GameEngineBench: Evaluating Coding Models on Real Runtime Environments

Brian La, Sejoon Chang, Ben Kim
Nitrode

Junyoung Bae
Nexon Intelligence Labs

Aamish Ahmad Beg[1], Sei Chang[2], Gonzalo Gonzalez-Pumariega[3]
[1]Dartmouth University [2]Columbia University [3]Cornell University

research@nitrode.com https://github.com/Nitrode-Research/GameEngineBench

Abstract

Game engines provide real-time simulation, rendering, physics, interaction, networking, and asset pipelines, making them valuable not only for games but also for 3D applications in healthcare, robotics, architecture, manufacturing, and related domains. Because game development is where these systems are most mature and publicly available, it offers a practical testbed for evaluating coding agents that must modify C++ code within stateful, interactive, real-time systems. We present GameEngineBench, a benchmark for evaluating coding agents on scoped C++ implementation tasks inside Unreal Engine 5 projects, built from nine real-world game repositories. The evaluation set consists of 110 tasks spanning gameplay mechanics, multiplayer behavior, AI and world orchestration, animation and movement, UI and session code, loading behavior, online-service integration, persistence, data serialization, XR behavior, and rendering-oriented plugins. These tasks require models to make native C++ changes that compile and satisfy behavioral tests within executable Unreal Engine projects. Across twelve evaluated configurations, the strongest model reaches 55.5% pass@1, while 31 tasks remain unsolved by every configuration. Our results demonstrate that frontier coding agents continue to struggle with deeply integrated C++ development for real-time interactive software, highlighting game-engine benchmarks as a valuable complement to existing software engineering evaluations.

1 Introduction

Coding agents are increasingly evaluated on realistic software engineering tasks, including long-horizon repository work and production-quality code review. However, existing benchmarks still largely focus on general-purpose software engineering. They do not directly measure whether generated C++ code can integrate with a running game engine, where correctness depends not only on language semantics but also on networking, object lifecycles, engine callbacks, assets, and interactions with existing gameplay systems. A patch may compile and appear plausible yet still fail because it updates only the local state, misses an object-lifecycle transition, or breaks an interaction with another engine system.

*The views and opinions expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of their respective employers or affiliated institutions.

This setting extends beyond entertainment software. Modern game engines provide real-time simulation, rendering, physics, interaction, networking, and asset pipelines that are widely used in simulation, digital twins, XR, robotics prototyping, virtual production, and interactive training. Engine-based game development is a practical source of benchmark tasks for this broader class of software because game repositories are mature, executable, stateful, and often publicly available.

Evaluating game-engine tasks is challenging due to models needing to satisfy both C++ compilation and engine runtime behavioral requirements. Even a small implementation change may need to execute on the authoritative server, synchronize state across clients, update UI only for the local player, and correctly manage when actors are destroyed or reused. These behavioral requirements are simple to describe at a high level but hard to verify completely: a test suite can only run selected scenarios, while a truly correct Unreal implementation must satisfy behavior across connected systems, such as gameplay logic, replicated state, UI updates, and cleanup behavior.

Existing coding benchmarks capture important aspects of agent capability, including isolated function synthesis, issue resolution in software repositories, long-horizon codebase tasks, and multimodal game construction. However, they do not directly evaluate scoped native C++ edits inside functioning game-engine projects, where correctness depends on engine execution, networking behavior, object lifecycles, subsystem initialization, and integration with existing gameplay code. GameEngineBench addresses this gap using Unreal Engine projects as an executable testbed.

GameEngineBench contains 110 tasks drawn from nine publicly available Unreal Engine repositories. Each task gives the model a buildable start state, scoped editable C++ files, and a behavior specification. After the model finishes, tests are injected and executed through Unreal's Play-in-Editor automation, and judge auditing determines whether the implementation satisfies the requested behavior rather than merely matching a reference solution. Across 12 evaluated configurations, the strongest model reaches 55.5% pass@1, while 31 tasks remain unsolved by every configuration. Many failed runs still recover substantial partial behavior, but the failures cluster around recurring runtime and integration patterns, including authority mistakes, state-synchronization errors, object-lifecycle bugs, initialization errors, and incomplete integration with surrounding game systems.

We make three contributions. First, we introduce a 110-task benchmark for evaluating C++ changes that require integration into functioning Unreal Engine projects. Models edit only native source files, although the projects themselves may include non-code assets required during execution.[1] Second, we define an evaluation protocol that combines scoped file edits, runtime tests, and judge auditing to score behavioral correctness rather than reference similarity. Third, we analyze the failure modes of current coding agents, showing that low pass@1 reflects structured runtime and integration failures rather than superficial syntax errors or compilation failures alone. By grounding coding tasks in executable game projects, GameEngineBench provides a concrete measure of coding-agent capability on game-engine development.

2 Related Work

Software Engineering Benchmarks. Software engineering has become a central domain for evaluating LLM-based coding agents. Earlier code-generation benchmarks such as HumanEval and MBPP measure isolated function synthesis [1, 3]. SWE-bench evaluates whether models can resolve real GitHub issues in conventional software repositories [11]; SWE-bench-Live and SWE-rebench emphasize freshness, continuous task collection, and contamination-resistant evaluation [2, 17]; Terminal-Bench evaluates agents on realistic command-line tasks that require operating inside an execution environment [13]. These benchmarks

[1] Some source projects use non-code assets during execution, but benchmarked edits are restricted to native C++ files.

Benchmark	Existing repo	Patch/edit task	Full project gen.	Game engine	Native C++	Server/client	Judge/rubric audit	Tasks
SWE-bench [11]	✓	✓	✗	✗	✗	✗	✗	2,294
DeepSWE [10]	✓	✓	✗	✗	✗	✗	✓	113
FrontierCode [12]	✓	✓	✗	✗	✗	✗	✓	150
ProgramBench [15]	✗	✗	✓	✗	✗	✗	✗	200
GameDevBench [4]	✗	✓	✗	✓	✗	✗	✗	333
AutoUE [16]	✗	✗	✓	✓	✗	✗	✗	–
GameEngineBench	✓	✓	✗	✓	✓	✓	✓	110

