bonus	3D basketball; movement, shooting, and defense
Movement & Timing			
TrackMania Nations Forever (TM)	Race the A02 course, reaching the finish or going as far as possible	Distance rewards unfinished runs; faster finishes earn more	High-speed control and trajectory optimization
Brotato (BRO)	Survive successive enemy waves and collect materials	Later waves drive the score; collected materials add points	Movement and combat; upgrade and resource allocation
BattleBlock Theater (BBT)	Collect gems to unlock the exit and complete a platforming level	Gems reward progress; finishing adds speed and yarn bonuses	Platforming; jump timing and route finding
Multi-step Task Execution			
Overcooked! 2 (OC2)	Prepare and serve orders by switching between two chefs	The final kitchen score rewards completed orders; episodes that do not complete the task receive no score, even if they end normally	Order planning; chef switching and coordination
Aimlab 2.0 (AIM)	Shoot as many blue targets as possible in 30-s Gridshot 3x3 by whoisaqua; three targets are visible, with each hit target replaced	One point per target destroyed; misses earn zero. OCR reads the final target-kill count from the results screen	Visual target localization; precise aiming and rapid shooting
Palworld (PAL)	Explore a dungeon and capture two target creatures	Captures earn points; early captures add bonuses; wasted balls incur penalties	Exploration; combat and capture planning

Table 6: **Games and evaluation tasks.** The 15 games are grouped by their principal interaction demands, with three games in each category. For each task, we summarize its objective, episode-scoring criterion, and required capabilities.

C CASE STUDIES

These cases analyze how coding agents revise perception, feedback, and control logic before selecting a frozen program. G levels describe the use of game information, and modification-depth badges describe changes between adjacent displayed archives using the rubric in Section 3.3; no aggregate IMS is computed for these illustrations. Screenshots provide context for the program analysis.

C.1 SLAY THE SPIRE: SELECTING AN EARLIER PROGRAM

Program selection need not favor the latest revision. Figure 6 follows seven revisions from a minimal fallback through visual combat control, map probing, and room handling. After further revisions and retests, the agent selected the earlier v4 archive (`79f0903e8ef7`).

The map routine makes the role of feedback concrete. It probes candidate nodes, checks for a visual response, waits for a transition, and verifies that the map has disappeared before treating a click as successful. Generic room handling similarly rotates candidate clicks after unsuccessful attempts and reclassifies the resulting screen. These checks encode a reusable interaction procedure: issuing an input alone is insufficient to establish that the intended transition occurred. The later attack-priority, probe-cursor, and turn-gating revisions modify different parts of this procedure.

The return to v4 separates code revision from artifact selection. Additional edits did not compel selection of the newest program. The selected artifact preserves one particular combination of combat, map, and room-handling routines from the development trajectory.

