GAMEMOD: EVALUATING SOFTWARE COMPLIANCE THROUGH GAME MODDING

Anonymous authors
Paper under double-blind review

ABSTRACT

Rapid advances have brought coding agents into everyday development workflows, yet they can still overlook existing components and compatibility constraints. Game modding provides a natural setting for evaluating this problem, as mods can be tested directly in the game runtime and alongside independently developed mods. To this end, we introduce GAMEMOD, a benchmark of 150 game-mod construction and repair tasks spanning 12 games for evaluating coding agents under these constraints. Our analysis shows that agents can break existing functionality when rewriting components, and that their fixes for reported bugs can cause previously compatible mods to stop working together. Moreover, even when an implemented mod passes the agent's own tests, it can still fail in the actual game, showing that the real runtime provides more rigorous compliance verification. By exposing these limitations, GAMEMOD provides concrete targets for improving generalist coding agents' ability to extend existing software while preserving compatibility, maintainability, and robustness. Our code, environments, scripts, and data are available at an anonymous repo.

1 INTRODUCTION

Coding agents have advanced at a remarkable pace, progressing from completing standalone functions (Chen et al., 2021) to resolving real-world GitHub issues (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025b), and they are now part of everyday development workflows (Robbes et al., 2026; Murphy-Hill et al., 2026). Yet everyday development rarely starts from scratch: most changes *extend* an existing system (Lehman, 1980). Beyond implementing the intended functionality, such a change should also *comply* with that system, i.e., run correctly together with the existing code and other independently developed components without introducing new bugs.

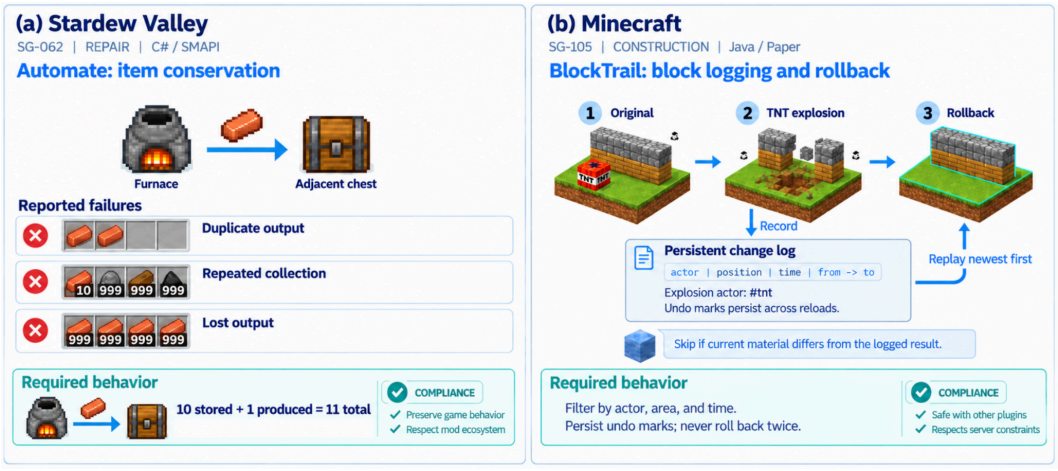


Figure 1: Two representative example tasks from GAMEMOD.

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.

Current coding agents sometimes fall short of this requirement, reimplementing components that already exist and neglecting how their changes interact with the code around them (Sonar, 2026; Ferdous et al., 2026). The costs of such changes are easy to miss at first but accumulate over time: open-source projects that adopted Cursor saw a short-lived gain in development velocity, followed by a lasting rise in code complexity and static-analysis warnings that slowed later development (He et al., 2026). Since compliance determines whether generalist coding agents can be trusted to extend real software without eroding its compatibility, maintainability, and robustness, it is essential to measure how compliant the changes made by current coding agents are.



Figure 2: **Overview of GAMEMOD.** Repair tasks provide a bug report and an existing implementation; construction tasks specify a new extension and its game interfaces. An agent inspects and modifies code, which is then built and tested in the designated game or toolchain. The scenes illustrate repair and construction, not recorded agent outputs. In the controlled study, the evaluator performs compilation and execution without returning feedback to the agent (Section 4.1).

To enable this measurement, we turn to game modding, a common real-world practice of extending existing software (Scacchi, 2011). A mod adds new content or mechanics to a released game and is expected to run correctly with both the game and the other mods installed alongside it, mirroring the compliance requirement of software extension. Game modding is well suited to evaluating compliance for two reasons. First, the real game runtime serves as a comprehensive and strict verifier of compliance. An implemented mod can be loaded into the game and checked for whether it works correctly with the game and introduces no new bugs. Because the verdict comes from running the game itself, it is faithful and does not rely on LLM-generated regression tests. Second, the compliance complexity is adjustable. In addition to the game, an implemented mod can be required to stay compatible with other independently developed mods. Since mods for the same game are written by different authors without coordination and often modify the same files (Lee et al., 2020), loading more of them into the same runtime makes compliance progressively harder, while the verification remains equally strict.

Building on these properties, we introduce GAMEMOD, a benchmark for evaluating the compliance of coding agents, consisting of 150 game-mod construction and repair tasks across 12 games (Figure 2). These games, including Minecraft and Stardew Valley, are popular and have active modding communities, which provide a rich source of high-quality mods for building our tasks. The benchmark is also multilingual, covering mods written in programming languages such as Lua and C#. In the 100 *repair* tasks, an agent receives an existing mod into which we have inserted bugs, together with the corresponding bug reports, and is asked to fix them without breaking the mod's compatibility with the game and other mods. In the 50 *construction* tasks, the agent receives a feature request and the relevant game interfaces and is asked to implement a new mod that provides the requested feature under the same compatibility requirement. Figure 1 shows a representative task. For every task, a withheld verifier builds the implemented mod and runs it in the real game runtime together with other independently developed mods. The mod passes only if the requested fix or feature works and all previously working functionality remains intact, a strict criterion that directly tests compatibility. All tests are validated by running them in the game and by human review: a correct reference solution passes them, while the initial code given to the agent fails them, confirming that the tests reliably distinguish compliant mods from non-compliant ones.

