GAMEENGINEBENCH: FULL-SESSION VERIFICATION OF CODING AGENTS IN A NETWORKED GAME ENGINE

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents face new challenges in game development, where they must produce code that is not only syntactically correct but also consistent with engine-managed lifecycle, state, and networking behavior. Existing game-development benchmarks, however, do not combine assertions over engine-based runtime states with cross-client consistency, failing to evaluate how well generated code adheres to the desired behavioral specifications. To address these limitations, we present GameEngineBench, a benchmark of 110 C++ implementation tasks derived from nine existing Unreal Engine 5 projects. Each task requires an agent to implement missing behavior within an established gameplay architecture and a restricted editable scope. We also introduce FULL-SESSION VERIFICATION, a testing framework that injects hidden tests after the agent finishes and evaluates the resulting implementation inside networked Play-in-Editor (PIE) sessions. The tests evaluate Unreal's game loop, actor lifecycle, subsystems, and replication stack while directly inspecting runtime state. Across 17 benchmark tasks, tests additionally inspect a distinct client world to determine whether the authoritative server state is correctly replicated. To our knowledge, this is the first coding benchmark to evaluate cross-client replication. A detailed analysis of 768 mutated implementations finds that 74% of distributed-property bugs are detected only by end-to-end tests involving full PIE sessions. Across twelve model–harness configurations, the strongest achieves 55.5% pass@1, while 31 tasks remain unsolved by every evaluated configuration. These results demonstrate that compilation and shallow verification overestimate agents' ability to implement behavior that remains correct during full engine execution.

1 INTRODUCTION

Advances in coding agents have expanded their use beyond traditional software engineering tasks, leading to their increasing evaluation in complex game development settings (Chi et al., 2026; Luo et al., 2026; Sun et al., 2026; Lu et al., 2026; Yin et al., 2026). An engine such as Unreal Engine 5 (UE5) provides the framework that determines when, where, and how gameplay logic is executed. Various components, including actor lifecycle, engine subsystems, networking, rendering, UI, and scripts, must all be integrated for full correctness. For instance, a single feature may need to execute on the authoritative server, replicate its state to clients, update only for the local player, and clean up actors when they are destroyed or reused. The same feature may also have a visual component involving meshes, animations, particle effects, and HUDs. Although prior work has measured limitations in models' visual reasoning capabilities (Chi et al., 2026), whether their generated code behaves correctly inside a running game engine remains underexplored.

Most software engineering benchmarks evaluate features that are largely self-contained, allowing tasks to be independently verified. However, when multiple independent subsystems interact at runtime, coding agents must also satisfy correctness requirements that cannot be verified from the source code alone. We refer to behaviors that can only be verified at runtime as *non-local properties* and divide them into two categories: temporal and distributed. A temporal property concerns the order and timing of events within the engine's lifecycle. Temporal violations include querying a subsystem before the engine has been initialized or invoking a callback after its target has been destroyed. Although ordering constraints are well established in the program verification literature (Strom & Yemini, 1986; Ball & Rajamani, 2001; Meier et al., 2019), coding benchmarks typically

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.

address them indirectly through input–output predictions (Gu et al., 2024). Lifecycle misuse is stated as a major cause of Android crashes (Fan et al., 2018), while ordering violations constitute a major class of concurrency bugs (Lu et al., 2008). By contrast, a distributed property defines how the program state is maintained across separate runtime endpoints. In server–client applications, logic and state must remain consistent as updates are exchanged. Systems research has tested such distributed consistency through formal specifications of permitted behavior (Lamport, 2002; Cheng et al., 2026); however, coding-agent benchmarks generally do not test application-level replication across multiple running endpoints.

For both types of non-local properties, code may still compile and pass isolated unit tests while still failing during full-system execution. These violations cannot be detected reliably through source inspection or unit testing alone, leading to critical errors in production. Although prior distributed systems research provides methods for testing properties such as database isolation (Kingsbury & Alvaro, 2020), applying comparable verification to gameplay logic requires engine-aware test oracles and maintained behavioral specifications (Lamport, 2002). Hence, game engines provide a useful evaluation setting, providing large, stateful systems with numerous non-local invariants where violations can be observed directly with the appropriate runtime verification methods.

We present GameEngineBench, a benchmark of 110 C++ implementation tasks embedded in nine UE5 projects. For each task, the agent begins from a buildable start state and may edit only designated source files to satisfy a behavioral specification. To verify correctness, we introduce FULL-SESSION VERIFICATION. After the agent completes its implementation, hidden tests are injected then executed in a Play-in-Editor (PIE) listen-server session. PIE sessions run the entire game loop, including actor lifecycle, player controllers, and networking, allowing the test suite to inspect the engine-managed state at runtime and detect failures that arise only during a full session.