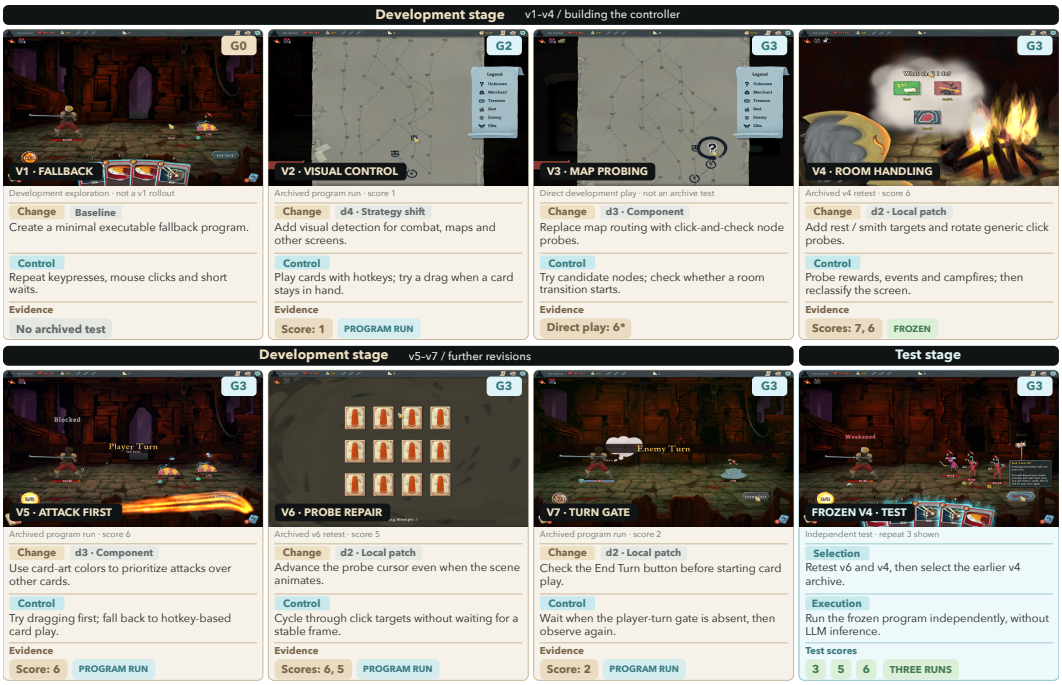

Figure 6: **Program development and selection in *Slay the Spire*.** Seven revisions led to frozen v4. G labels indicate grounding; Change badges apply the IMS depth rubric to adjacent archives: d2, local patch; d3, component change; d4, strategy shift. v1 is the baseline. Scores are individual runs; v1 shows exploration, and v3's starred score is from direct play. v2 groups two snapshots.

C.2 FORZA HORIZON 5: REVISING PERCEPTION AND RECOVERY

The Claude Opus 5 driver turns minimap observations into steering and recovery decisions. Figure 7 shows revisions to the signals used by these decisions, followed by selection of v10 (`d5035a5b764b`) after a v11 lookahead experiment.

Two changes address different sources of ambiguity. Main-view frame differences can reflect scene animation as well as vehicle movement; minimap displacement provides a more specific motion signal for the implemented stall detector. Separately, radial groups of route pixels can mix nearby portions of a winding route. Connected-route tracing instead orders pixels by distance along the detected route before estimating near and far bearings. These are changes to the information used for control, while v11's altered bearing weights change how the same estimates influence steering.

The trajectory also includes reversals in design: v8 restores the earlier control law while retaining minimap-based stall detection, and the final selection returns from v11 to v10. This illustrates development as revising and recombining executable components.