With this verification in place, we evaluate six frontier models on GAMEMOD as coding agents, namely GPT-6 Astra (OpenAI, 2026), Claude Opus 5 (Anthropic, 2026), DeepSeek-V4.1-Flash (DeepSeek-AI, 2026), GPT-OSS 120B (OpenAI, 2025), Nemotron 3.5 Lightning 30B A3B (NVIDIA, 2026), and Qwen 3.8 27B (Qwen Team, 2026). The results show that current coding agents are not yet reliably compliant. Our analysis (Section 4.2) reveals three limitations shared across these models:

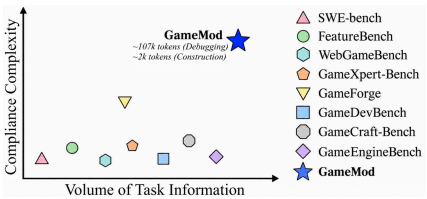
- **Rewriting a component can erase functionality that worked before.** Rather than building on existing components with targeted edits, agents often regenerate entire files and silently drop what the rest of the mod depends on, such as a field required by its save format. The resulting mod no longer works with the code it was meant to extend.
- **Repairing one bug can break mods that used to coexist.** When a bug report is ambiguous or conflicts with the existing code, agents tend to force a change that satisfies the report, for example by rejecting a dependency pattern on which two cooperating mods rely. Such conflicts only appear when mods share a runtime, which GAMEMOD reproduces by co-loading them.
- **An agent's own tests are a poor proxy for success in the game.** Without access to the game during implementation, agents often check their mods against mock environments built on their own assumptions. Mods that pass these checks can still fail once loaded into the game, for instance by accessing globals that the mod environment does not expose. These failures evade the agents' local checks and appear only when the mod actually runs, underscoring the value of verifying mods in the real game runtime.

Although GAMEMOD is built on games, the same challenge arises for plugins and extensions in other software ecosystems. We see GAMEMOD as a realistic and verifiable starting point for studying compliance, not a complete measure of it. We also maintain GAMEMOD as a living benchmark, adding new tasks over time to limit contamination while publishing fixed releases for reproducible evaluation.

2 RELATED WORK

Benchmarks for coding agents. Evaluation of code generation has moved from function-level synthesis in HumanEval (Chen et al., 2021) to repository-level issue resolution in SWE-bench (Jimenez et al., 2024), which checks both the requested change and the tests that passed before it. Subsequent benchmarks broaden this setting along several axes: SWE-PolyBench spans multiple programming languages and task types (Rashid et al., 2025), SWE-bench Multimodal adds visual context to issues (Yang et al., 2025a), SWE-Lancer evaluates professional freelance tasks through end-to-end application tests (Miserendino et al., 2025), SWE-Bench Pro targets complex, long-horizon changes (Deng et al., 2025), and FeatureBench evaluates feature implementation with and without the surrounding codebase (Zhou et al., 2026). Our verifier follows SWE-bench in checking both the requested fix or feature and previously working functionality. However, these benchmarks evaluate a change within a single repository, typically against that repository's own tests. GAMEMOD instead evaluates changes that extend a system maintained by others, verifies them in that system's real runtime, and additionally requires compatibility with other independently developed mods.

Game development benchmarks. A growing line of work evaluates coding agents on game development. GameDevBench builds tasks from Godot tutorials to test the implementation of gameplay, graphics, and UI features (Chi et al., 2026), and GameEngineBench asks agents to develop gameplay systems, networking, and persistence in existing Unreal Engine projects (La et al., 2026). GameCraft-Bench (Luo et al., 2026) and WebGameBench (Zhang et al., 2026) evaluate complete games by interacting with them at runtime, while GameXpert-Bench covers the generation, repair, and iterative improvement of browser games with regression checks (Chen et al., 2026). These benchmarks mainly target the development of a game project itself. Closest to our work, GameForgeBench evaluates repairs and feature completion in Godot projects, standalone C++ subsystems, browser games, and Minecraft Paper plugins, using executable checks that also cover existing functionality, and verifies some of its plugin tasks on a live Paper server (Zi, 2026). GAMEMOD differs from it in three respects (Table 1). First, GAMEMOD is built entirely around modding released games and spans 12 of them, whereas the released-game tasks in the cited version of GameForgeBench come from



Benchmark	Avg. token	Compliance verifier	Compliance complexity	Capability evaluated
SWE-bench	6.8M	✗	Low	Resolve repository issues
FeatureBench	8.2M	✗	Low	Implement repository features
GameDevBench	34K	✗	Low	Godot gameplay, graphics, UI
GameEngineBench	460K	✗	Low	Unreal C++ gameplay systems
GameCraft-Bench	8.4K	✗	Low	Build complete Godot games
GameXpert-Bench	7.2K	✗	Low	Generate, repair, improve games
GameForge	1.1K	✓	Medium	Edit and repair game projects
GAMEMOD	**72K**	✓	High	**Repair and extend game mods**

Table 1: **Benchmark landscape and comparison with related benchmarks.** *Left:* positions are qualitative, and the volume of task information is measured by the median per-task context. *Right: Avg. token* is the mean number of tokens in the raw input supplied to the agent per task. *Compliance verifier*: the implemented code is verified in the shipped game or its dedicated server. *Compliance complexity*: *Low*, no strict compliance requirement; *Medium*, the code runs inside the extension framework of a released game but is tested on its own; *High*, it is tested together with other independently developed mods in the same runtime.

Minecraft alone. Second, every implemented mod in GAMEMOD is verified in the real game runtime together with other independently developed mods, which raises compliance complexity beyond running an extension on its own; we find no such co-loading in the public GameForgeBench tasks. Third, agents in GAMEMOD work with considerably larger task contexts, since each task includes a full mod and the game interfaces it builds on. The left panel of Table 1 places these benchmarks by task information and compliance complexity.

Code reuse and software evolution. Beyond functional correctness, recent work examines how agent-written code fits into and evolves with a codebase. R2E studies generation with repository context and reports cases in which models reimplement helpers that the repository already provides (Jain et al., 2024), and SlopCodeBench shows that code quality degrades as agents repeatedly extend their own implementations under changing specifications (Orlanski et al., 2026). These studies focus on the agent's own code, either a single generation or a sequence of edits to it. GAMEMOD complements them by evaluating compliance with software that others develop and maintain: an implemented mod is judged by whether it works with the real game and with other mods, rather than by the quality of its code in isolation. Appendix G discusses further related work, and Appendix E documents the sources of our comparison.

