GAMECODE-BENCH: BENCHMARKING REPOSITORY-SCALE GAME DEVELOPMENT WITH CODING AGENTS

Anonymous authors
Paper under double-blind review

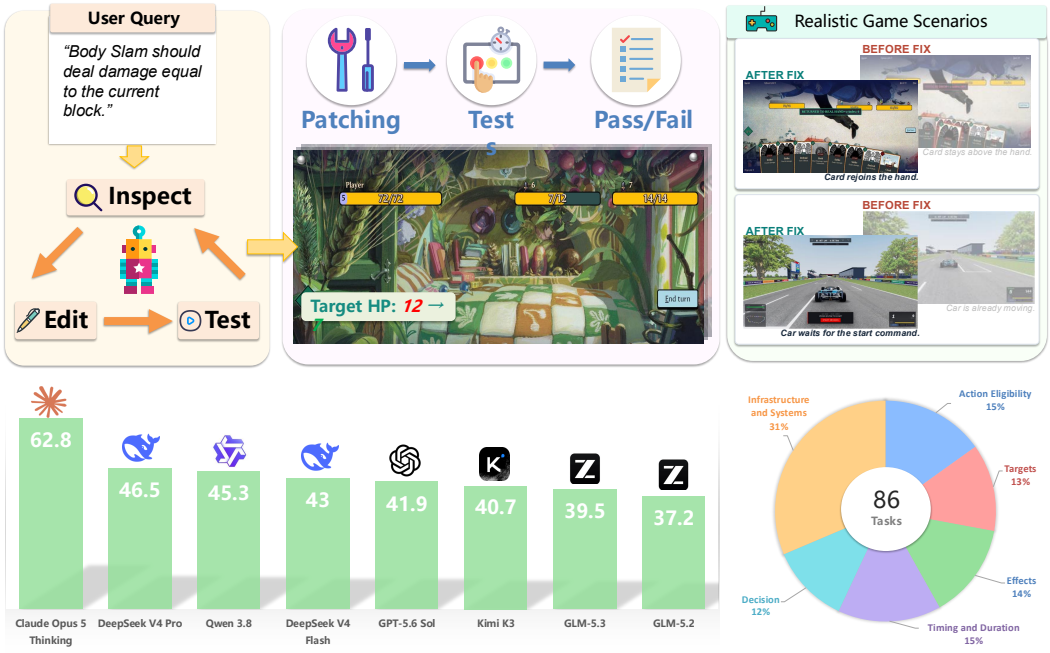

Figure 1: **GameCode-Bench at a glance.** Agents modify mature game repositories from natural-language development requests and are evaluated with hidden behavioral scenarios. Bottom panels show task composition and performance across eight evaluated models.

ABSTRACT

As large language models rapidly evolve, the capabilities of autonomous coding agents are advancing to new limits. Recent software engineering benchmarks, however, primarily evaluate conventional software projects, while game benchmarks focus on generating miniature, tutorial, or demo-level games. Agents' capacity to handle mature, highly interactive game systems, therefore, remains largely underexplored. To address this gap, we introduce **GameCode-Bench**, a benchmark for repository-scale gameplay engineering comprising 86 tasks across eight independent, open-source browser game repositories. Our benchmark captures complementary developer- and community-facing development workflows, and agents operate directly within mature codebases containing a median of 162,055 non-test source lines, with the largest exceeding one million. Alongside frontend, backend, and API integration maintenance tasks, GameCode-Bench centers on core game logic and challenges models to localize relevant implementation paths, infer intended behavior, and modify existing systems while preserving surrounding functionality. Across eight evaluated models, we find that failures concentrate in state integration, conditional logic, and incomplete behavioral branches; trajectory analysis further shows that additional exploration or testing is beneficial only when accurately targeted. Evaluated against hidden, deterministic, human-reviewed scoreable variants, GameCode-Bench provides a rigorous

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.

testbed for measuring and diagnosing coding-agent capabilities in complex game development.

1 INTRODUCTION

The rapid advancement of large language models has enabled autonomous coding agents to tackle increasingly complex repository-scale software engineering tasks. Yet existing software engineering benchmarks predominantly focus on conventional applications, utilities, and libraries, leaving game development comparatively underexplored. Meanwhile, recent game-oriented benchmarks largely evaluate generation from scratch (Luo et al., 2026; Jiang et al., 2026), tutorial-derived development (Chi et al., 2026), or isolated scripting tasks. These settings capture important aspects of game creation, but not the challenge faced by developers working inside mature games: understanding a large historically evolved codebase and modifying its behavior without violating the interconnected rules, state transitions, and event lifecycles on which the game depends.

This distinction is essential. While bugs are often forgivable in isolated demos, production games depend entirely on the consistent execution of rules, physics, and interconnected state over time. A seemingly isolated modification can ripple through input processing, game state, rendering, networking, and backend persistence, severely disrupting the player experience (Santos et al., 2018; Lewis et al., 2010; Hunicke et al., 2004). Consequently, benchmarking agents in this setting requires more than larger codebases: tasks must mirror authentic developer workflows, and evaluation must test observable behavior rather than dictate implementation.

We therefore introduce **GameCode-Bench**, a benchmark of 86 historically grounded development tasks drawn from eight mature, open-source browser game repositories. Given a natural-language development request, an agent must work directly within the complete repository, with a median non-test source size of 162,055.50 lines and the largest codebase exceeding one million lines. Tasks are recovered from real repository artifacts and code evolution rather than synthetic bug injection, and 59 tasks (68.6%) directly target gameplay mechanics and state coordination with manually audited behavioral tests designed to verify runtime functionality while permitting diverse implementations. We further evaluate eight open- and closed-source models under varying reasoning-effort settings and conduct in-depth trajectory-level analyses to characterize the challenges that emerge in realistic game development. Our analysis reveals substantial behavioral differences in how models reason about temporal ordering, state ownership and provenance, multi-source interactions, and cross-phase lifecycle integration. GameCode-Bench therefore serves not only as a measure of development capability, but also as a diagnostic testbed for understanding how different agent strategies succeed or fail on realistic game-engineering workloads.

Our contributions are fourfold:

Repository-scale game development. We introduce GameCode-Bench, a benchmark for evaluating coding agents on complete, mature game repositories. Its tasks require autonomous localization and reasoning across interconnected gameplay and software subsystems, extending evaluation beyond greenfield generation and tutorial-scale projects.