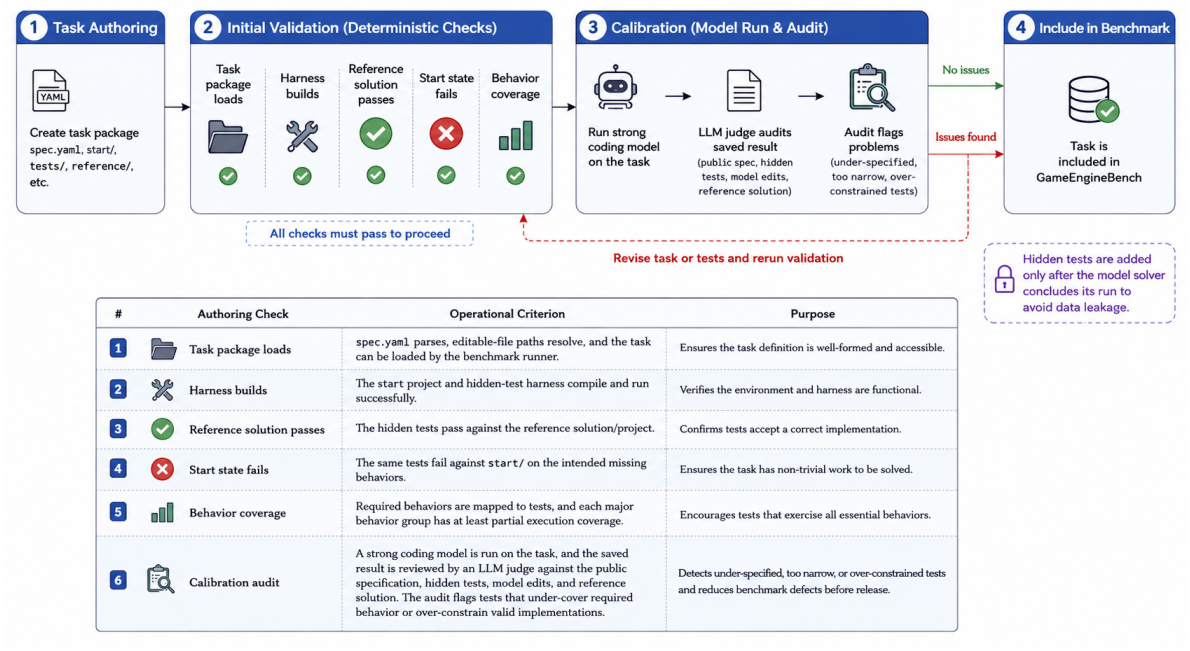
Table 1. High-level comparison with related coding-agent and game-development benchmarks. Checkmarks denote properties that are central to the benchmark design rather than incidental capabilities. GameEngineBench uniquely combines existing game-engine repositories, scoped native C++ edits, server/client correctness, and LLM-reviewed runtime behavior.

establish bug fixing and environment-level execution as important alternatives to isolated coding problems. GameEngineBench targets a different part of the software-engineering spectrum by evaluating larger implementation tasks inside functioning runtime systems, using game projects as the concrete execution environment. Across the 110 evaluated tasks, reference solutions add a mean of 511 lines and a median of 362 lines within the editable source files, making the tasks larger than the SWE-bench Verified and SWE-bench Pro reference edits reported by DeepSWE [10]. This shifts the evaluation from localized bug repair toward implementing missing behavior inside an existing runtime system.

Frontier Agent Benchmarks. Recent agent benchmarks have shifted from step-by-step implementation specifications to realistic behavior-focused instructions that require deeper reasoning and understanding. DeepSWE evaluates frontier coding agents on original, long-horizon software engineering tasks with broad repository coverage and behavior-focused verifiers [10]. ProgramBench evaluates whether agents can rebuild complete software projects from documentation and executable behavior, showing that agents often produce codebases that behave partially correctly but diverge sharply from human-written implementations [15]. FrontierCode shares the emphasis on realistic codebase tasks, but uses maintainer-authored tasks and rubric-based grading to shift the target from behavioral correctness alone toward mergeability, evaluating whether generated code would meet production codebase standards [12]. GameEngineBench follows this direction while focusing on runtime-integrated C++ programming.

Game Development Benchmarks. GameDevBench establishes game development as a meaningful domain for agent evaluation with tasks derived from web and video tutorials, emphasizing multimodal game-development work such as visual scenes, shaders, sprites, and animations [4]. AutoUE studies automated 3D game generation in Unreal Engine through multi-agent systems and automated play-testing, but targets end-to-end game creation rather than constrained C++ changes inside existing projects [16]. GameDevBench does not target multiplayer or server/client correctness, which are central sources of failure in networked game development. GameEngineBench instead evaluates native C++ changes inside existing Unreal projects. This focuses the benchmark on runtime-integrated programming, where code must work with existing gameplay logic, networking, and engine architecture.

Game-Engine Runtime Behaviors. We define four runtime behaviors that are fundamental to correctly solving Unreal Engine programming tasks. *Multiplayer authority* requires authoritative gameplay decisions to run on the server rather than on individual clients; this follows the client/server model used by Unreal networking [6]. *Replication* is Unreal's mechanism for synchronizing gameplay state and procedure calls between server and clients; property replication sends server-side updates to connected clients, which then apply those values to their local actor instances [6, 8]. *Object lifecycle* refers to the sequence of actor and object events around spawning, initialization, gameplay start, teardown, destruction, and garbage collection [5]. *Subsystem architecture* refers to Unreal-managed systems such as game-instance, world, and local-player



#	Authoring Check	Operational Criterion	Purpose
1	Task package loads	`spec.yaml` parses, editable-file paths resolve, and the task can be loaded by the benchmark runner.	Ensures the task definition is well-formed and accessible.
2	Harness builds	The `start` project and hidden-test harness compile and run successfully.	Verifies the environment and harness are functional.
3	Reference solution passes	The hidden tests pass against the reference solution/project.	Confirms tests accept a correct implementation.
4	Start state fails	The same tests fail against `start/` on the intended missing behaviors.	Ensures the task has non-trivial work to be solved.
5	Behavior coverage	Required behaviors are mapped to tests, and each major behavior group has at least partial execution coverage.	Encourages tests that exercise all essential behaviors.
6	Calibration audit	A strong coding model is run on the task, and the saved result is reviewed by an LLM judge against the public specification, hidden tests, model edits, and reference solution. The audit flags tests that under-cover required behavior or over-constrain valid implementations.	Detects under-specified, too narrow, or over-constrained tests and reduces benchmark defects before release.

Figure 1. Task authoring, test validation, LLM-as-a-Judge review, and task calibration process.

subsystems that the engine creates, initializes, and exposes to gameplay code [9]. Correct implementations often depend on coordinating these behaviors to produce a correct solution to a development task. The emphasis on cross-system behavior is consistent with prior work showing that game-engine subsystems are tightly coupled and hard to understand in isolation [14]. A compilable solution can still fail if it runs on the wrong machine, updates only local state, cleans up too late, or registers a component after another system expects it to exist.

3 Benchmark Design

Every task provides the model with a buildable starting state, a list of editable C++ files, and a public set of text instructions that define observable task-specific behavior. The test suite is withheld from the solver workspace and injected only after the model completes its implementation, preventing leakage of evaluation logic during the solve phase.