3 GAMEMOD: EVALUATING COMPLIANCE THROUGH GAME MODDING

This section describes the design of GAMEMOD. We first define the task setting and summarize the task collection, then present the compliance requirements that the tests target and how the verifier evaluates an implemented mod, and finally describe how the tasks are annotated and calibrated across games.

3.1 TASK FORMULATION AND COLLECTION

Task formulation. Each task in GAMEMOD asks an agent to change a mod for an existing game. The agent receives a task description d, the initial code c_0, and the game interfaces and tools it may use, and it produces modified code c. A verifier, withheld from the agent, then evaluates c in an environment E consisting of the real game runtime and a set of other independently developed mods. GAMEMOD includes two task types. In *repair* tasks, c_0 is an existing mod containing inserted bugs and d consists of bug reports for them. In *construction* tasks, c_0 is an empty mod project or a provided skeleton and d is a feature request. Figure 4 shows one example of each type.

Task statistics. GAMEMOD contains 150 tasks, including 100 repair and 50 construction tasks, with 10 to 18 tasks per game (Figure 3). The code is written in Lua (including the GLua dialect of Garry's Mod), Java, C#, SourcePawn, and Squirrel, and it runs in desktop games or dedicated game servers. Several tasks share a codebase, such as the Automate mod for Stardew Valley and the DarkRP game mode for Garry's Mod; we record these groups so that repeated use of one codebase remains visible in the analysis (Appendix A).



Figure 3: **Task composition and game coverage.** (a) The 150 tasks comprise 100 repairs and 50 constructions. (b) Covers identify 12 games; equal-sized tiles do not encode task counts.

3.2 Compliance Requirements

Following the definition in Section 1, the tests of each task target three facets of compliance, which we illustrate below with concrete tasks.

Compatibility with the game. A mod should build on the interfaces and operations that the game already provides rather than reimplementing them. In a Minecraft plugin task, reloading the configuration fails to update the server's command registry; a correct fix makes new and renamed commands available and removes obsolete ones. Blueprint, a Garry's Mod construction task, saves and restores groups of connected objects, and the restored objects are expected to work with the game's ordinary Undo action. Loadout, another construction task, adds equipment profiles that persist when the player dies and respawns and when the server restarts.

Compatibility with other mods. A mod should keep serving the mods that depend on it and coexist with those loaded alongside it. Automate is a Stardew Valley mod that lets machines exchange items with nearby storage chests, and another mod reads Automate's API to show players the status of each machine. In one repair task, the status report is missing when no machine at a location is connected to storage. A compliant fix restores the report through the existing API, since the consuming mod has no way to reach a newly added function. Beyond such direct dependencies, every implemented mod is evaluated together with other independently developed mods (Section 3.3).

Preservation of existing functionality. A change should introduce no new bugs into functionality that already worked. In the Minecraft task, unchanged commands keep working after the reload fix; in the Automate task, reports for machines that are already connected to storage remain correct. Appendix C gives the full test setups.

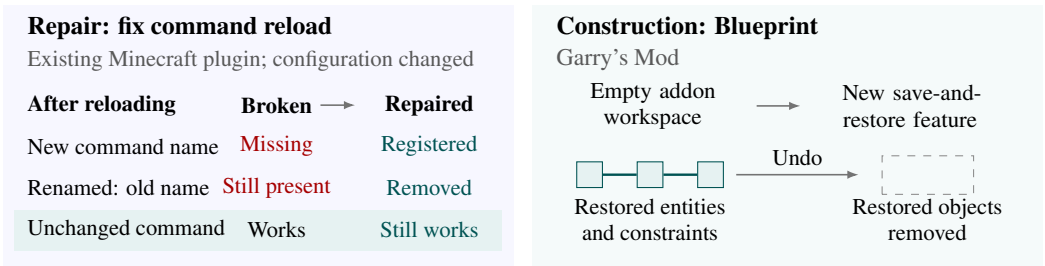


Figure 4: **Compliance requirements in repair and construction.** Left: a repair restores command reloading, so that added and renamed commands are registered, obsolete ones are removed, and unchanged ones keep working. Right: a construction adds save-and-restore support whose restored objects work with the game's ordinary Undo. Both panels illustrate task requirements rather than agent outputs.

3.3 VERIFICATION AND VALIDATION

Verification. The verifier builds c, loads it into the game together with the co-loaded mods in E, and runs a set of tests. The co-loaded mods of each task are independently developed mods for the same game, and their versions are pinned with the task. Each test starts from a specified game state, performs a sequence of actions, and checks the outcome; we refer to the state and actions as the test's *fixture*. *Completion tests* check that the requested fix or feature works, and *preservation tests* check functionality that worked before the change; the latter are included whenever the unmodified game and initial code demonstrate that functionality.

Metrics. An implemented mod passes a task, and the task counts as *resolved*, only if it passes all completion and preservation tests in E and the run terminates cleanly. A mod that fails to build or load produces no test outcomes and leaves the task unresolved. We report the resolution rate, the percentage of resolved tasks, separately for repair and construction, and record completion and preservation outcomes separately to distinguish an incomplete fix from one that breaks existing functionality (Section 4.2). Each task pins the versions of the game, the mods, and the verifier, so that results remain reproducible.

Validation. Before release, every task is checked in the real game runtime: the reference solution passes all tests, and on the initial code the completion tests fail while the preservation tests pass. A human reviewer additionally runs each task in the game, including actual gameplay, to confirm that its description matches what happens in the game and that the tests check it. Tasks that fail any check are revised or discarded. New tasks go through the same process, which allows GAMEMOD to grow as a living benchmark while each fixed release remains reproducible.

3.4 DATA ANNOTATION

Data annotation. We construct our tasks from publicly available, open-source mods. For each selected mod, we record its upstream source, revision, license, and attribution, and retain the original licensing information in the released benchmark. Repair tasks are derived from existing implementations, while construction tasks are based on functionality exposed or documented by the corresponding modding ecosystem and are paired with reference implementations written by us. GAMEMOD is intended solely for academic research and non-commercial evaluation. We do not redistribute proprietary game binaries or other restricted game assets; benchmark artifacts are released only to the extent permitted by their upstream open-source licenses. Appendix B gives further details.