We evaluate GameEngineBench on twelve agent configurations. The strongest configuration solves 55.5% of the tasks, while 31 tasks remain unsolved by every evaluated configuration. We examine failure modes across temporal and distributed properties and make two contributions:

- GameEngineBench, a benchmark consisting of 110 C++ tasks drawn from nine UE5 projects, each requiring proper integration with a running game engine.
- FULL-SESSION VERIFICATION, a methodology that evaluates agent-generated code through controlled tests executed within a networked game session, providing, to our knowledge, the first cross-client replication testing in a coding benchmark.

2 RELATED WORK

Software Engineering Benchmarks. Evaluating coding agents on realistic software engineering tasks has become an active area of research. Most notably, SWE-bench evaluates whether models can resolve real GitHub issues in existing software repositories (Jimenez et al., 2023), while Terminal-Bench evaluates agents on realistic command-line tasks that require operating inside an execution environment (Merrill et al., 2026). A more recent benchmark, DeepSWE, handles long-horizon tasks with behavioral verifiers rather than inherited pull-request tests (Huang et al., 2026). While these benchmarks provide executable feedback, they do not specifically evaluate lifecycle-dependent or distributed behavior. AppForge evaluates applications in an Android emulator(Ran et al., 2025), while SWE-Lancer and ABC-Bench interact with applications through a browser or HTTP client (Miserendino et al., 2025; Yang et al., 2026). In contrast, GameEngineBench uses each game project as both the source repository and execution environment, allowing tests to inspect behavior managed by the engine during a complete gameplay session.

Game Development Benchmarks. GameDevBench establishes game development as a setting for evaluating agents on multimodal tasks involving visual scenes, shaders, sprites, and animations (Chi et al., 2026). AutoUE measures model capability in generating complete 3D environments within UE5 (Yin et al., 2026), while GameCraft-Bench, WorldCoder-Bench, and JAMER all evaluate game-development capabilities in other execution settings (Luo et al., 2026; Lu et al., 2026; Sun et al., 2026). Their verification methods, however, observe different aspects of correctness. GameDevBench primarily checks scene structure and callbacks rather than behavior under networked execution. GameCraft-Bench executes Godot games but uses a multimodal judge as ground truth, lacking any nuanced assertions other than a successful launch. AutoUE similarly evaluates generated levels

Table 1: Comparison of related coding benchmarks.

Benchmark	Test-based behavioral verdict	Runs the game loop	Asserts on runtime-owned state	Second participant
SWE-bench (Jimenez et al., 2023)	✓	✗	✗	✗
AppForge (Ran et al., 2025)	✓	✓	✗	✗
GameDevBench (Chi et al., 2026)	✓	✗	✓	✗
GameCraft-Bench (Luo et al., 2026)	✗	✓	✗	✗
WorldCoder-Bench (Lu et al., 2026)	✓	✓	✗†	✗
JAMER (Sun et al., 2026)	✗	✓	✗	✗
GameEngineBench	✓	✓	✓	✓

through screenshot-based judging. WorldCoder-Bench performs runtime tests in a browser, but the generated program is responsible for exposing the state. JAMER checks for integrity, compilation, and crashes while evaluating behavioral correctness through similarity to a reference. In contrast to prior benchmarks, GameEngineBench directly inspects engine-managed state during execution, combining runtime assertions with verification across separate server and client worlds for the first time.

Verification Work. Reliable verification is necessary to minimize false positives and false negatives in measuring coding agent performance (Zhu et al., 2025). Related verifier-side work on browser games injects runtime instrumentation and checks behavioral predicates(Jia et al., 2026). UTBoost, a test augmentation framework, finds deficient test cases on coding agent benchmarks that result in incorrect labels of task success and failure (Yu et al., 2025; Wang et al., 2025). SWE-ABS strengthens tests with coverage and mutation tests and reveals false successes that were previously attributed to models (Yu et al., 2026). Agents may also exploit visible tests rather than implement the intended behavior(Zhao et al., 2026; Zhong et al., 2025). GameEngineBench therefore withholds its tests during the solve and injects them only after agent implementation. Our mutation analysis complements prior work by measuring which defects become observable only when verification progresses from static tests to full engine execution.