Historically grounded task construction. We develop complementary pipelines that recover engineering tasks from real issues, tests, commits, and code evolution. These pipelines support task construction when development records lack either a task specification or suitable behavioral tests.

Behavioral, implementation-agnostic evaluation. We provide 86 manually audited tasks with 102 hidden evaluator variants that assess agents' solutions through observable gameplay and system behavior while accommodating different valid implementations of the requested functionality.

Cross-model performance and behavioral analysis. We evaluate eight open- and closed-source models and analyze their solution trajectories, examining accuracy–efficiency trade-offs, solution strategies, and recurring failure patterns in repository-scale game development.

Table 1: Comparison of task settings and evaluation protocols. GameCode-Bench combines historically grounded modifications to existing games with predefined executable evaluation. ✓ indicates a property of the reported benchmark protocol; ✗ indicates that the protocol does not provide that property.

Benchmark	Game development	Existing project edits	Historical repo tasks	Fixed behavioral tests	Deterministic scoring	Tasks
SWE-bench (Jimenez et al., 2024)	✗	✓	✓	✓	✓	2,294
OpenGame-Bench (Jiang et al., 2026)	✓	✗	✗	✗	✗	150
GameCraft-Bench (Luo et al., 2026)	✓	✗	✗	✗	✗	140
WebGameBench (Zhang et al., 2026)	✓	✗	✗	✗	✗	111
GameDevBench (Chi et al., 2026)	✓	✓	✗	✓	✓	333
GameXpert–GameGen (Chen et al., 2026)	✓	✗	✗	✗	✗	97
GameXpert–GameFix (Chen et al., 2026)	✓	✓	✗	✓	✓	100
GameXpert–GameOpt (Chen et al., 2026)	✓	✓	✗	✗	✗	17
GameEngineBench (La et al., 2026)	✓	✓	✗	✓	✗	110
GameCode-Bench (ours)	✓	✓	✓	✓	✓	**86**

2 Related Works

From Game Generation to Real-World Maintenance. With recent advancements in tool-using coding agents (Yang et al., 2024), game development has emerged as a premier frontier for evaluating autonomous software engineering. Existing benchmarks predominantly center on greenfield, end-to-end generation from natural-language prompts. For instance, OpenGame-Bench (Jiang et al., 2026) targets browser-game generation, GameCraft-Bench (Luo et al., 2026) synthesizes Godot projects, and WebGameBench (Zhang et al., 2026) assesses live browser-game requirements. Other efforts move toward scoped development: GameDevBench (Chi et al., 2026) derives implementation tasks from online tutorials, while GameXpert-Bench (Chen et al., 2026) investigates multi-turn refinement and synthetic bug injection. However, while state-of-the-art agents demonstrate remarkable proficiency at scaffolding self-contained, demo-scale games from scratch, it remains an open question whether these abilities transfer to authentic game maintenance and complex feature development. GameCode-Bench bridges this gap by shifting the benchmark paradigm from greenfield generation to repository-grounded engineering, evaluating gameplay implementation and maintenance tasks reconstructed from historical repository artifacts, including issues, pull requests, commits, and developer-authored tests from mature, independently developed games.

Repository-Grounded Game Engineering. The closest comparator to our work is GameEngineBench (La et al., 2026), which evaluates scoped C++ tasks in Unreal Engine via synthesized tasks and pre-scoped editable file lists. GameCode-Bench differs in three respects. First, *task provenance and localization*: we reconstruct tasks from historical development artifacts and require agents to investigate repositories without pre-scoped editable-file lists. Second, *development environment*: our browser-based setting examines gameplay engineering without requiring Unreal-specific compilation and editor workflows. This complements engine-integrated evaluation by testing whether agents can correctly modify mature games within the broader browser-game ecosystem. Third, *behavioral grounding*: our tests primarily exercise behavior rather than inspect source patterns. Our audit (protocol in Appendix A.2) finds that 41.2% of the synthesized tests inspected in GameEngineBench rely exclusively on source inspection. Such checks can over-constrain valid implementations when they encode choices not required by the task. We instead emphasize checking the requested behavior across game states and lifecycle phases.

Dynamic Behavioral Evaluation. A central challenge in game benchmarking lies in designing fair, robust, and reproducible evaluation harnesses. Several recent benchmarks rely on multimodal judges and rubric-based visual evaluation: Cookie-Bench (Yang et al., 2026) scores aesthetics and functionality via agent-driven interaction, GameCraft-Bench (Luo et al., 2026) grades gameplay replays with multimodal LLMs, and GameGen-Verifier (Jia et al., 2026) pairs programmatic assertions with vision-language model (VLM) inspectors. While flexible, rubric- and VLM-based evaluations

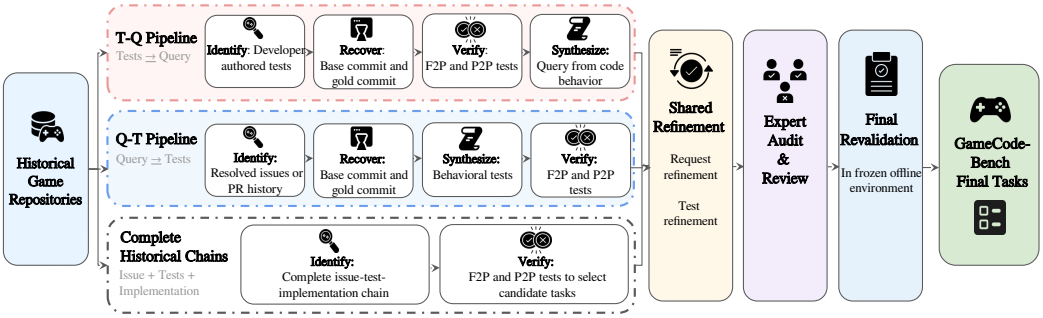

Figure 2: Task Pipeline. The benchmark is constructed through three provenance routes grounded in real development artifacts.

are bounded by non-deterministic judgments, perceptual hallucinations, and evaluator bias. Following the rigorous test-driven paradigm of SWE-bench (Jimenez et al., 2024), GameCode-Bench exclusively employs deterministic, executable behavioral test suites. To avoid over-constraining solutions while ensuring strict requirement coverage, our curation pipeline couples fail-to-pass execution validation with iterative automated review and manual verification of task–test alignment.