Before a task is included in the benchmark, we use a multi-step authoring process with two distinct checks. *Test validation* checks that the task package loads, the start project builds, the reference solution passes the tests, and the starting state fails those tests on the intended missing behaviors. *LLM-as-a-Judge review* runs a strong coding model on the task and asks an LLM judge to review the saved result against the public behavior specification, test outcomes, model edits, and reference solution. This review identifies tests that are under-specified, too narrow, or over-constrained relative to the requested behavior. When issues are found, we revise the task or test suite and repeat the validation process. We refer to this iterative pre-release workflow of task validation, judge review, task refinement, and revalidation as *task calibration*. Figure 1 summarizes the process.

Figure 2. Example tasks from GameEngineBench. Screenshots are taken from reference solutions to illustrate the gameplay and engine-system behaviors evaluated by the benchmark.

4 Current C++ Task Set

The current evaluated task set consists of 110 active tasks and is sourced from nine Unreal projects. The tasks focus on gameplay and systems programming within existing codebases, spanning gameplay mechanics, multiplayer behavior, AI and world orchestration, animation and character systems, UI and player-session systems, subsystem and plugin integration, runtime object management, data/configuration logic, movement systems, XR behavior, and rendering-oriented plugin code. The benchmark is not intended to cover every aspect of C++ game development; coverage is intentionally lighter for domains such as audio, performance, platform support, security, and tooling.

The tasks are sourced from nine open-source Unreal projects: HordeTemplateV2Native, ActionRoguelike, Bomber, TargetVector, GASShooter, Eternal Crusade Resurrection, LASAA, PBMovementBench, and NanoGSBench. We selected these repositories to maximize diversity in project context. Because each task remains embedded within its original codebase, the model must work with existing C++ code, engine callbacks, networking setup, and gameplay architecture instead of solving isolated programming problems. Figure 3 summarizes the distribution of tasks across game-development areas. Two sample tasks are described in Appendix A.

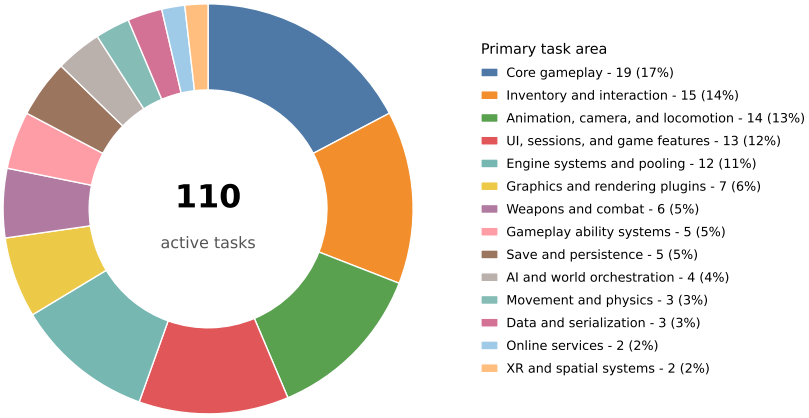


Figure 3. Primary game-development areas covered by the current task set. Categories are assigned once per task for readability; many tasks also exercise cross-cutting runtime requirements such as authority, state synchronization, object lifecycle handling, and subsystem initialization.

5 Evaluation Protocol

The primary evaluation metric used in this work is *pass@1*, defined as the fraction of tasks solved by a single model attempt. A run is counted as successful when the LLM judge determines that the generated implementation satisfies the requested behavior. This LLM-judged definition is necessary because the tasks often span several engine systems, making it impractical for any test suite to capture every valid implementation and interaction path.

The LLM review follows the LLM-as-a-Judge paradigm. It receives the behavioral specification, test source, execution results and assertion failures, model's code edits, and reference solution, then determines whether the implementation satisfies the intended task behavior [18]. The LLM's role is not to measure similarity to the reference solution, but to assess behavioral correctness and determine whether the test results accurately reflect that correctness. To mitigate self-judging bias, the evaluation protocol supports cross-family judging whenever multiple model families are available.

Each run is executed by the benchmark CLI in an isolated temporary workspace copied from the task's public `start/` state. The CLI invokes each agent through its public command-line wrapper with the same task payload: the behavior specification, the list of editable files, and the requirement that the modified project must compile successfully. Public CLI wrappers provide a reproducible interface while preserving each agent's standard tool-use behavior. After the solve phase, tests are injected and executed through Unreal's automation framework in Play-in-Editor (PIE) listen-server mode [7]. The benchmark preserves the resulting workspace, logs, and judge output for auditing. Staging the evaluation process reduces leakage, preserves reproducibility, and maintains a clear distinction between the public task specification, tests, and judge review.

Execution detail	Current setup
Evaluated model setups	`gpt-5.5` (`xhigh/high/medium`) + `codex`; `claude-fable-5` (`max`), `claude-opus-4-8` (`max/high`), `claude-opus-4-7` (`high`), and `claude-sonnet-4-6` (`high`) + `claude-code`; `Gemini 3.1 Pro` + Antigravity CLI; `DeepSeek 4 Pro` and `Qwen 3.7 Plus` + `qwen-code`; `Kimi for Coding` + `kimi-code`
Task mode	Unreal automation in PIE listen-server mode
Solve attempts	One per task/model pair; no retries or majority vote
Reasoning effort	Explicit where supported by the wrapper: `gpt-5.5` uses `xhigh/high/medium`; Claude configurations use `max/high`; other wrappers use their configured default.
Timeouts	Solver: 3600 s; compile and test: 600 s each
Scoring	pass@1

Table 2. Execution setup for the current evaluated task set.

6 Results

We report pass@1 as the primary evaluation metric. The results show that the benchmark is not saturated by current coding agents. The strongest evaluated configuration reaches 55.5% pass@1, while several other frontier configurations remain far below that level. The results also show that model capability is not strictly ordered: a configuration with higher overall pass@1 can still miss tasks solved by a lower-scoring configuration. In addition, 31 tasks are not solved by any evaluated configuration.

6.1 Success Rates

Figure 4 provides the main model comparison. The best evaluated configuration is `claude-fable-5` with `max` reasoning effort at 55.5% pass@1. The next strongest configuration is `GPT-5.5` with `xhigh` reasoning effort at 29.1%, followed by `claude-opus-4-8` with `max` effort at 23.6% and `Gemini 3.1 Pro` at 18.2%.

The gap between the strongest configuration and the rest of the evaluated agents is large. This suggests that the benchmark is sensitive to meaningful differences between model–harness setups rather than assigning similar scores to all frontier systems. At the same time, the strongest configuration still leaves nearly half of the benchmark unsolved. The task set is therefore not saturated: even the best current agent does not reliably solve C++ systems work inside existing Unreal projects.

The `GPT-5.5` reasoning-effort sweep gives a second view of this difficulty. Increasing reasoning effort from `medium` to `high` to `xhigh` raises pass@1 from 9.1% to 19.1% to 29.1%. Additional reasoning helps substantially, but it does not close the gap to the strongest configuration and does not make the benchmark easy. This indicates that some failures are recoverable through more search and deliberation, while many still require better handling of runtime behavior and project integration.