Difficulty calibration across games. Because the 12 games differ substantially in programming language, API design, and modding conventions, we calibrate task difficulty separately for repair and construction tasks to make cross-game comparisons meaningful. For repair tasks, annotators control the scope of the required fix by matching the complexity of the reference patch across games, considering factors such as the amount of code that must be inspected or modified, the number of affected files or functions, and the number of behaviors involved in the bug report. For construction tasks, we similarly match the complexity of the reference implementation, including its implementation scope, the number of components or interfaces involved, and the amount of new code required. Tasks that are substantially easier or harder than others of the same type are revised or excluded. This calibration keeps the coding burden within a comparable range across games, so that differences in benchmark performance more directly reflect an agent's ability to satisfy the compliance requirements rather than incidental differences in task size or implementation complexity.

4 EXPERIMENTS

We evaluate how well current coding agents satisfy the compliance requirements of GAMEMOD. This section describes the evaluated models and agent setup, reports resolution rates across the 12 games, and analyzes the failures behind unresolved tasks.

Table 2: **Main results across the 12 game environments in GAMEMOD.** Each cell reports task success rate (%) for the corresponding game. **Total** averages the per-game rates, weighting each game equally.

Model / Agent	Success Rate (%) ↑												
	Minecraft	Stardew Valley	Garry's Mod	Don't Starve Together	Terraria	RimWorld	Mindustry	Palworld	The Binding of Isaac	Left 4 Dead 2	Balatro	Slay the Spire	Total
GPT-6 Astra	66.7	73.3	53.3	60.0	50.0	60.0	66.7	50.0	33.3	40.0	30.0	50.0	52.8
Claude Opus 5.0	33.3	26.7	0.0	26.7	0.0	40.0	26.7	0.0	16.7	0.0	0.0	30.0	16.7
DeepSeek-V4.1-Flash	50.0	60.0	33.3	46.7	50.0	40.0	66.7	40.0	33.3	20.0	10.0	40.0	40.8
GPT-OSS 120B	0.0	0.0	0.0	6.7	0.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0	1.4
Nemotron 3.5 30B A3B	8.3	0.0	0.0	6.7	10.0	10.0	13.3	0.0	0.0	0.0	0.0	0.0	4.0
Qwen 3.8 27B	16.7	0.0	13.3	6.7	0.0	10.0	6.7	0.0	0.0	10.0	0.0	0.0	5.3

4.1 EXPERIMENTAL SETUP

Models. We evaluate six models as coding agents: GPT-6 Astra (OpenAI, 2026), Claude Opus 5 (Anthropic, 2026), DeepSeek-V4.1-Flash (DeepSeek-AI, 2026), GPT-OSS 120B (OpenAI, 2025), Nemotron 3.5 Lightning 30B A3B (NVIDIA, 2026), and Qwen 3.8 27B (Qwen Team, 2026). They cover both proprietary and open-weight models across a wide range of sizes.

Agent setup. All models run in the same agent scaffold, mini-swe-agent, with text and shell tools and limits of 100 steps and 90 minutes per attempt. Network access is restricted to the model endpoint. The agent can inspect and edit the task's code, but it has no access to the game runtime, the co-loaded mods, or the verifier during implementation. After the agent finishes, or when it reaches a limit, the verifier evaluates its final code as described in Section 3.3. The agent therefore receives no feedback from the game, and compliance must be achieved from the code and the task description alone.

Metrics. We report the resolution rate defined in Section 3.3: a task is resolved only if the implemented mod builds, loads, and passes all completion and preservation tests together with the co-loaded mods. For each game, we compute the resolution rate over its tasks; the overall score averages the per-game rates, so that each game contributes equally regardless of its number of tasks.

4.2 MAIN RESULTS

Table 2 reports the resolution rate of each model in each game.

Current agents are far from reliably compliant. GPT-6 Astra achieves the highest overall resolution rate at 52.5%, followed by DeepSeek-V4.1-Flash at 40.8% and Claude Opus 5 at 17.3%. Even the strongest model fails on nearly half of the tasks: its per-game rates range from 30.0% on Balatro to 70.0% on Stardew Valley. Because a task is resolved only when the mod works with the game, the co-loaded mods, and the functionality that existed before the change, these rates show that producing a compliant change remains difficult even for frontier models.

Performance differs sharply across models. GPT-6 Astra and DeepSeek-V4.1-Flash are the only models that resolve tasks in all 12 games. GPT-6 Astra achieves the highest rate in every game, tied with DeepSeek-V4.1-Flash in Terraria, Mindustry, and The Binding of Isaac. DeepSeek-V4.1-Flash in turn outperforms Claude Opus 5 in eleven games, the exception being RimWorld (40.0% vs. 44.4%). Claude Opus 5 resolves no task in five games: Garry's Mod, Terraria, Palworld, Left 4 Dead 2, and Balatro. The three smaller open-weight models stay below 6% overall. GPT-OSS 120B resolves tasks in only two games, Nemotron 3.5 Lightning 30B A3B in five, and Qwen 3.8 27B in six, with no per-game rate above 16.7%. For these models, compliance failures leave most games without a single resolved task.

Difficulty varies across games. Don't Starve Together and RimWorld are the only games in which every evaluated model resolves at least one task. In contrast, only GPT-6 Astra and DeepSeek-V4.1-Flash resolve any task in Palworld and Balatro, and only these two models and Claude Opus 5 do so in Stardew Valley, The Binding of Isaac, and Slay the Spire. Balatro is the hardest game overall, with no model exceeding 30.0%. Since task difficulty is calibrated across games (Section 3.4), these differences suggest that the modding ecosystems themselves, including their languages, interfaces,

and conventions, pose unequal challenges to current agents. A model's performance in one game is therefore a poor predictor of its performance in another, and evaluating across many games is necessary to characterize an agent's compliance.

4.3 Error analysis

To understand why tasks remain unresolved, we analyze one run of Qwen 3.8 27B on 87 tasks across eight games, with one attempt per task. We classify all 79 unresolved attempts using the structured evaluation records, and inspect the logs, submitted code, and available trajectories of representative cases. Part of this run was ended by an external deadline before the agent's time limit, and most of these attempts left the initial code unchanged. We therefore report such attempts separately and base our conclusions on attempts whose submitted code differs from the initial code.