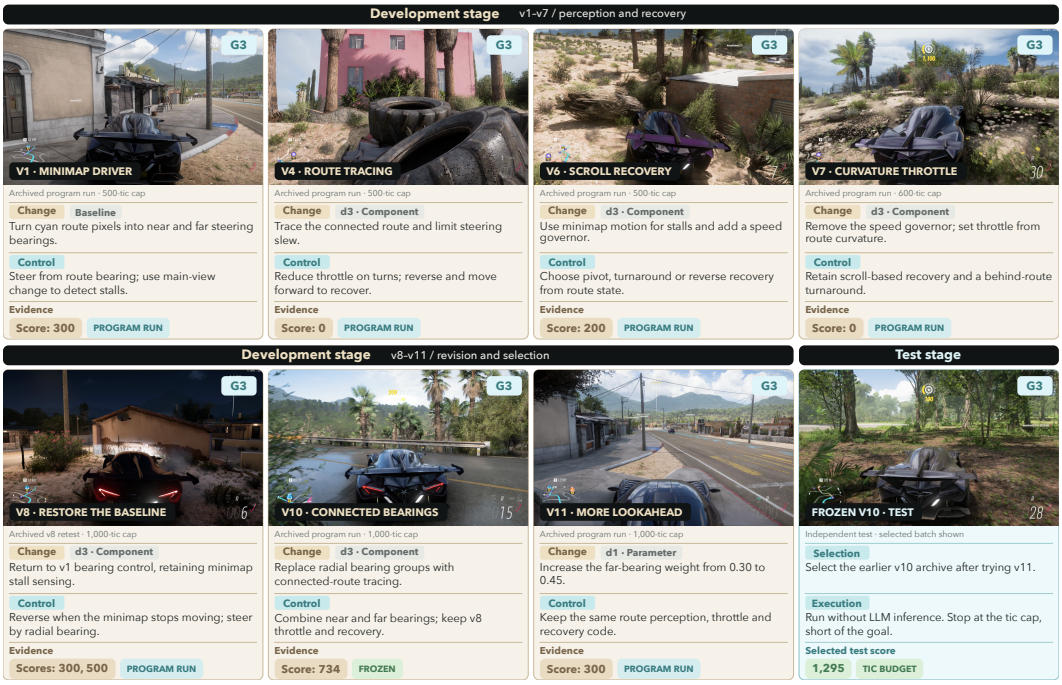

Figure 7: **Program development and selection in *Forza Horizon 5*.** Seven selected revisions illustrate changes to route perception, stall detection, and steering before selection of frozen v10; v9's throttle experiment is omitted. v8 retains minimap-based stall detection while restoring the earlier control law. The final selection returns from the v11 lookahead experiment to v10.

C.3 BattleBlock Theater: Revising the Signal for Progress

A controller must distinguish visible motion from useful progress. In Figure 8, the Claude Opus 5 program moves from fixed traversal to camera-motion feedback and persistent recovery state. The agent selects v5 (`f4d89e0b7753`) as the frozen program.

The transition from v4 to v5 exposes a specific interaction between recovery rules. In v4, vertical motion can reduce the same stall counter that eventually triggers direction reversal. Repeated jumping can therefore postpone reversal even without horizontal advance. v5 separates these signals: vertical motion still affects local recovery, but a distinct dead-end counter accumulates absent horizontal progress and reverses direction after 16 observations. The code change preserves the existing jump and back-off routines while changing when the controller abandons its current direction.

This revision makes horizontal progress a separate control signal, so vertical movement no longer prevents the controller from abandoning a blocked direction. The budget-guard revision separately accounts for execution limits. Development and independent testing use a per-episode limit of 3,000 game tics.

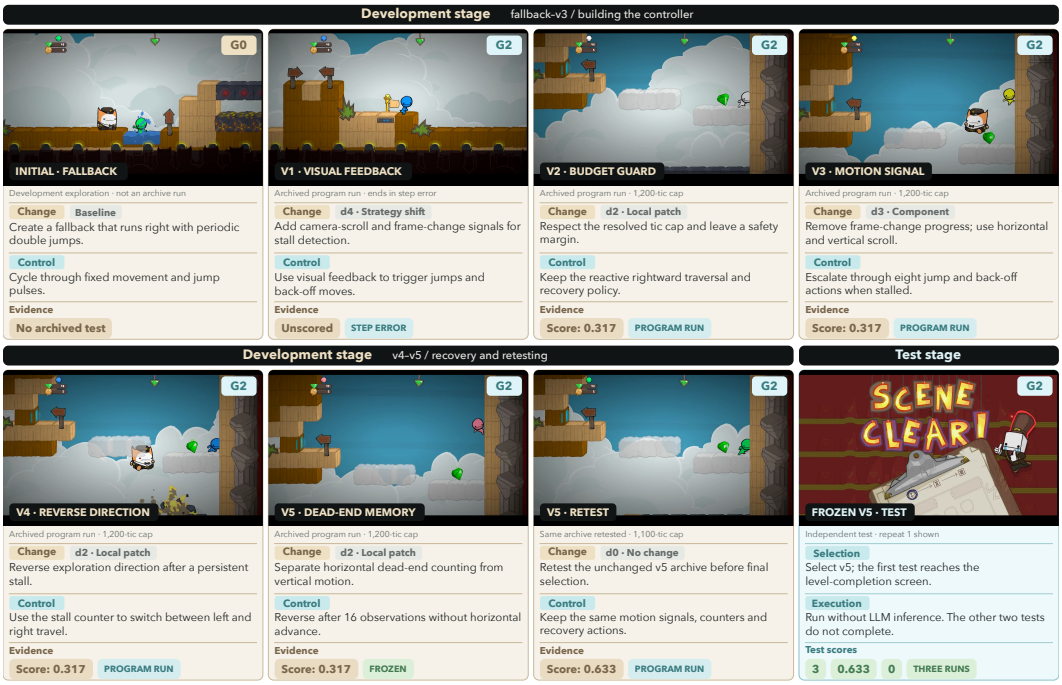

Figure 8: **Program development and testing in *BattleBlock Theater*.** The initial screenshot shows development exploration; subsequent panels illustrate program runs. Revisions add visual feedback, budget handling, motion signals, and persistent recovery state before selection of v5. Development and independent testing use a 3,000-game-tic episode limit.