6.2 Efficiency Tradeoffs

Appendix Figure 6 shows the resource tradeoffs behind the pass@1 ranking. `claude-fable-5` gives the strongest pass@1, while `GPT-5.5` at `xhigh` is faster and less expensive in this run. `Gemini 3.1 Pro` is faster and cheaper than both, but with lower pass@1. These tradeoffs must be considered because the best agent may not be the most practical agent in the workflow. Cheaper agents can properly solve simpler tasks and can

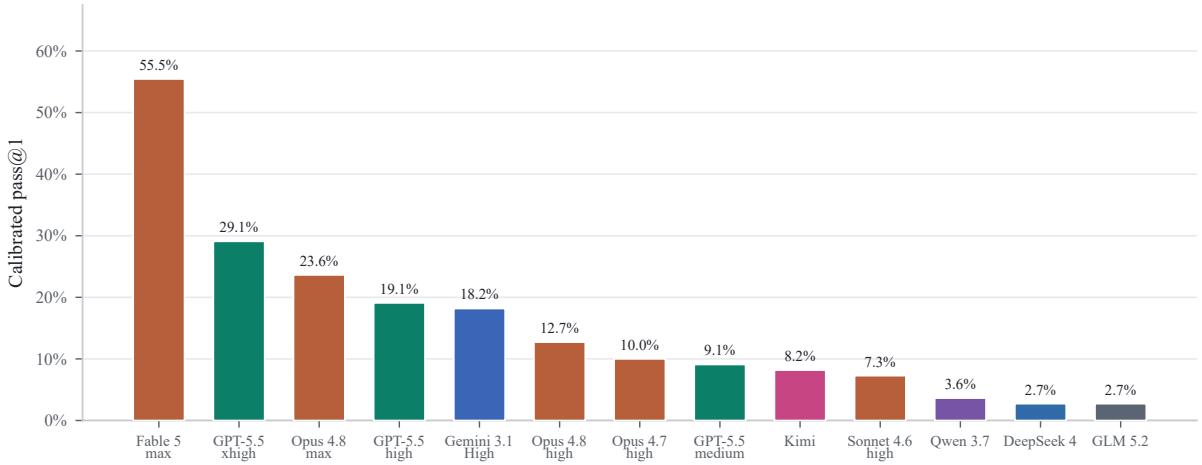


Figure 4. Success rate by model configuration over the active 110-task set. Each label includes the model and reasoning effort where applicable.

be deployed more at scale effectively. The efficiency of these models should be considered in conjunction with the pass@1 leaderboard.

6.3 Solved Sets Are Complementary

The leaderboard does not imply a strict nesting of capabilities. Appendix Figure 7 shows that 79 tasks are solved by at least one configuration, while 31 tasks are not solved by any of the twelve configurations. Conversely, no task is solved by all configurations.

The complementarity analysis reveals structure in model performance that is not visible from a single ranking. Some tasks are solved by only one or two configurations, indicating that changing models can recover specific behaviors. At the same time, the 31-task no-solve subset shows that model selection alone is insufficient to cover the benchmark. Overall, the current frontier is broad but incomplete: different agents succeed on different parts of the task set, while a substantial set of tasks remains unsolved across all evaluated configurations.

6.4 Coverage Varies By Task Area

The no-solve set is not uniformly distributed across task categories. Figure 8 groups tasks by task area and stratifies tasks solved by at least one configuration from unsolved tasks. In this view, save and persistence have the largest unresolved share. Furthermore, weapons/combat, serialization, UI/game-feature work, AI/world orchestration, animation/locomotion, engine systems/pooling, core gameplay, rendering plugins, and inventory/interaction all retain at least one unsolved task.

Tasks that remain unsolved for any configuration share a common failure mode: cross-system coordination. The SPUD persistence tasks require actor identity, destroyed actors, serialized properties, streaming levels, and save/load lifecycle code to remain consistent across sessions. Task 1 (Zombie System) requires AI control, round-state updates, server-authoritative damage, player-state rewards, replicated feedback, and spawn metadata to agree. Task 19 (Generated Map Orchestrator) requires generation, actor pooling, replication, and readiness signaling to stay synchronized. These tasks are difficult for agents because correctness requires coordination between several runtime systems, rather than the use of any single Unreal API call.

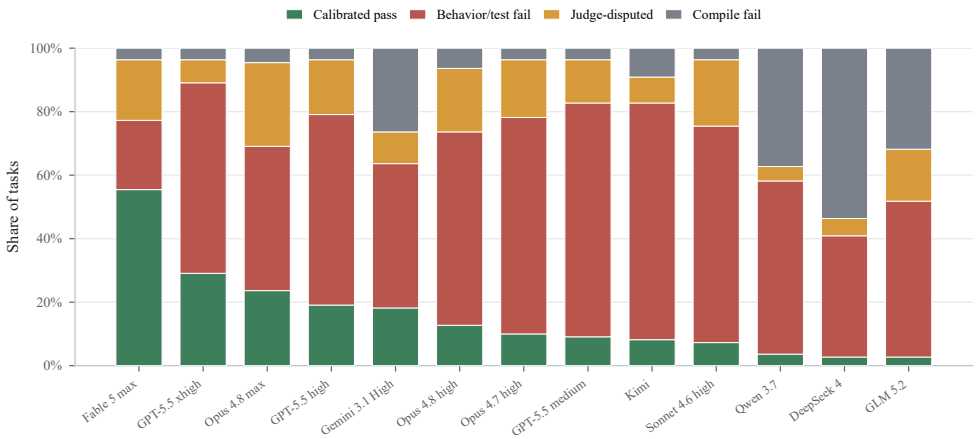


Figure 5. Outcome decomposition per configuration.

6.5 Failure Mechanisms

We uncover failure mechanisms from model-level and task-level results that explain the task-coverage gap. Figure 5 decomposes outcomes into passes, test failures, judge-disputed cases, and compile failures. The higher-performing wrappers usually reach compilation and execution; their remaining errors are mostly behavioral rather than purely syntactic.

For example, in active-item replication tasks, models occasionally add an `OnRep` hook but leave the client without the replicated source state needed to update visible UI. In player-controller and menu tasks, models can omit local-controller checks, allowing client-specific browsing or playback state to run on the wrong controller instance. In ability and action-system tasks, models often recover declarations but miss constructor defaults, activation timing, or teardown behavior. The generated output resembles plausible Unreal C++ code, but the runtime behavior is incomplete.

7 Limitations and Next Steps

The current task set provides a strong foundation for evaluating model capabilities on Unreal C++ development. The tasks offer broad coverage of gameplay-facing C++ systems within existing projects, including combat, inventory, networking, actor lifecycle, player/session flows, subsystem integration, and data/configuration behavior. However, the task coverage does not extend to other important areas of C++ game engineering, such as audio, memory, performance, platform support, editor tools, build systems, and security-sensitive code.