Overview. The agent resolves 8 of the 87 tasks, all of them repairs (8 of 62); it resolves none of the 25 construction tasks. Of the 35 attempts that modify the code, 8 are resolved, 18 fail, and 9 remain unattributed (Table 3). Among the 18 modified attempts with complete completion and preservation observations, 8 pass both and 10 keep existing functionality intact but leave the request incomplete. The failures below come from these modified attempts and illustrate how code written without the game fails once it runs there.

The agent's own checks are a poor proxy for success in the game. Several submissions pass the agent's local checks but fail in the game. In a Don't Starve Together construction task, the submitted logic passes all six hidden logic tests, but the mod calls the global `net_string`, which is not defined in the mod environment, and the game disables the mod at load time. In a repair of The Binding of Isaac, the agent's local harness and summary claim that repeated item use yields positive, diminishing gains. In the game, the second use instead lowers damage from about 10.58 to 9.78, because the fix scales the current bonus without preserving the gains already accumulated. In a Left 4 Dead 2 task, the agent concludes that all requirements are met after checking bracket balance and searching for interface names. The submission writes each record back with a single-element array call where the whole record must be written, and the game server rejects it with `Invalid block 82 (blocksize: 82)`, so all 42 behavioral tests fail. A single misused container operation makes the entire feature unusable in the game.

Code does not conform to the interfaces and contracts of the existing ecosystem. Six modified submissions fail source, metadata, or build checks, and the logs of three unattributed attempts show concrete build errors. In Mindustry, the submitted Java code compares and assigns incompatible generic types from the existing codebase (`Seq<PatchAsset>` and `Seq<DataAsset>`). In Slay the Spire, it calls `getReturnType()` on a `CtBehavior` object from the mod loader's bytecode library, which provides no such method. In Left 4 Dead 2, the SourcePawn submission references unresolved symbols and types. The agent writes plausible code against interfaces it assumes rather than those the existing code actually provides. Construction tasks show the same problem with packaging conventions: three Garry's Mod addons implement most of the requested behavior, passing 16 of 18, 11 of 14, and 11 of 13 core tests, but are rejected because their `addon.json` lacks a valid title or declares the wrong addon type.

Changes are not integrated with the full lifecycle of game state. Among the 12 modified attempts that build but fail behavioral tests, six fail only one or two tests, indicating that the requested change is largely implemented but not fully connected to the game. In an Isaac repair, the agent registers a new item in the table of charged items but does not implement how its charge is assigned and restored. The game reports a fresh charge of 6 and a charge of 0 restored after an item swap, where 3 was stored before the swap, and the same two failures recur in a second Isaac task. In a Don't Starve Together repair, the submission adds the call that drops a gift's contents on the ground and passes a syntax check, yet opening the gift from the inventory still does not release the creature. Such failures appear only when the game executes the full sequence of events.

Without a runtime, the agent can stall instead of editing. Two attempts end because the agent repeats the same action five times in a row. In a Mindustry construction task, the agent notes that no Java toolchain is available but keeps searching for a JDK instead of writing the implementation. In a

Table 3: **Outcomes of the 79 unresolved attempts of Qwen 3.8 27B.** Categories are mutually exclusive and describe observed outcomes rather than root causes.

Category	Attempts	Share (%)
Unchanged code, terminated before submission	49	62.0
Unchanged code, stopped for repeating the same action	2	2.5
Modified code that fails to build or pass submission checks	6	7.6
Modified code that builds but fails behavioral tests	12	15.2
Evaluation outcome not attributed	10	12.7
Total	79	100.0

Garry's Mod repair, it repeatedly reads the same file while investigating one bug. Both attempts end with the initial code unchanged.

5 LIMITATIONS

Scope of the setting. GAMEMOD studies compliance through game modding. Mods share the core constraints of software extension, but other ecosystems, such as IDE plugins or shared libraries, may involve factors that games do not capture, such as versioning policies and code review. Our findings therefore characterize compliance in this setting and may not transfer directly to every other one. Adding tasks from other plugin ecosystems would show how far these findings generalize.

Evaluation setting. Agents implement mods without access to the game runtime, using a single agent scaffold with fixed step and time limits. This setting isolates how well agents handle compliance from the code alone, but agents with runtime access, other scaffolds, or larger budgets may perform differently. Measuring how much agents gain from executing their mods in the game during implementation is left to future work.

Reproducibility and data. We will release all tasks, verifiers, and the evaluation harness as open source. The mods in our tasks are also open source, but the games themselves cannot be redistributed: running the verifier requires licensed copies, and some games run only on specific platforms. Since these mods are public, models may have seen their original versions during training. Inserted bugs and new feature requests reduce the benefit of memorization, and we continue to add tasks over time to further limit contamination.

6 CONCLUSION

We presented GAMEMOD, a benchmark that measures whether coding agents can extend existing software without disrupting it. Game modding makes this measurement possible: the shipped game offers an authoritative check of each implemented mod, and loading other mods alongside it raises the difficulty without weakening that check. Accordingly, GAMEMOD asks agents to repair or construct mods across 150 tasks in 12 popular games. Across six frontier models, compliance emerges as a clear weakness: agents discard working functionality when regenerating code, introduce conflicts between mods while fixing bugs, and rely on local tests that the real runtime later contradicts. Because these failures often occur even when the core logic is correct, the capability that most needs improvement is integrating code with the software around it rather than generating the code itself. Promising directions include giving agents feedback from the running game and extending this form of evaluation to extension ecosystems beyond games. We hope GAMEMOD helps the community develop coding agents whose changes remain compliant with the software they extend.

AI USE STATEMENT

AI tools assist manuscript preparation through language editing and polishing, literature research, figure creation and refinement, final proofreading, and typesetting. The authors take responsibility for the accuracy of the content, citations, and reported results.

ETHICS STATEMENT

GAMEMOD is built on open-source community mods, and all upstream code keeps its original license and attribution. The artifact excludes proprietary game binaries, runtime caches, credentials, and private operational state; running the benchmark requires appropriately licensed game installations. Because implemented mods run inside the game process, we recommend evaluating them in isolated environments. Game artwork in Figure 3 belongs to its rights holders (Appendix F).