C.4 BLACK MYTH: WUKONG: REVISING FEEDBACK AND ACTION TIMING

The Qwen3.8 Flash controller illustrates how coding makes state interpretation and action timing explicit. Figure 9 traces HUD-based combat control, healing feedback, and opening-sequence revisions. After testing later variants, the agent selected v8 (`6a2980c43be9`).

The revisions address both what the controller observes and when it observes again. v2 retains prior bar estimates when the HUD is absent instead of treating missing pixels as empty resources. v3 checks subsequent health readings to limit unsuccessful healing retries, while v8 inserts waits after healing inputs. These rules encode a distinction between issuing an action and observing its effect, although retaining old estimates can also leave the controller with stale state. v9 changes another aspect of feedback: batching three attacks postpones the next observation until the burst finishes.

The opening gate exposes a concrete control-flow issue. In v10, initial lock acquisition precedes the new gate; v11 moves that branch below it so opening actions occur first. This repair changes executable behavior even though both versions describe an opening gate. The final return to v8 shows that additional revisions need not determine the submitted artifact.

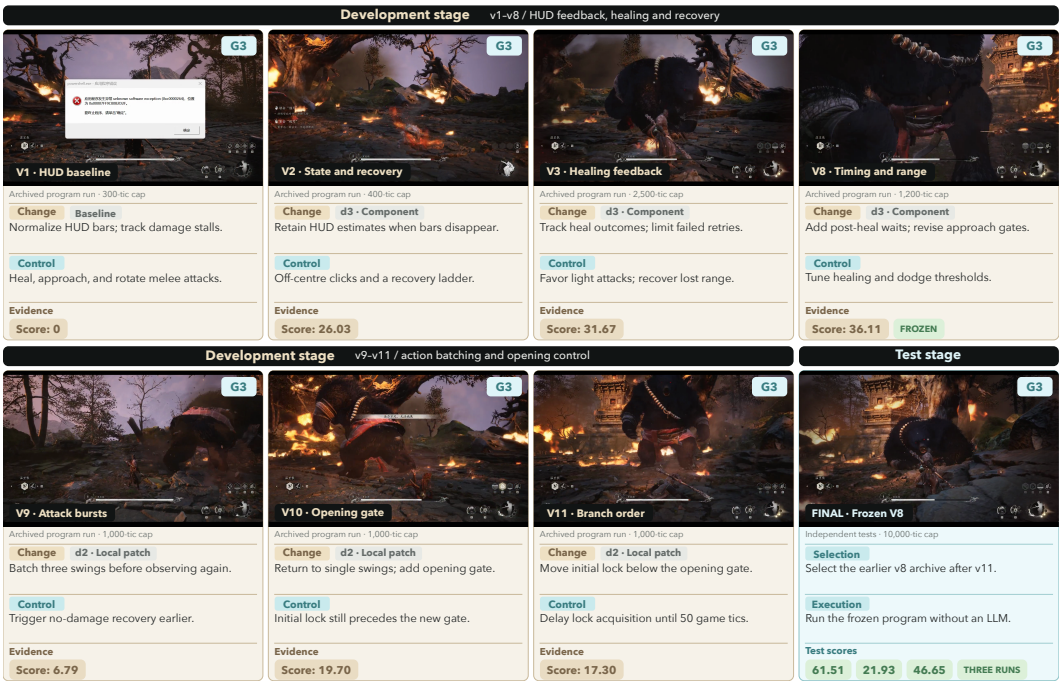

Figure 9: **Program development and selection in *Black Myth: Wukong*.** Seven selected archives illustrate revisions to HUD interpretation, healing feedback, action timing, and opening control flow; v4, v5, and v7 are omitted. The agent selects v8 after exploring later variants. The final panel shows execution of the selected frozen program.