The choice of source repositories also imposes an inherent ceiling on coverage. Open-source Unreal projects typically expose gameplay and framework code more readily than proprietary engine modifications, rendering work, platform integrations, performance infrastructure, or security-sensitive systems. As a result, the current evaluated set captures a substantial but incomplete slice of professional C++ game engineering.

The execution harness introduces an additional boundary. Running tasks through PIE listen-server automation is a practical way to test runtime behavior, authority, replication, and lifecycle correctness, but it does not fully reflect production environments that rely on dedicated server fleets, console hardware, performance profiling, large QA matrices, or long-running live operations. Moreover, test suites are inherently imperfect due to not capturing all valid implementations, making evaluation sensitive to both test design and judge calibration.

Future expansion could expand coverage in two directions. First, the benchmark should include more tasks

that require coordination across multiple engine systems, which are common in game repositories but still remain difficult for frontier coding agents. Second, a separate expansion should incorporate underrepresented areas of C++ game development, such as audio, performance, editor tooling, build/platform code, and security-sensitive gameplay infrastructure. Together, these additions would improve representation while preserving the engine-integrated runtime challenges that define the benchmark.

We also identify two methodological extensions that strengthen evaluation reliability. First, because the evaluated models are run through different wrappers, the observed pass@1 differences partially reflect wrapper effects rather than pure model capability. Standardizing the execution wrapper will yield fairer comparisons. Second, LLM review can convert some test failures into valid solutions, showing that the tests do not always capture the full behavior space of these tasks. Improving test coverage and calibration would reduce reliance on judge intervention and produce a more direct execution signal.

8 Conclusion

GameEngineBench marks a concrete step toward benchmark design for AI-assisted C++ programming. Rather than evaluating agents on narrowly specified instructions, our benchmark requires generated outputs to align with behavioral requirements at runtime, thereby testing whether models understand context and correctly integrate with the surrounding game system. The tasks are extracted from executable Unreal Engine repositories, enabling evaluation of production-like C++ runtime behaviors while leaving several areas of game development for future expansion. The 110-task set shows that frontier coding agents do not reliably integrate with gameplay-facing C++ systems easily, failing to replicate runtime behavior despite meeting syntax and compilation requirements. While the benchmark is specific to Unreal Engine, the observed performance gaps reflect broader challenges in engine- and framework-integrated software development, where generated code must operate within existing systems, respect execution constraints, and preserve behavior under realistic runtime conditions. As a result, GameEngineBench serves both as a focused benchmark for challenging C++ game development and as a foundation for more comprehensive evaluations of programming within game engines in future work.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

[2] 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, 2025. URL https://arxiv.org/abs/2505.20411.

[3] 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, 2021. URL https://arxiv.org/abs/2107.03374.

[4] Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating agentic capabilities through game development, 2026. URL https://arxiv.org/abs/2602.11103.

[5] Epic Games. Actor lifecycle in Unreal Engine. Unreal Engine 5.8 Documentation, 2026. URL https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-actor-lifecycle. Accessed July 2, 2026.

[6] Epic Games. Networking and multiplayer in Unreal Engine. Unreal Engine 5.8 Documentation, 2026. URL https://dev.epicgames.com/documentation/en-us/unreal-engine/networking-and-multiplayer-in-unreal-engine. Accessed July 2, 2026.

[7] Epic Games. Play in editor multiplayer options in Unreal Engine. Unreal Engine 5.8 Documentation, 2026. URL https://dev.epicgames.com/documentation/en-us/unreal-engine/play-in-editor-multiplayer-options-in-unreal-engine. Accessed July 2, 2026.

[8] Epic Games. Replicate actor properties in Unreal Engine. Unreal Engine 5.8 Documentation, 2026. URL https://dev.epicgames.com/documentation/en-us/unreal-engine/replicate-actor-properties-in-unreal-engine. Accessed July 2, 2026.

[9] Epic Games. Programming subsystems in Unreal Engine. Unreal Engine 5.8 Documentation, 2026. URL https://dev.epicgames.com/documentation/en-us/unreal-engine/programming-subsystems-in-unreal-engine. Accessed July 2, 2026.

[10] Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. Datacurve technical report, 2026. URL https://deepswe.datacurve.ai/blog/deepswe. Published May 26, 2026.

[11] 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?, 2023. URL https://arxiv.org/abs/2310.06770.

[12] Eric Lu, Ben Pan, Deniz Birlikci, Sam Lee, Ray Wang, Rohan Choudhury, Fermi Ma, TC Qin, Carlo Baronio, and Silas Alberti. Introducing FrontierCode. Cognition research blog, 2026. URL https://cognition.com/blog/frontier-code. Published June 8, 2026.

[13] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

[14] Gabriel C. Ullmann, Yann-Gaël Guéhéneuc, Fabio Petrillo, Nicolas Anquetil, and Cristiano Politowski. Visualising game engine subsystem coupling, 2023. URL https://arxiv.org/abs/2309.06329.

[15] John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

[16] Lei Yin, Wentao Cheng, Zhida Qin, Tianyu Huang, Yidong Li, and Gangyi Ding. AutoUE: Automated generation of 3d games in Unreal Engine via multi-agent systems, 2026. URL https://arxiv.org/abs/2603.07106.

[17] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live!, 2025. URL https://arxiv.org/abs/2505.23419.

[18] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena, 2023. URL https://arxiv.org/abs/2306.05685.

A Sample Tasks

Two examples illustrate the kinds of Unreal work included in the current evaluated task set. We choose these examples because both remain pass@1 failures for all evaluated configurations and expose different forms of engine-level coordination.

Task 1 (Zombie System) is a broad multiplayer gameplay task in HordeTemplate. It requires coordinated edits across eight source files so zombies spawn through the round system, update the alive-zombie counter, detect and pursue living players, stop acting after death, enforce melee range, award kill and headshot rewards, and expose attack/death feedback to clients. The task is difficult because correctness depends on several systems agreeing at once: AI control, game-mode round progression, player-state rewards, server-authoritative damage, replicated feedback, and spawn/patrol metadata.

Task 19 (Generated Map Orchestrator) is narrower in file count but still integration-heavy. It requires a generated map actor to populate a procedural grid at round start, reuse pooled actors across regenerations, send one ready signal after local generation completes, replicate map-size changes, and keep peers synchronized. This task is difficult because the implementation must coordinate generation, actor pooling, replication, editor preview behavior, and readiness signaling without firing too early, firing twice, or leaking actors across regenerations. Together, these examples show that the benchmark's hardest tasks are not defined only by edit size; they require consistent behavior across interacting Unreal systems.