3 Benchmark Design

We formulate each benchmark task as a tuple $\tau = (R, b_0, q, g, t)$, where R denotes the repository, b_0 is the exact pre-change base commit, q is the natural-language development request visible to the solver, g represents the ground-truth gold patch, and t is the held-out evaluation test patch. During evaluation, an autonomous agent is provided solely with the repository R checked out at commit b_0 and the natural-language specification q; both the gold patch g and the test suite t remain hidden.

The Scarcity of Complete Artifact Chains. Repository-level software engineering benchmarks typically mine tasks by identifying historical instances with an unbroken chain: an explicit issue description, an associated pull request, the merged implementation patch, and an authored regression test suite. While viable in mature enterprise software with strict continuous integration practices, such complete chains are exceptionally scarce in open-source game repositories. These codebases are predominantly maintained by small teams or decentralized enthusiast communities, where game logic, balancing tweaks, and bug fixes are routinely committed without formal issue tracking, or where comprehensive test suites are added retrospectively without explicit bug reports. Rather than resorting to synthetic bug injection, we recover historically grounded tasks through two complementary, artifact-grounded curation pipelines: *Test-to-Query* (T-Q) and *Query-to-Test* (Q-T).

Pipeline Overview. As illustrated in Figure 2, the two pipelines address complementary forms of missing historical artifacts, capturing distinct realities of game maintenance:

- **T-Q (Internal Developer Workflows):** Anchored on developer-authored tests and commits, T-Q recovers the missing task specification (q). This route reflects internal development practices such as proactive refactoring, engine upgrades, and test-driven feature additions that lack standalone issue tickets.
- **Q-T (Community Feedback Loops):** Anchored on community issue reports and merged implementation diffs, Q-T recovers the missing behavioral test harness (t). This route reflects user-facing maintenance, player bug reports, and rule dispute resolutions that initially shipped without dedicated unit or regression tests.

Both pipelines, alongside complete-chain candidates, traverse four sequential stages: *Mining*, *Synthesis*, *Refinement*, and *Review*. While the first two stages vary with artifact availability, the final two apply shared refinement procedures and manual expert audits to retain challenging tasks and ensure consistent behavioral standards across the benchmark. Figure 2 provides a simplified overview of the construction process, while the full step-by-step procedures are detailed in Appendix A.1.

Table 2: Task provenance and composition of GameCode-Bench. The upper panel distinguishes the artifact provenance of each construction route. The lower panel reports the benchmark's primary-category distribution.

	Artifact provenance			Composition	
Pipeline	**Issue / query**	**Code change**	**Regression test**	**Tasks**	**Share**
Test-Query (T-Q)	**Synthesized**	Historical	Historical	74	86.0%
Query-Test (Q-T)	Historical	Historical	**Synthesized**	9	10.5%
Complete chain	Historical	Historical	Historical	3	3.5%
Primary category				**Tasks**	**Share**
Game Logic and Mechanics				59	68.6%
Game Infrastructure and Systems				27	31.4%
Total				**86**	**100.0%**

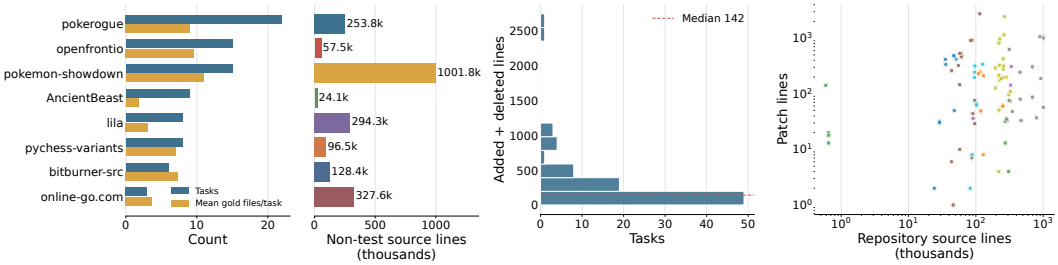

Figure 3: Repository and task-scale characteristics of GameCode-Bench. From left to right: task distribution and mean gold-modified files by repository, non-test source-code size, patch-size distribution, and patch size relative to repository size.

3.1 Benchmark Breakdown and Categorization

To comprehensively benchmark browser game maintenance, *GameCode-Bench* spans two complementary dimensions: **Game Logic and Mechanics** and **Game Infrastructure and Systems** (Table 2). The majority of tasks (68.6%, $n = 59$) focus directly on core game logic, requiring agents to reason over complex state transitions, rule evaluations, and domain-specific entities. The remaining 31.4% ($n = 27$) cover essential infrastructure, including UI rendering, data serialization, state persistence, network protocols, and build tooling. This dual composition ensures agents are evaluated on both domain-specific gameplay systems and standard software engineering practices.

To systematically assess core in-game systems, the 59 game logic tasks are distributed across five functional categories (showcased in Figure 1): **Action Eligibility** (22.0%), which evaluates prerequisite execution rules and move validation; **Targets** (18.6%), focusing on entity resolution, range constraints, and valid selection criteria; **Effects** (20.3%), addressing state mutations, attribute updates, and status application; **Timing and Duration** (22.0%), covering event sequencing, turn lifecycles, and cooldown decay; and **Decisions** (16.9%), involving rule-evaluation trees, branching logic, and automated game flow arbitration.

4 Evaluation and Results

4.1 Evaluation Setup

We evaluate eight state-of-the-art autonomous coding agents across the 86 source tasks in our benchmark. For each task, agents are supplied with the repository checked out at the pre-change base commit b_0 alongside our standardized natural-language specification. This query explicitly delineates: (i) the high-level objective, (ii) concrete behavioral requirements, (iii) edge cases and exceptions, and (iv) backward-compatibility and lifecycle invariants. All evaluations execute within isolated, network-disabled container environments to eliminate external search leakage. Furthermore, repos-

Table 3: Task-level results and efficiency statistics for the low- and maximum-reasoning settings. "–" indicates that the model was not scheduled or not available in that effort campaign at the time.