REPRODUCIBILITY STATEMENT

We will release all tasks, verifiers, reference solutions, and the evaluation harness as open source. Appendix A lists the games and their runtimes, Appendix B describes how they are constructed and validated, and Appendix D specifies how outcomes are determined. Each task pins the versions of the game, the mods, and the verifier. Running the verifier requires licensed copies of the games, which we cannot redistribute.

REFERENCES

Anthropic. System card: Claude Opus 5, July 2026. URL https://www-cdn.anthropic.com/c5fbac3f0b1280a933ebd26d3cb8bb9f5bdeaf48/Claude%20Opus%205%20System%20Card.pdf. Accessed September 26, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. arXiv preprint arXiv:2505.20411, 2025. URL https://arxiv.org/abs/2505.20411v2. Version 2.

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. URL https://arxiv.org/abs/2608.21833v1. Version 1.

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. URL https://arxiv.org/abs/2107.03374v2. Version 2.

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 *Forty-Third International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/64919.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression, 2026. URL https://arxiv.org/abs/2609.19969.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? arXiv preprint arXiv:2509.16941, 2025. URL https://arxiv.org/abs/2509.16941v2. Version 2.

K M Ferdous, Dipayan Banik, Kowshik Chowdhury, and Shazibul Islam Shamim. Safer builders, risky maintainers: A comparative study of breaking changes in human vs agentic PRs. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, 2026. URL https://arxiv.org/abs/2603.27524.

Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality: How Cursor AI increases short-term velocity and long-term complexity in open-source projects. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, 2026. URL https://arxiv.org/abs/2511.04427.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2E: Turning any github repository into a programming agent environment. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 21196–21224. PMLR, 2024. URL https://proceedings.mlr.press/v235/jain24c.html.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. arXiv preprint arXiv:2504.07164, 2025. URL https://arxiv.org/abs/2504.07164v1. Version 1.

Shufan Jiang, Chios Chen, and Zhiyang Chen. GBQA: A Game Benchmark for Evaluating LLMs as Quality Assurance Engineers. arXiv preprint arXiv:2604.02648, 2026. URL https://arxiv.org/abs/2604.02648v1. Version 1.

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://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

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/2607.03525v2. Version 2.

Daniel Lee, Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Building the perfect game – an empirical study of game modifications. *Empirical Software Engineering*, 25(4):2485–2518, 2020. doi: 10.1007/s10664-019-09783-w.

Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68 (9):1060–1076, 1980. doi: 10.1109/PROC.1980.11805.

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*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

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/2606.17861v1. Version 1.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 44412–44450. PMLR, 2025. URL https://proceedings.mlr.press/v267/miserendino25a.html.

Emerson Murphy-Hill, Jenna Butler, and Alexandra Savelieva. Adoption and impact of command-line AI coding agents: A study of Microsoft's early 2026 rollout of Claude Code and GitHub Copilot CLI. *arXiv preprint arXiv:2607.01418*, 2026. URL https://arxiv.org/abs/2607.01418v1. Version 1, July 1, 2026.

NVIDIA. NVIDIA-Nemotron-3.5-Lightning-30B-A3B. Hugging Face model card, August 2026. URL https://huggingface.co/nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16. Accessed September 26, 2026.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

OpenAI. GPT-6 Astra system card, September 2026. URL https://deploymentsafety.openai.com/gpt-6-astra. Accessed September 26, 2026.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026. URL https://arxiv.org/abs/2603.24755v2. Version 2, May 7, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training Software Engineering Agents and Verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Zhiyuan Peng, Wei Tao, Xin Yin, Chenhao Ying, Yuan Luo, and Yiwen Guo. PlayCoder: Making LLM-Generated GUI Code Playable. arXiv preprint arXiv:2604.19742, 2026. URL https://arxiv.org/abs/2604.19742v1. Version 1.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8. Accessed September 26, 2026.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. arXiv preprint arXiv:2504.08703, 2025. URL https://arxiv.org/abs/2504.08703v3. Version 3.

Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. Agentic much? adoption of coding agents on GitHub. *arXiv preprint arXiv:2601.18341*, 2026. URL https://arxiv.org/abs/2601.18341v2. Version 2, April 8, 2026.

Walt Scacchi. Modding as an open source approach to extending computer game systems. In *Open Source Systems: Grounding Research (OSS 2011)*, volume 365 of *IFIP Advances in Information and Communication Technology*, pp. 62–74. Springer, 2011. doi: 10.1007/978-3-642-24418-6_5.

Sonar. State of code developer survey report 2026. Technical report, SonarSource, 2026. URL https://www.sonarsource.com/state-of-code-developer-survey-report.pdf. Survey of 1,149 developers fielded October 2025; published January 2026. Accessed September 26, 2026.

Weihao Tan, Changjiu Jiang, Yu Duan, Mingcong Lei, Jiageng Li, Yitian Hong, Xinrun Wang, and Bo An. StarDojo: Benchmarking Open-Ended Behaviors of Agentic Multimodal LLMs in Production-Living Simulations with Stardew Valley. arXiv preprint arXiv:2507.07445, 2025. URL https://arxiv.org/abs/2507.07445v4. Version 4.

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. arXiv preprint arXiv:2305.16291, 2023. URL https://arxiv.org/abs/2305.16291v2. Version 2.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. SWE-Mirror: Scaling Issue-Resolving Datasets by Mirroring Issues Across Repositories. arXiv preprint arXiv:2509.08724, 2025a. URL https://arxiv.org/abs/2509.08724v1. Version 1.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025b. URL https://arxiv.org/abs/2407.16741.

Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pp. 226–236. Association for Computing Machinery, 2017. doi: 10.1145/3092703.3092718. URL https://doi.org/10.1145/3092703.3092718.

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*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains? In *International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/07d6332ae36730707fddddba736d7b6c-Abstract-Conference.html.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling Data for Software Engineering Agents. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc., 2025b. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.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/2605.17637v2. Version 2.

Pujun Zheng, Zixin Shang, Shufan Jiang, Wenhui Tian, Dongsheng Zhu, Zerun Ma, Dingbo Yuan, and Qi Zhang. SWE-Bench Pro Verified: A Reliable Benchmark for Software Engineering Agents. arXiv preprint arXiv:2609.08149, 2026. URL https://arxiv.org/abs/2609.08149v1. Version 1.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking Agentic Coding for Complex Feature Development. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=41xrZ3uGuI.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training Versatile Coding Agents in Synthetic Environments. arXiv preprint arXiv:2512.12216, 2025. URL https://arxiv.org/abs/2512.12216v2. Version 2.