B Supplemental Result Figures

The main text uses the figures needed for the core findings. This appendix includes denser diagnostic views that support those findings without interrupting the Results narrative. These figures are intended as audit views: they expose alternative scoring lenses, task-level structure, metadata breakdowns, compiler behavior, and resource use behind the headline pass@1 table.

Figure 9 preserves the task-level detail that is hidden by aggregate pass@1. Rows show model configurations and columns show tasks. Long red columns identify tasks that remain unsolved across the evaluated configurations, while mixed columns identify tasks where model choice changes the outcome. This view is the basis for the capability-frontier and complementarity claims in the main text.

Figure 10 checks whether the benchmark difficulty is concentrated in particular kinds of game-engine work. Because several categories contain only a small number of tasks, these plots should be read as directional evidence rather than as precise estimates for each domain. They are most useful for seeing which areas contribute to the unsolved set and for guiding future task expansion.

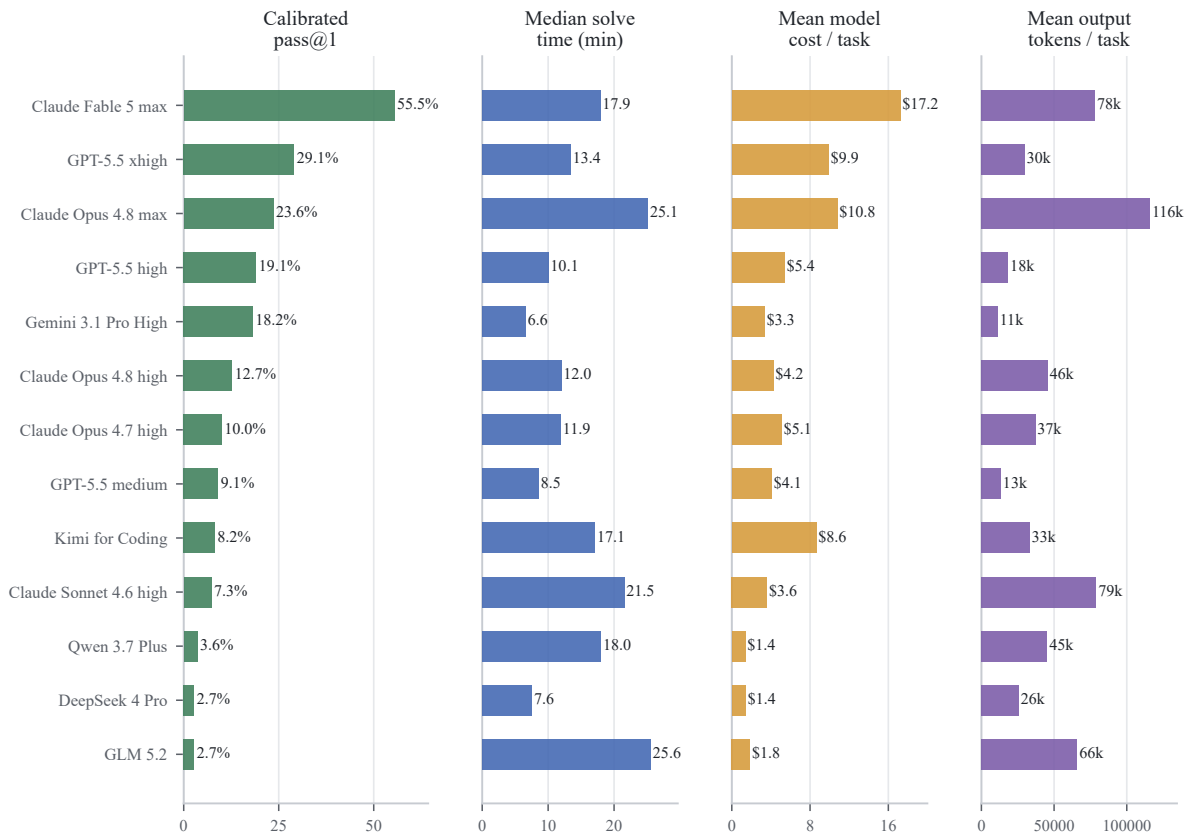


Figure 6. Efficiency profile across evaluated configurations. Rows are sorted by pass@1 and compare success rate with median solve time, mean model cost, and mean output tokens per task.

Figure 11 is the denser diagnostic version of the complementarity result in Figure 7. The difficulty curve shows that the benchmark contains both broadly solved tasks and a substantial no-solve region. The pairwise matrix then asks a different question: for each pair of configurations, how many tasks does one solve that the other misses? Large off-diagonal values mean that the leaderboard is not simply a nested ordering. Appendix C walks through two concrete task examples behind this pattern.

Figure 12 summarizes compiler use when the wrapper trajectory exposes it. The figure should be interpreted as a wrapper-visible behavior diagnostic, not as a complete comparison for every agent interface. It helps separate configurations that actively iterate through Unreal compilation from configurations where compile behavior is either less frequent or less visible in the saved trace.

Figure 13 provides the resource context for the efficiency discussion in the main text. Cost and token accounting are not equally complete for every wrapper, so these plots are best treated as available-data diagnostics rather than as a definitive pricing comparison. They still show that higher pass@1 is not free: stronger configurations often spend more inference, compilation, or review budget per task.