Model	Low						Max					
	Resolved	PASS@1	Avg. steps	Solve time (min)	In (M)	Out (M)	Resolved	PASS@1	Avg. steps	Solve time (min)	In (M)	Out (M)
Claude Opus 5	**33/86**	**38.4%**	39.2	27.3	1.12	0.007	**54/86**	**62.8%**	87.6	63.7	7.73	0.029
DeepSeek v4 Pro	32/86	37.2%	87.7	21.2	5.19	0.041	40/86	46.5%	96.9	65.8	8.54	0.053
Qwen 3.8	8/86	9.3%	62.9	30.1	2.06	0.030	39/86	45.3%	89.0	82.1	6.86	0.076
DeepSeek v4 Flash	20/86	23.3%	89.3	45.7	7.68	0.081	37/86	43.0%	108.6	47.5	8.68	0.064
GPT-5.6 Sol	26/86	30.2%	11.9	2.0	0.15	0.003	36/86	41.9%	20.8	6.9	0.58	0.007
Kimi K3	–	–	–	–	–	–	35/86	40.7%	83.1	121.5	6.71	0.059
GLM-5.3	24/86	27.9%	90.4	55.4	4.32	0.056	34/86	39.5%	94.9	55.3	4.39	0.061
GLM-5.2	–	–	–	–	–	–	32/86	37.2%	120.3	62.0	8.33	0.063

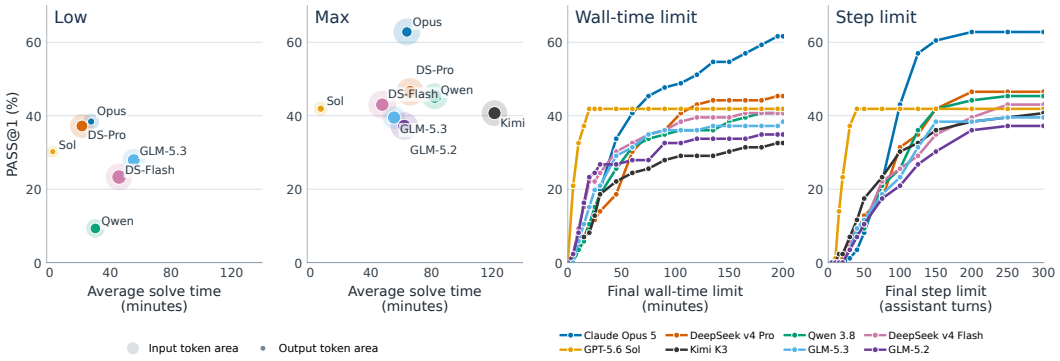

Figure 4: Performance and efficiency of evaluated models on GameCode-Bench. Left: PASS@1 versus average solve time under Low and Max effort; bubble areas encode input and output token usage. Right: retrospective PASS@1 under simulated wall-time and step limits, where a successful run is counted as resolved only if it completes within the corresponding budget.

itory git histories are pruned to the target base commit, preventing models from inspecting subsequent commits, merge histories, or pull request metadata. All runs use a maximum budget of 300 interaction steps with no wall-clock time limit. Solvers submit their finalized solution in the standard unified git diff patch format. To ensure fair, reproducible, and standardized execution, we use `mini-swe-agent` as our agent scaffold.

Evaluation Metrics and Reasoning Budgets. We report the PASS@1 resolution rate. A task is considered resolved if and only if the submitted patch applies cleanly and passes all assertions in the held-out evaluation test suite t. This ensures the implementation satisfies the requested gameplay changes while strictly preserving existing game rules, state transitions, and lifecycle invariants. Where supported by the model API, we assess performance across two reasoning effort regimes (low and max effort) to examine the impact of extended test-time computation.

4.2 Main Results

Table 3 shows that Claude Opus 5 leads in both low and max settings. Its 62.8% PASS@1 places it 16.3 percentage points ahead of the nearest competitor, DeepSeek v4 Pro, leaving the remaining models in a narrow band below the 50% threshold. Scaling from low to maximum reasoning effort yields uneven gains across the six models. Qwen 3.8 extracts the most value from increased compute, while Opus continues to widen its lead.

Token use and efficiency. GPT-5.6 Sol has the lowest reported average step count, completion time, and input and output token use in both settings. At maximum effort, it achieves 41.9% PASS@1 with 20.8 steps, 6.9 minutes, 0.58M input tokens, and 0.007M output tokens. Opus re-

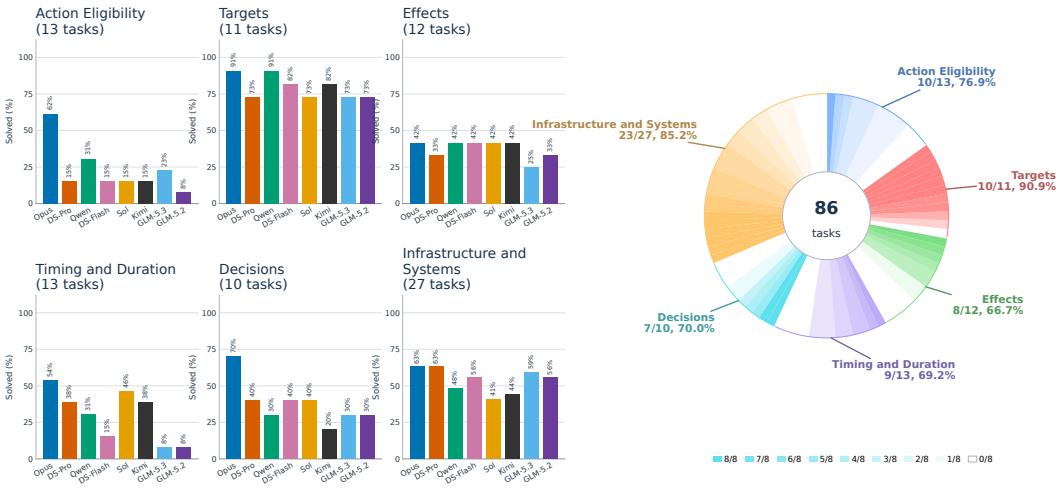

Figure 5: **Per-category performance on GameCode-Bench. Left:** Per-model resolution rates across six task categories. **Right:** Task-level coverage grouped by category, where each segment represents one task and darker shading indicates resolution by more of the eight evaluated models. Annotations show the number and percentage of tasks resolved by at least one model.

solves 18 more tasks, but uses approximately 13.3 times as many input tokens, 4.1 times as many output tokens, and 9.2 times as much time. Higher resource use, nevertheless, does not consistently correspond to higher accuracy across models. Kimi K3 takes nearly twice as long as Opus (121.5 versus 63.7 minutes), yet resolves 35 tasks (40.7%) compared with Opus's 54 (62.8%). Compared with DeepSeek v4 Pro, Opus also achieves a 16.3-point advantage while using fewer input tokens (7.73M versus 8.54M) and fewer output tokens (0.029M versus 0.053M). Nor does increasing reasoning effort necessarily increase token volume: DeepSeek v4 Flash improves from 20 to 37 resolved tasks while input tokens decrease from 7.68M to 6.88M and output tokens decrease from 0.081M to 0.064M. Its average step count rises from 89.3 to 108.6.