Chenyi Zi. GameForge. Software repository, 2026. URL https://github.com/GamePhanesStudio/GamePhanes/tree/4449f2930b26608789e404089c6d7672d47ad592. Software branded GameForgeBench in the README. CITATION.cff version 0.1.0; commit 4449f2930b26608789e404089c6d7672d47ad592. Accessed 2026-09-13.

A Task Inventory

Table 4 lists the 12 games in GameMod, together with the language of their mods and the runtime in which the verifier runs them; Figure 3 gives the number of tasks per game. Tasks derived from the same codebase form a source group, such as Automate for Stardew Valley, DarkRP for Garry's Mod, and Exodus for The Binding of Isaac, and we report results with these groups in mind. Task descriptions keep their original natural language; for example, the Loadout description is written in Chinese.

Table 4: **Games in GameMod.** Runtime is the environment in which the verifier runs the implemented mod together with the co-loaded mods.

Game	Language	Runtime
Minecraft	Java	Paper server
Stardew Valley	C#	Desktop game (SMAPI)
Garry's Mod	GLua	Dedicated server
Don't Starve Together	Lua	Dedicated server
Terraria	C#	Dedicated server (tModLoader)
RimWorld	C#	Desktop game
Mindustry	Java	Headless server
Palworld	Lua (UE4SS)	Dedicated server
The Binding of Isaac	Lua	Desktop game
Left 4 Dead 2	SourcePawn, Squirrel	Dedicated server
Balatro	Lua	Desktop game
Slay the Spire	Java	Desktop game (ModTheSpire)

B Task Construction Details

This appendix details how the tasks in GameMod are constructed and validated, complementing Sections 3.3 and 3.4.

Mod sources. Tasks start from open-source mods published by the modding communities of the 12 games, at pinned revisions. Upstream code keeps its original license and attribution.

Repair tasks. A repair task starts from a working version of a mod. We insert bugs in one of two ways. When the mod's history contains a suitable fix, we use the code immediately before that fix as the initial code and the fixed version as the reference solution. Otherwise, we introduce a new defect into the working code, which then serves as the reference solution. Each inserted bug is described in a separate bug report, and a single task may bundle several bugs.

Construction tasks. A construction task specifies new functionality in terms of the interfaces and operations that the game already provides. Some feature requests adapt features documented on the Steam Workshop into self-contained tasks. The initial code is either an empty mod project or a skeleton that sets up the mod's entry points. We implement each requested feature ourselves, and this implementation serves as the reference solution.

Co-loaded mods. For every task, we select other independently developed mods for the same game and load them alongside the implemented mod during verification. Their versions are pinned with the task.

Test authoring. We write all tests by hand, following the three facets of compliance in Section 3.2. Each reported bug or requested feature is covered by at least one completion test. Preservation tests cover functionality that already works in the unmodified game and initial code, including interactions with the co-loaded mods; a construction task that starts from an empty project has no such baseline and contains completion tests only. Every test is a fixture that sets up a game state, performs a sequence of actions, and checks the outcome in the real game runtime.

Validation. A task enters the benchmark only after passing three checks. First, the reference solution passes all tests with the co-loaded mods present. Second, on the initial code, every completion test fails and every preservation test passes, confirming that the tests detect the inserted bugs or the missing feature and that the preserved functionality works before any change. Third, a human reviewer runs the task in the game, including actual gameplay, and confirms that the task description matches the effects observable in the game and that the tests check those effects. Tasks that fail any check are revised and validated again, or discarded.

Versioning and maintenance. Each task pins the versions of the game, the target mod, the co-loaded mods, and the verifier. New tasks go through the same construction and validation process and are published in fixed releases, so results reported on one release remain comparable over time.

C Task Examples

This appendix describes the four tasks used as examples in Section 3.

C.1 Automate: returning missing reports through the existing API

Automate is a Stardew Valley mod that lets machines, such as furnaces, exchange items with nearby storage chests. Another mod, the Data Layers overlay, reads machine states through Automate's API to show players the status of each machine. In the repair task, Automate returns no reports for a location in which no machine is connected to storage. The requested fix adds these reports while keeping the reports for machines that are already connected. The tests query the API immediately after the game loads and expect a nonempty result, then place a chest next to a furnace and expect a report for the connected furnace together with the other reports. A compliant fix changes what the existing API returns; adding a separate reporting function would leave the data that the overlay receives unchanged.

C.2 Chatty: keeping the command registry consistent after reload

Chatty is a Minecraft plugin for the Paper server that defines chat commands in its configuration. In the repair task, reloading an edited configuration leaves the server's command registry out of date. After a correct fix, the plugin registers newly configured commands, removes the names of deleted or renamed commands, and keeps unchanged commands and their aliases working. The tests edit the configuration, reload the plugin on the Paper server, and check which commands the server registers for the plugin and whether the console can dispatch them.

C.3 Blueprint: restored objects that work with Undo

Blueprint is a Garry's Mod construction task that starts from an empty addon workspace. The feature request asks for saving and restoring a contraption, a group of game objects together with the physical constraints that connect them, and requires the restored objects to work with the game's ordinary Undo action. The test restores a contraption of three entities and two constraints, invokes Undo, and checks that all restored objects are removed. Because the initial workspace is empty, the task contains completion tests only.

C.4 Loadout: equipment profiles across death and restart

Loadout is a Garry's Mod construction task whose description is written in Chinese. The requested addon consists of a manifest, a profile store, and an adapter that connects the store to the game server. The store keeps named equipment profiles for each player, supports a preferred profile, assigns revision numbers that only increase, and returns the same result when a request is repeated. Applying a profile removes the player's weapons, clears their ammunition, grants the profile's equipment and ammunition, and selects a weapon. Reapplying a profile does not accumulate pickups, and if a weapon cannot be granted, the adapter stops before granting ammunition.