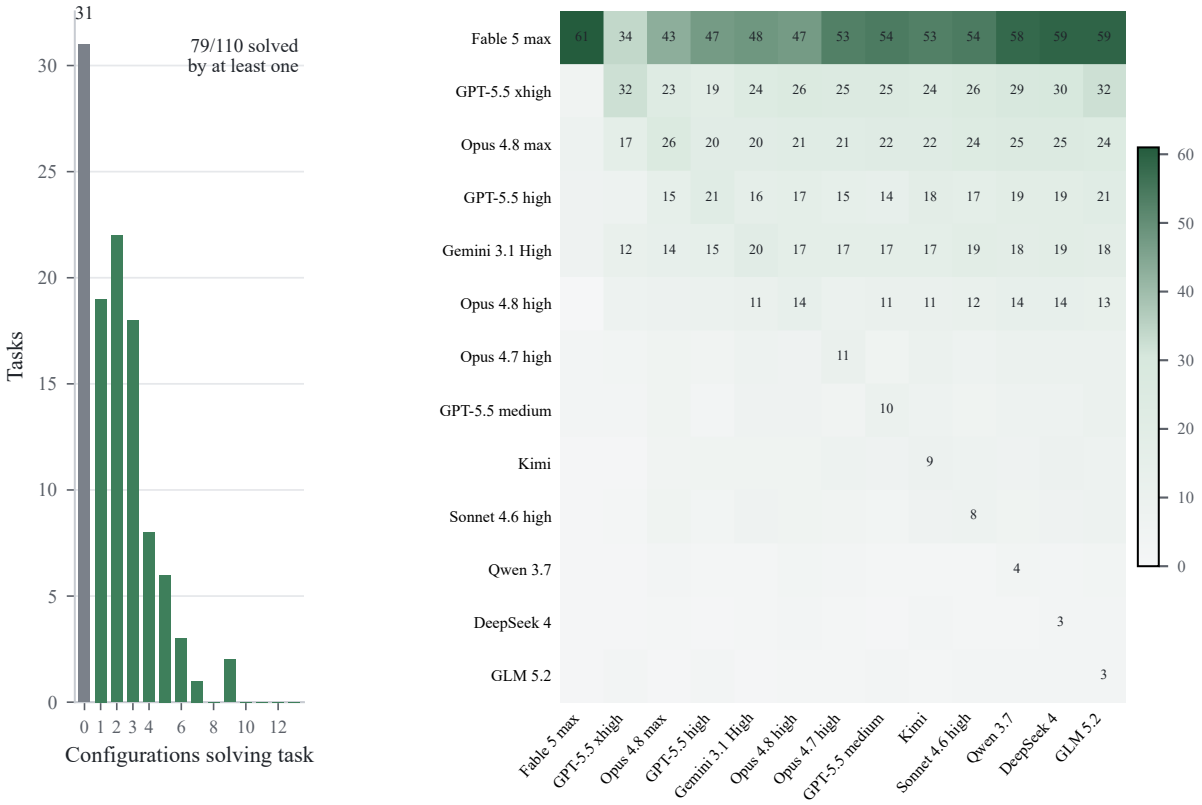


Figure 7. Task coverage and model complementarity. Left: how many configurations solve each task. Right: each off-diagonal entry counts tasks solved by the row configuration but missed by the column configuration; diagonal entries are own solved counts.

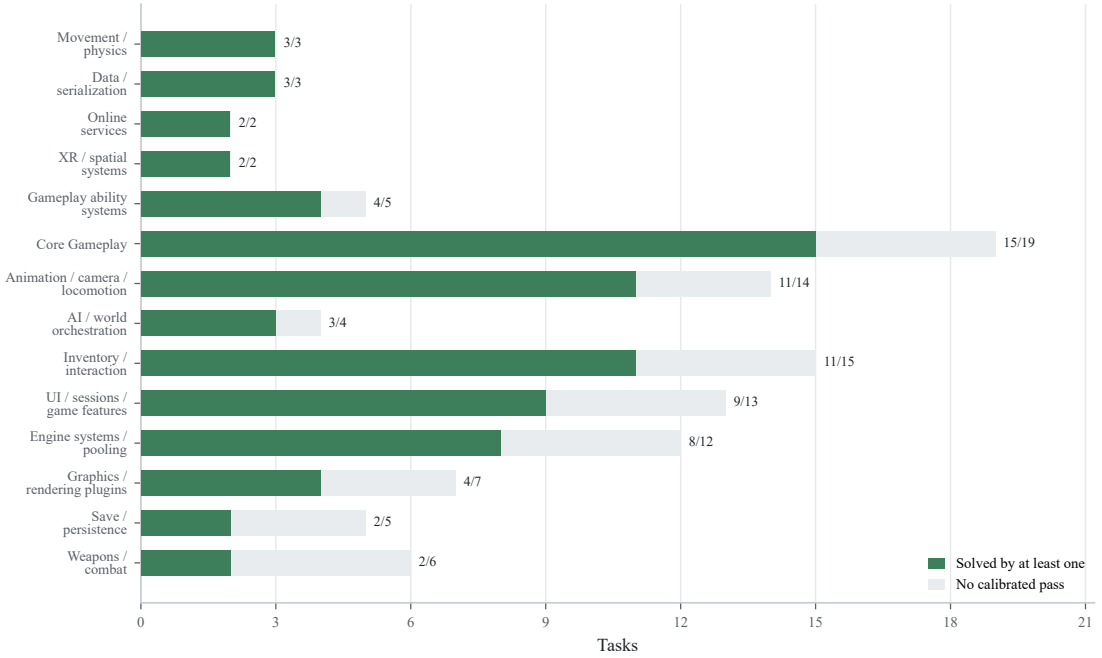


Figure 8. Task-area coverage under pass@1. Green segments count tasks solved by at least one evaluated configuration; gray segments count tasks with no successful runs. Labels at the end of each bar show the number of solved tasks over the total number of tasks in each area.

Figure 9. Per-task outcome matrix over the active 110-task set. Green cells are passes; red cells are non-passes. This dense view supports the capability-frontier analysis in the main text.

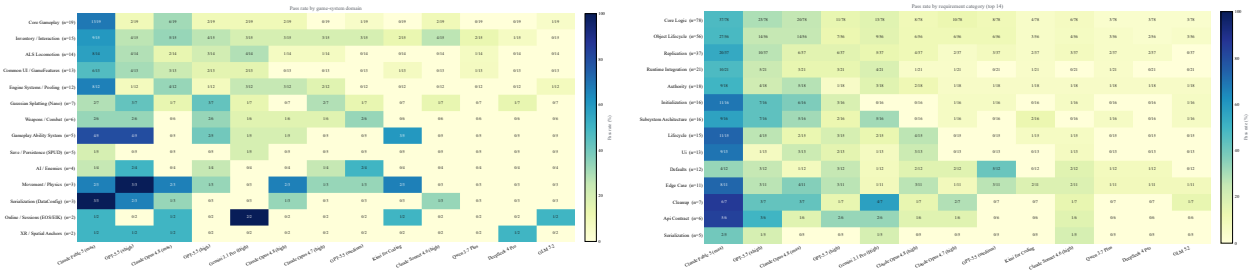


Figure 10. Supplemental pass@1 breakdowns by task metadata. Left: pass@1 by game-system domain. Right: pass@1 by requirement category.