5 Analysis

In this section, we further analyze the benchmark results from three perspectives. First, we conduct a **Category Analysis** to evaluate performance across task types and highlight the most challenging aspects of game development. Second, we examine **Failure Modes** to identify recurring patterns in unsuccessful attempts in these categories. Finally, motivated by the 15.1% gap between the strongest individual model and cumulative coverage across all models (77.9%), we perform a **Behavioral Analysis** to explore how models differ in their approaches and what drives their strengths.

5.1 Categorical Analysis and Failure Modes

Figure 5 identifies a consistent set of strengths and bottlenecks. Targets has the highest success rate for every model, with a cross-model average of 79.5%, while Action Eligibility and Timing and Duration have the lowest averages, 23.1% and 29.8%. Effects is another shared bottleneck: its average is 37.5%, and no model resolves more than 5 of its 12 tasks. Overall, the oracle union covers 67/86 tasks (77.9%), adding 13 beyond Opus's 54 successes, and 19 tasks have no recorded success from any model.

The category-level contrast suggests that difficulty increases when a task requires several interacting rules to remain consistent. An easier tasks often involve a relatively local decision within a single game state. For example, a Target task that tests whether an action can reach a particular location, whether a destination is available, or which direction an attack may take. Their central behavior can therefore often be determined at one decision point.

The harder categories, by contrast, more frequently require behavior to remain correct across multiple characters, modifiers, or stages of execution. In **Effects** tasks, for instance, changing one

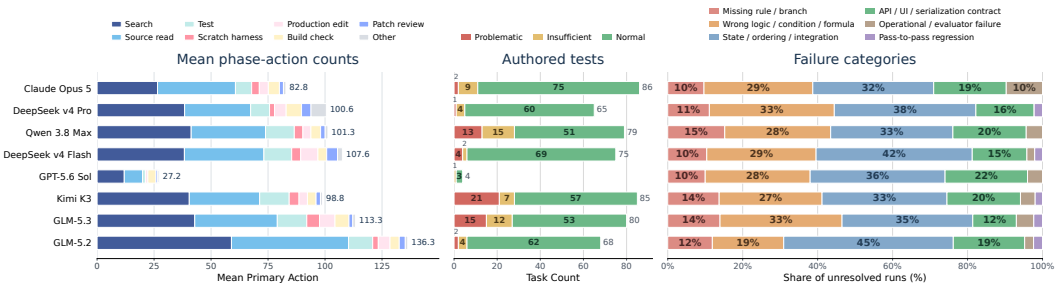

Figure 6: Maximum-effort trajectory and failure analysis. Left: mean primary actions in eight displayed phases over all trajectories per model. Center: task runs with authored tests or probes, partitioned by audit verdict. Right: failure categories as shares of each model's evaluated failures. Detailed annotation criteria is provided in Appendix A.3.

interaction can unintentionally alter another. In one task, all seven evaluated patches successfully repaired the requested interaction between one game effect and its modifiers, yet each broke another effect that had previously behaved correctly. **Action Eligibility** introduces a related challenge: a restriction must apply to the correct character and to the correct game state. In one task, an action that copies another character's ability must obey different restrictions depending on whose ability is being changed. Several patches implemented a single generic restriction, but missed an important class of abilities that should not be copied. The resulting implementation therefore appeared correct for common cases while remaining incomplete for a closely related branch. **Timing and Duration** extends this coordination problem across execution phases. In one task involving an action that unfolds over multiple turns, some patches consumed a limited resource too early, while another cleared the temporary action state during the wrong phase. Each implementation captured a plausible local rule, but placed it at the wrong point in the broader game lifecycle.

Across these examples in the most challenging categories, we see that implementing the requested behavior is insufficient: agents must determine where the rule belongs, which related cases it affects, and what existing behavior must remain unchanged. These burdens are also reflected in the right panel of Figure 6, where we systematically categorize task failures: the majority of failures (77.3%) arise from three primary categories: state, ordering, or integration issues (36.8%); flawed logic or conditions (28.4%); and missing rules or branches (12.1%), reinforcing that the main challenge lies in correctly placing, scoping, and completing behavior within the broader game system.

5.2 Agent Analysis

Having identified the primary failure modes, we next examine how models approach these tasks in practice. Since model successes overlap only partially rather than forming strict subsets, we analyze trajectory-level behavior to uncover the problem-solving patterns underlying their performance:

Compact repair vs. comprehensive exploration. As reported in Table 3, Sol averages only 27.2 actions across the eight displayed phases, compared with 82.8–136.3 for the other models. Its mean test and scratch-harness counts are 1.27 and 0.01, respectively, versus 7.12–12.95 and 2.20–5.58 for its peers. This large discrepancy between numerics reflects two vastly different task solving strategy. In Figure 6, we observe two broad behavioral patterns: GPT-5.6 Sol follows compact repair sequences, whereas the other models combine substantially more repository exploration.

Localization bottlenecks. Figure 7 shows how these strategies unfold. Sol adopts the most direct approach, typically moving from focused investigation to editing with minimal intermediate validation. In sharp contrast, GLM-5.2 invests heavy effort into locating and understanding the relevant code. It averages 58.9 search actions and 51.4 source reads (compared to 42.8 and 36.2 for GLM-5.3) and consumes nearly twice the input tokens (8.33M vs. 4.39M). In its eight runs that exhaust the step limit without producing a patch, 91.5% of normalized action time is spent searching and reading. For GLM-5.2, the primary bottleneck therefore appears to be discovery rather than premature editing: whereas Sol commits rapidly once a plausible intervention point is identified, GLM-5.2 often exhausts much of its budget simply trying to locate one.