The saved state contains logical player identities and profile data rather than game objects, so that a profile can be restored for a new player object. The main fixture runs in two dedicated-server

processes. In the first, a bot receives a profile with 60 rounds of primary and 10 rounds of secondary ammunition, which combine into a 70-round plan; the bot then picks up 50 additional rounds, is killed, and respawns, and the test checks that its ammunition returns to 70 rounds. After the profile is saved, a second server process binds a new bot to the same logical identity and checks that the 70-round plan is restored. Further fixtures cover disconnecting and reconnecting a player.

D VERIFICATION DETAILS

This appendix specifies how the verifier determines test and task outcomes and how it handles failures unrelated to the implemented mod.

Test outcomes. Each test ends in one of three states. It *passes* when its checks succeed, *fails* when a check is violated, and is *unobserved* when execution stops before the test runs, for example because the mod crashes. We keep unobserved tests separate from failed ones, so that a single crash does not appear as many independent failures.

Task outcomes. A task is resolved only if every completion and preservation test passes and the run terminates cleanly, including the required cleanup. A failed test, a crash caused by the implemented mod, or a failure to build or load leaves the task unresolved. We additionally record the completion and preservation outcomes separately: each is *passed* when all of its tests pass, *failed* when at least one of its tests fails, and *unknown* otherwise, as when a build failure prevents any test from running.

Failures unrelated to the implemented mod. An execution can also fail for reasons outside the implemented mod, such as a missing runtime or a faulty fixture. If no test has failed because of the implemented mod, such an execution is marked invalid and rerun once with the same code, runtime, initial state, seed, and limits. A valid failure is never replaced by a later passing run.

Versions. Each task pins the game build, the revision of the target mod, the co-loaded mods, and the verifier. Reported results refer to a specific release of GAMEMOD, and a change to any pinned component produces a new task version.

E BENCHMARK COMPARISON DETAILS

This appendix describes how we construct the comparison in Table 1.

Sources. We use the ICLR 2024 version of SWE-bench, version 1 of FeatureBench and GameXpert-Bench, version 2 of GameDevBench and GameEngineBench, the cited arXiv versions of GameCraft-Bench and WebGameBench, and the pinned software release of GameForgeBench, listed as Game-Forge in Table 1. Each entry reflects the documented scope of these sources; a feature absent from a description may still exist in the benchmark.

Task context. *Avg. token* is the mean number of tokens in the raw input supplied to the agent per task, which consists of the task statement and the files of the initial workspace. We measure it directly on the released tasks of each benchmark. For GAMEMOD, the left panel of Table 1 additionally reports the median separately for repair and construction tasks, because repair tasks include a full existing mod while construction tasks start from an empty project or a skeleton.

Compliance verifier. A benchmark has a compliance verifier if its tests run the implemented code in the shipped game or its dedicated server. At the cited commit, GameForgeBench contains 17 task packages: 10 Godot projects, 4 Minecraft Paper plugins, 2 C++ subsystems, and 1 browser game. Two of its Minecraft tasks build the plugin and verify it on a live Paper server through a protocol client and a verifier plugin, while the other packages run engine projects, standalone probes, or a headless browser.

Compliance complexity. Compliance complexity is *Low* when the task imposes no strict compliance requirement, *Medium* when the code runs inside the extension framework of a released game

but is tested on its own, and *High* when it is tested together with other independently developed mods in the same runtime. GameForgeBench's Paper plugins run inside Minecraft's plugin framework, but we find no task in its public release that loads independently developed plugins alongside the candidate.

Evaluation practices of the closest benchmarks. SWE-bench combines fail-to-pass tests of the requested change with pass-to-pass tests of previously working functionality (Jimenez et al., 2024); our completion and preservation tests follow the same distinction. FeatureBench evaluates feature implementation with and without the surrounding repository and separates partial from full success (Zhou et al., 2026). GameEngineBench asks agents to edit C++ in buildable Unreal Engine projects and evaluates the results through tests and an LLM judge (La et al., 2026). GameXpert-Bench counts a bug as fixed only when both its repair and associated preservation tests pass (Chen et al., 2026). GameForgeBench also checks existing functionality; for example, a Godot status-bar task requests a new visual style while requiring the bars' layout and value computation to stay unchanged (Zi, 2026).

F Figure Sources

The task illustrations in Figures 2 and 4 are schematic; their game scenes represent the two task types and are not captured agent outputs or execution results. The cover tiles in Figure 3 use official storefront artwork from Steam and Minecraft.net to identify the 12 games. They are promotional images rather than captures from benchmark executions, their equal sizes do not encode task counts, and the artwork belongs to its respective rights holders.

G Extended Related Work

This appendix relates GameMod to interactive testing, task construction, and evaluation validity.

G.1 Interactive testing and game-playing agents

PlayCoder couples repository-aware GUI code generation with automated playthroughs and iterative repair (Peng et al., 2026), and GBQA studies bug discovery through interaction with games that contain injected defects (Jiang et al., 2026). Both connect code quality to what happens in a running game, a view that GameMod shares. Game environments also serve a different purpose: Voyager builds reusable code skills for exploring Minecraft (Wang et al., 2023), and StarDojo evaluates open-ended agent behavior in Stardew Valley (Tan et al., 2025). In these works the agent plays the game, while in GameMod the agent extends it.

G.2 Constructing executable tasks

SWE-Gym provides executable repository environments for training agents (Pan et al., 2025), and SWE-rebench automates the collection and validation of issue-based tasks (Badertdinov et al., 2025). R2E-Gym generates tests and derives problem statements from code (Jain et al., 2025), SWE-smith combines programmatic and model-generated modifications with pull-request inversion (Yang et al., 2025b), SWE-Mirror transfers issues between repositories through separate test and defect generation (Wang et al., 2025a), and SWE-Playground generates projects and tasks from interface scaffolds with injected bugs (Zhu et al., 2025). These approaches scale task collection through automation. In GameMod, tests are written by hand and every task is validated in the real game runtime and through human review, trading scale for verification that reflects how mods are actually used.

G.3 Evaluation validity

A passing test suite can leave incorrect code undetected. Program-repair research uses additional tests to expose patches that overfit the original suite (Xin & Reiss, 2017), EvalPlus shows that stronger tests reveal incorrect code accepted by weaker ones (Liu et al., 2023), and SWE-Bench Pro Verified addresses leakage and inconsistencies between task descriptions and tests (Zheng et al., 2026). The

validation in GAMEMOD, which combines reference solutions, the initial code, and human review in the game, reduces but does not eliminate such risks.