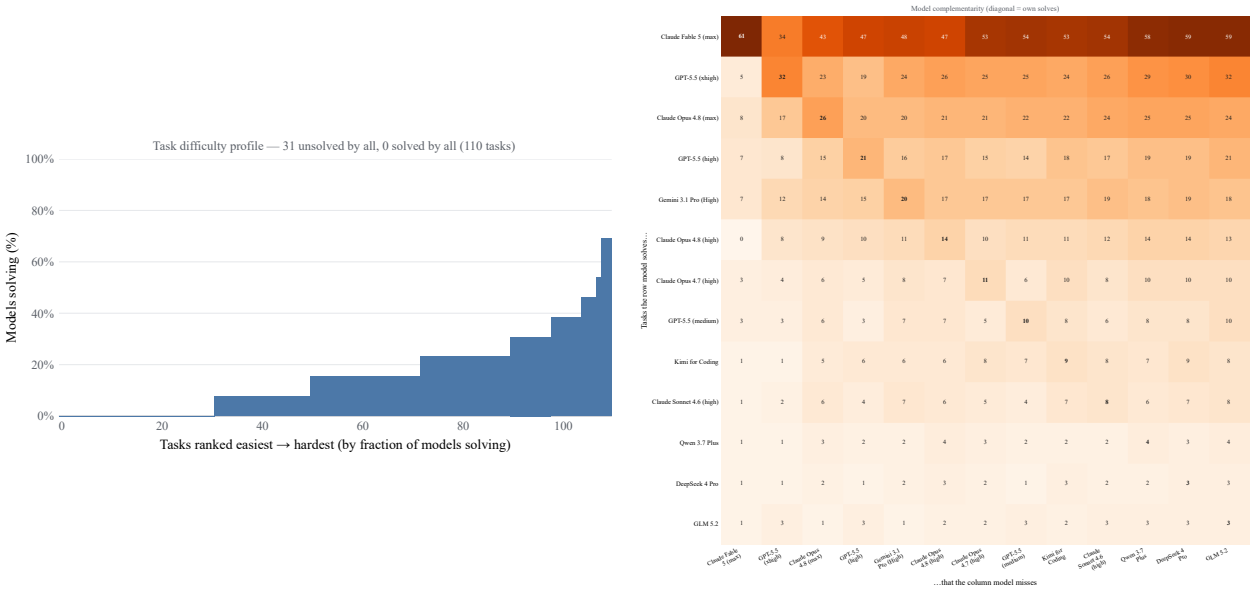


Figure 11. Supplemental task difficulty and complementarity views. Left: tasks ranked by fraction of configurations that solve them. Right: pairwise task complementarity between configurations.

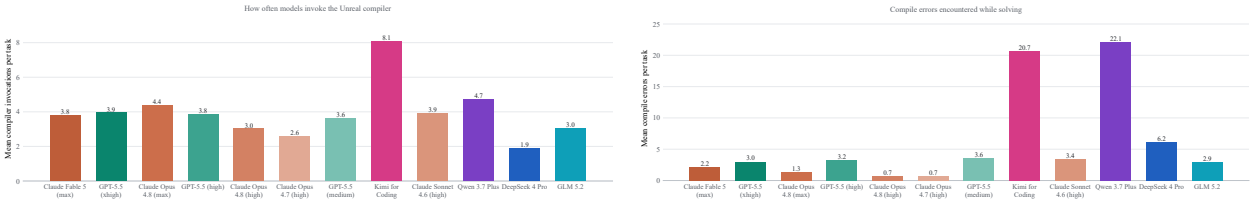


Figure 12. Supplemental compile diagnostics from available trajectories. Left: mean number of Unreal compiler invocations per task. Right: mean number of compile errors encountered during solving.

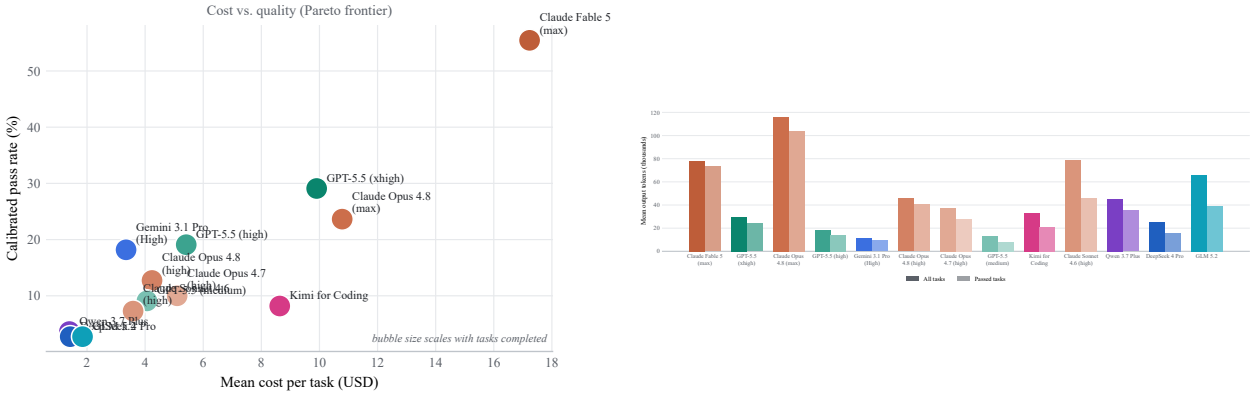


Figure 13. Supplemental resource-use diagnostics. Left: cost versus task-pass rate where cost data is available. Right: output tokens per task.

C Complementarity Examples

Figure 7 and Figure 11 show that model capability is not fully nested. This matters because a pure leaderboard view would suggest that a lower-ranked model adds little once a stronger model is available. The task-level matrix shows a different picture: some tasks are recovered by one configuration while being missed by configurations that are stronger overall. Complementarity therefore indicates that the benchmark is testing multiple kinds of game-engine reasoning rather than one scalar notion of C++ skill.

Task `15` (AI Controller + Behavior Tree) is solved only by `GPT-5.5` at `xhigh` in the active 110-task matrix. The task requires the agent to start an assigned behavior tree, register a blackboard observer after blackboard initialization, notify a player through a client RPC when the target changes, set team affiliation before pawn component registration, implement EQS target context lookup, and fill several behavior-tree services, tasks, and decorators. This is not a large rendering or UI task; it is an AI-control and initialization-order task where correctness depends on several behavior-tree and perception pieces becoming available at the right time. Its one-model success shows that `GPT-5.5` recovers at least some AI orchestration behavior that the other configurations miss.

Task `19` (Generated Map Orchestrator) shows the opposite direction. It is solved only by `claude-fable-5` at `max`. The task asks the model to populate a procedural grid at round start, run expensive cell assignment off the game thread, reuse pooled actors across regenerations, replicate generated-map containers and generation tokens, fire exactly one ready signal after local actors are present, and handle runtime map-size changes. This task combines replication, object lifecycle, pooling, asynchronous generation, and readiness signaling. Its one-model success shows why complementarity is useful: even the strongest non-Fable configurations miss some cross-system lifecycle and replication problems that another model can solve.

These examples make the off-diagonal entries in the complementarity matrix concrete. They are not just statistical noise around the pass@1 ranking. They correspond to qualitatively different task structures: behavior-tree initialization and AI control in one case, procedural map generation with pooling and replicated readiness in the other. For benchmark design, this means that additional model runs can reveal distinct capability pockets, while the no-solve tasks still identify behaviors that likely require better training, tooling, or task-specific reasoning.

D Reproducibility Notes

The current repository already contains the key scripts required to reproduce benchmark runs, including `GameEngineBench/src/ue_benchmark_runner.py` and `GameEngineBench/src/authoring/generate_task_tests.py`. The task packages live under `tasks_unreal/<task-id>/`, and saved benchmark snapshots live under `tasks_unreal/test_result/`. The main text reports the execution setup used for the evaluated task set; a stronger public release should bundle explicit OS, hardware, wrapper-version, and prompt-metadata files alongside the benchmark artifacts.