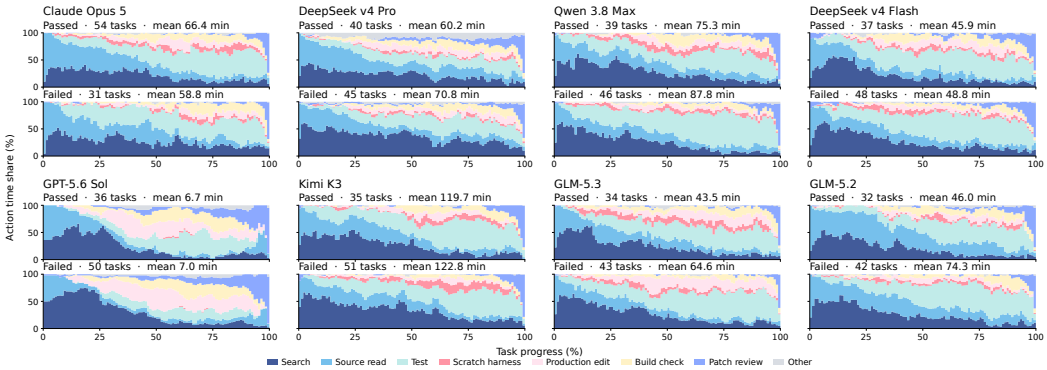

Figure 7: Mean action-time profiles. Evaluated passes above and failures below. Each task's duration is normalized to 100 progress bins and tasks are weighted equally; vertical shares sum to 100%.

Lifecycle-aware validation. This efficiency, however, comes with a different trade-off. Sol authors tests or runtime probes in only 4/86 runs, compared to 65–86 runs across the other evaluated models. Its concise trajectories offer little empirical validation that a locally plausible edit preserves correctness across subsequent phases or related game states. Opus offers an instructive counterpoint: it more consistently transitions from implementation to validation, and its authored checks span multiple stages of the game lifecycle. This broader validation is especially valuable on challenging tasks, where locally correct edits can still introduce regressions in downstream state transitions.

Validation frequency vs. validation quality. Yet increased validation does not uniformly improve outcomes. Across the seven non-Sol models, authored tests or probes appear in 89.4% of runs, but their diagnostic value varies widely. As shown in the center panel of Figure 6, flawed or insufficient checks are prevalent in Qwen (35.4%), GLM-5.3 (33.8%), and Kimi (32.9%), compared to only 12.8% for Opus and 7.7% for DeepSeek v4 Pro, a pattern that disproportionately burdens failed trajectories with test-related overhead and inflates solve times. Thus, the benefit of validation depends not simply on how often agents test, but on whether those checks faithfully capture the requested behavior and relevant lifecycle conditions.

Taken together, our analysis shows that performance differences are not explained by interaction volume alone. Excessive exploration can prevent an agent from reaching the correct intervention point, while frequent but poorly specified validation can add substantial cost without reducing behavioral errors. To succeed in realistic game development, agents need to localize the relevant code efficiently, reason about how the requested behavior propagates across the game lifecycle, and validate their implementations with targeted checks that faithfully capture the intended semantics.

6 Conclusion

We introduce **GameCode-Bench**, a benchmark for repository-scale game development comprising historically grounded tasks from mature open-source games and implementation-agnostic behavioral evaluation of repository-grounded development requests. Our evaluation and analysis show that while current models handle many localized changes reliably, modifications spanning multiple actors, states, and execution phases remain challenging. Failures are dominated by state and integration mismatches, faulty conditional logic, and unhandled branches, revealing persistent difficulty in preserving behavior across interconnected game systems, and neither extensive exploration nor frequent testing alone is sufficient for success. Effective agents must therefore localize relevant code efficiently, reason about how requested behavior propagates across the game lifecycle, and validate implementations with targeted checks that faithfully capture the intended behavior. By exposing these state-driven and architectural failure modes, **GameCode-Bench** establishes a critical foundation for building AI agents capable of tackling the unique engineering demands of real-world game development.

AI USE STATEMENT

In this work, we used GPT-5.6-Sol to assist with drafting task requests, generating supplementary tests, preliminarily categorizing failure cases, implementing code, and debugging. The primary author reviewed and verified all AI-assisted code. For manuscript preparation, we used AI tools for proofreading, language refinement, and LaTeX formatting. The authors take full responsibility for the final manuscript and all associated claims, code, tests, and other artifacts, including materials produced with AI assistance.

REPRODUCIBILITY STATEMENT

We make several efforts to support reproducibility. Section 3 describes the benchmark formulation, artifact provenance, and task-construction procedure, while Appendix A.1 provides the full stage-by-stage curation pipeline, including base- and gold-commit recovery, fail-to-pass verification, request and test refinement, expert audit, and final revalidation. Each task is tied to an exact repository and base commit, and evaluation uses deterministic held-out behavioral tests in frozen offline environments. Section 4.1 specifies the agent evaluation protocol, including repository preparation, network isolation, interaction budgets, reasoning-effort settings, evaluation criteria, and the mini-swe-agent scaffold used across models. Tables 2 and 3 further report the benchmark composition and model configurations needed to interpret the experiments. We will release the benchmark tasks, evaluation harnesses, environment specifications, and analysis artifacts once accepted to enable reproduction of both the benchmark evaluation and the reported analyses.

REFERENCES

Kun Chen, Haorong Hong, Peizhong Gao, Jianfeng Lin, Tongxu Luo, Yuxuan Xie, Chenxu Liu, Jieling He, Zhongyuan Liu, and Zeno Zeng. Gamexpert-bench: How far are coding agents from expert game development? *arXiv preprint arXiv:2608.21833*, 2026.

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

Robin Hunicke, Marc LeBlanc, Robert Zubek, et al. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, pp. 1722. San Jose, CA, 2004.

Chaobo Jia, Ruipeng Wan, Ting Sun, Weihao Tan, Borui Wan, Yuxuan Tong, Guangming Sheng, and Hong Xu. GameGen-Verifier: Parallel keypoint-based verification for LLM-generated games via runtime state injection. *arXiv preprint arXiv:2605.07442*, 2026. URL https://arxiv.org/abs/2605.07442.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Brian La, Sejoon Chang, Ben Kim, Junyoung Bae, Aamish Ahmad Beg, Sei Chang, Gonzalo Gonzalez-Pumariega, and Kanav Goyal. GameEngineBench: Evaluating coding agents on real C++ runtime environments. *arXiv preprint arXiv:2607.03525*, 2026. URL https://arxiv.org/abs/2607.03525.

Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. What went wrong: a taxonomy of video game bugs. In *Proceedings of the fifth international conference on the foundations of digital games*, pp. 108–115, 2010.

Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, Fan Bu, Wenyu Du, Xiaotong Zhang, Kyle Li, Shaobo Wang, Linfeng Zhang, Yuxuan Liu, Xin Lai, Chenxin Li, Yiduo Guo, Zhexin Zhang, Xinyuan Wang, Tianyi Bai, Ziniu Li, and Benyou Wang. GameCraft-Bench: Can agents build playable games end-to-end in a real game engine? *arXiv preprint arXiv:2606.17861*, 2026. URL https://arxiv.org/abs/2606.17861.

Ronnie ES Santos, Cleyton VC Magalhães, Luiz Fernando Capretz, Jorge S Correia-Neto, Fabio QB Da Silva, and Abdelrahman Saher. Computer games are serious business and so is their quality: particularities of software testing in game development from the perspective of practitioners. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10, 2018.

Haoyue Yang, Zhangxiao Shen, Fan Ding, Hangting Lou, Yifeng Kou, Haoqing Yu, Jingyao Li, Zhengfan Wu, Siqi Bao, Jing Liu, and Hua Wu. Cookie-Bench: Continuous on-screen key interaction evaluation for web generation. *arXiv preprint arXiv:2605.30000*, 2026. URL https://arxiv.org/abs/2605.30000.

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*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Wenyu Zhang, Guoliang You, Tianlun, Haotian Zhao, Tianshu Zhu, Haoran Wang, Xiaoxuan Tang, Mingyang Dai, Jingnan Gu, Daxiang Dong, and Jianmin Wu. WebGameBench: Requirement-to-application evaluation for coding agents via browser-native games. *arXiv preprint arXiv:2605.17637*, 2026. URL https://arxiv.org/abs/2605.17637.

APPENDIX

A Pipeline and Categorizations **13**

A.1 Task curation pipeline by stage and step 13

A.2 Audit of GameEngineBench evaluators. 13

A.3 Analysis Annotation Protocol 13

A.4 Sample Task 14

B Additional Result Visualizations **15**

B.1 Per task pass 15

A PIPELINE AND CATEGORIZATIONS

A.1 TASK CURATION PIPELINE BY STAGE AND STEP

Table 4: Task curation pipeline by stage and step. T-Q and Q-T recover missing specifications and behavioral tests, respectively. Refinement and review stages are jointly applied across both pipelines and complete-chain candidates.

Stage	Step	T-Q: Tests → Request	Q-T: Request → Tests
Mining	Source	Developer-authored test suites and the historical commits introducing or modifying them.	Community-resolved issue reports and their associated implementation commits.
	Base commit b_0	Scan upstream from the test commit's parent to identify the nearest uncontaminated base commit where tests fail.	Scan upstream from the parent of the initial issue commit to locate the nearest uncontaminated base commit.
	Gold commit b_g	Locate the downstream commit establishing passing test execution.	Extract the terminal issue-linked commit as the candidate gold implementation.
	Screen	Retain candidate pairs exhibiting reproducible fail-to-pass behavior without partial implementations.	Retain candidate pairs exhibiting coherent, issue-aligned changes between b_0 and b_g.
Synthesis	Generate	Synthesize dual-granularity specifications: a detailed test-focused query and a general feature-intent request.	Synthesize behavioral test suites from the request and b_0 checkout, strictly withholding the gold patch and post-change code.
	Check	Verify behavioral compatibility and scope consistency across both generated query variants.	Execute deterministic fail-to-pass validation: confirm test failure at b_0 and passing execution upon applying the gold patch.
	Repair	Iteratively revise under-specified, ambiguous, or misaligned query drafts.	Refine failing, flaky, or brittle test assertions and re-execute fail-to-pass validation.
	Filter	Discard candidates with persistent semantic divergence or unresolvable incoherence.	Discard candidates failing deterministic fail-to-pass validation or exhibiting environment flakiness.
Refinement	Request	Standardize all queries into a uniform schema: high-level architectural objective, concrete required runtime behaviors, edge cases and exceptions, and lifecycle/compatibility invariants; sanitize all implementation leaks.	
	Tests	Desensitize test assertions to evaluate observable runtime behavior and state transitions; eliminate AST-coupled checks, unstated requirements, and over-constrained private method assertions.	
Review	Audit	Conduct manual expert reviews to excise trivial tasks, under-specified requirements, non-deterministic mechanics, and tests that unfairly reject valid alternative architectures.	
	Revalidate	Execute end-to-end fail-to-pass verification on all finalized tasks in isolated, frozen offline environments.	

A.2 AUDIT OF GAMEENGINEBENCH EVALUATORS.

To characterize the evaluation style of the closest prior benchmark, we manually audited the synthesized tests released with GameEngineBenchLa et al. (2026). GPT-5.6 Sol was used to assist with an initial classification of each inspected test, after which the classifications were manually reviewed against the test implementation and the behavior it was intended to evaluate.

The unit of analysis is the individual synthesized test. We classify a test as *source-inspection only* when its verdict can be determined entirely from static properties of the submitted source code, without executing the relevant program behavior. This includes checks based on source-text matching, regular expressions, AST or structural inspection, the presence or absence of particular identifiers, functions, methods, imports, API calls, or prescribed implementation patterns. By contrast, a test is classified as *behavioral* when it executes the relevant implementation and determines correctness from observable runtime state, outputs, side effects, interactions, or lifecycle behavior. Tests that combine source inspection with substantive runtime behavioral assertions are not counted as exclusively source-inspection based.

Ambiguous cases were reviewed manually by examining both the evaluator and the corresponding task requirement, with classification determined by the evidence on which the test's pass/fail decision ultimately depends. Under this criterion, 41.2% of the synthesized tests we inspected in GameEngineBench rely exclusively on source inspection.

A.3 ANALYSIS ANNOTATION PROTOCOL

We audit maximum-effort agent trajectories to characterize both **failure modes** and the quality of **agent-authored validation**. For each run, GPT-5.6 Sol is first used to produce a preliminary categorization from the task specification, agent trajectory, submitted patch, and held-out evaluator

outcome. These preliminary labels are used only to assist the annotation process and are not treated as final. All reported labels are subsequently verified manually by four annotators familiar with the benchmark tasks. Each evaluated unresolved run is assigned a single primary failure category corresponding to the error most directly responsible for the failed behavioral requirement.

Failure categories. We use six mutually exclusive categories. *Missing rule / branch* denotes an omitted required case, exception, or execution path. *Wrong logic / condition / formula* captures implementations in which the relevant rule is present but its predicate, computation, or decision logic is incorrect. *State / ordering / integration* covers errors in state ownership or propagation, execution timing, lifecycle ordering, or interactions between otherwise plausible local changes. *API / UI / serialization contract* denotes violations of interfaces, rendered behavior, data formats, or other system-level contracts. *Pass-to-pass regression* captures changes that satisfy the target behavior but break previously correct behavior. Finally, *Operational / evaluator failure* covers runs whose outcome is primarily attributable to execution, environment, patch-application, or evaluation failures rather than the submitted implementation itself.

Agent-authored validation. For every run containing an agent-authored test, runtime probe, or scratch harness, we separately audit the validation artifact against the development request and the behavior it is intended to verify. During this audit, we inspect whether each check (i) exercises behavior reachable from the requested change, (ii) uses an oracle supported by the request, (iii) covers the relevant actor and state, (iv) respects lifecycle ordering, and (v) avoids introducing requirements absent from the task specification.

We classify authored validation into three categories. *Normal* denotes checks that are behaviorally valid and provide useful evidence for the relevant requirement. *Insufficient* denotes checks that are individually valid but too narrow to establish the requested behavior, for example because they omit relevant branches, actors, states, or lifecycle stages. *Problematic* denotes checks that provide misleading feedback because they encode an incorrect oracle, introduce an unstated requirement, exercise an unreachable or irrelevant condition, or otherwise conflict with the intended task behavior.

Annotation procedure. GPT-5.6 Sol provides the initial failure and validation-quality labels to support systematic review of the trajectories. Four annotators then independently inspect each preliminary label against the task specification, full trajectory, submitted patch, and held-out evaluator behavior, and either confirm or revise the label. When the two annotators disagree, the case is jointly reviewed and resolved through discussion. Cases for which no single category is immediately supported are revisited against the available evidence until a consensus label is reached. The resulting annotations underlie the failure-mode and authored-validation distributions reported in Figure 6.

A.4 Sample Task

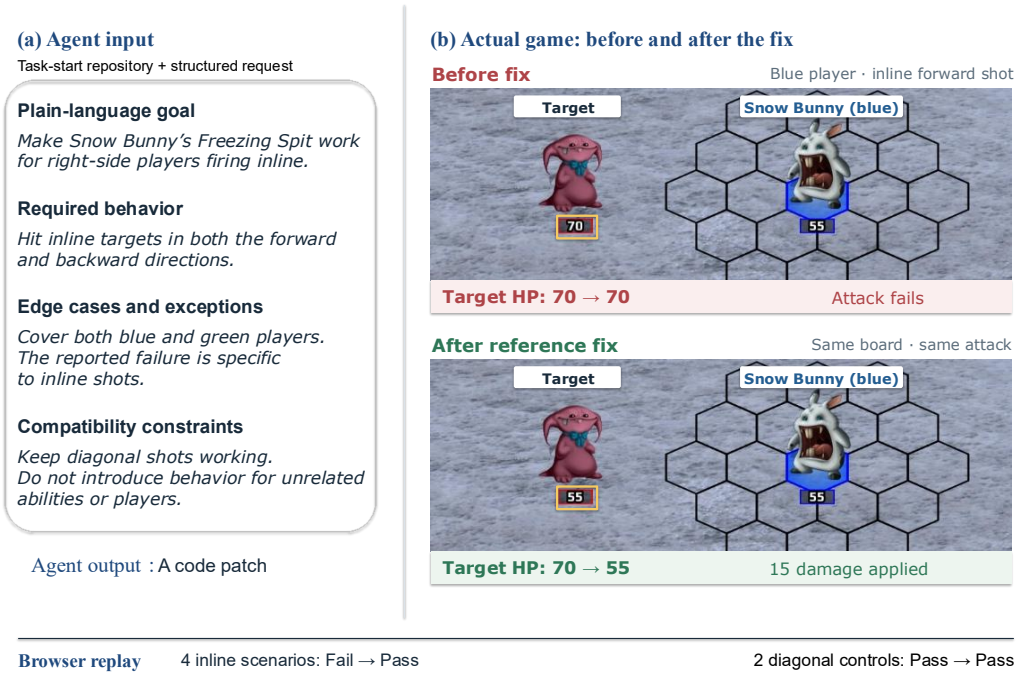

Figure 8: A sample task from Ancient Beast repository.

B Additional Result Visualizations

B.1 Per task pass

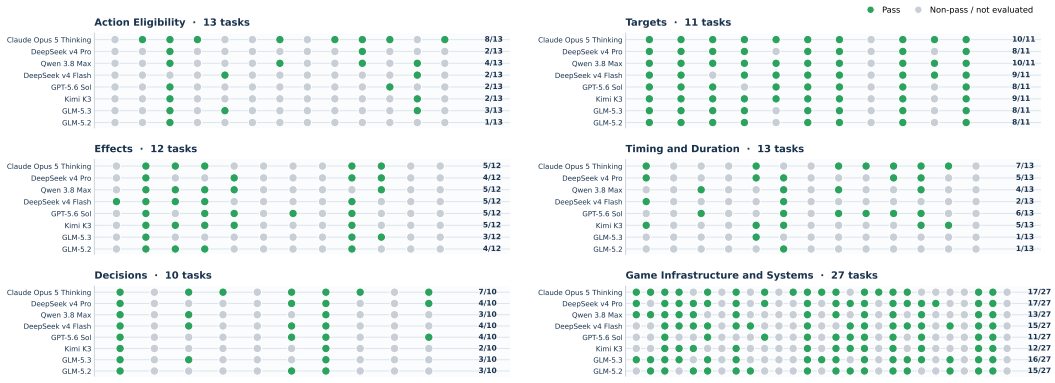

Figure 9: Results showing pass per model task.