3 GameEngineBench

Each task in GameEngineBench is a feature originally from a working UE5 project, and thus every task has a reference implementation that ran in the original game and inherits the project's build and assets. This section describes how tasks are formulated, how the benchmark is constructed and calibrated, and what the evaluated task set covers. Section 4 describes the verification and evaluation pipeline in detail.

3.1 Task Formulation

The coding agent must implement the missing behavior according to the behavioral specification provided. The agent begins with a buildable state, a list of editable files, and the specification, and must integrate with the project's existing architecture to complete the task. After the agent finishes, the agent output is evaluated using a hidden test suite consisting of three methodologies, as appropriate to the task requirements: direct API, integration, and end-to-end (E2E) tests. These tasks are substantially larger than those in issue-resolution benchmarks, with reference solutions spanning a median of 362 lines across several files.

3.2 Benchmark Construction

Tasks are sourced from public UE5 repositories chosen for gameplay diversity and code quality. The tasks come from nine projects: HordeTemplateV2Native, ActionRoguelike, Bomber, TargetVector, GASShooter, Eternal Crusade Resurrection, LASAA, PBMovementBench, and NanoGSBench. For each repository, we identify existing features in the codebase and remove their implementations from the source files. We then construct a behavioral specification that describes the missing feature. To

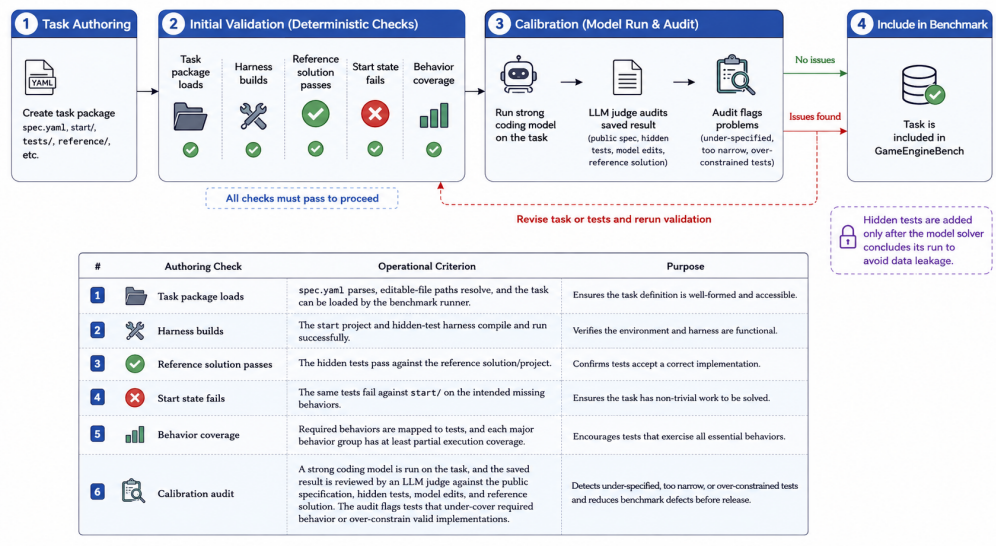


Figure 1: Task authoring, test validation, LLM-as-a-Judge review, and task calibration process.

solve the task, the model must reason through the existing C++ code and gameplay architecture rather than solving isolated programming problems.

To ensure the quality of our tasks, 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 behavior specification, test outcomes, model edits, and reference solution. This review identifies tests that are under-specified 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 as *task calibration*.Figure 1 summarizes the process. After this process, two game development experts with industry Unreal Engine experience manually review the task for validity.

3.3 Dataset Composition

The evaluated set consists of 110 active tasks focusing on gameplay and systems programming within existing codebases, covering 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 designed to incorporate every facet of C++ game development; coverage is intentionally lighter for domains such as audio, performance, platform support, security, and tooling.

Two sample tasks are described in Appendix A.

4 Verification and Evaluation

GameEngineBench employs four verifier classes with increasing execution fidelity: *static*, *unit*, *integration* (synthetic world), and end-to-end (*E2E*). Static checks serve as a non-execution baseline, while the other three tiers execute the agent's implementation in progressively more complete Unreal Engine contexts. For each behavioral requirement, we use the lowest verification tier that can observe the required property. Therefore, tiers differ in the observability of certain non-local properties rather than strictness alone. A verification pipeline cannot establish a property if the relevant state, lifecycle event, or network endpoint does not exist in its context.



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

4.1 Verification Tiers

Static checks inspect source declarations, code patterns, and Unreal macro annotations (such as `UPROPERTY` replication specifiers) and class metadata without code execution. Static checks can be useful in checking for missing interfaces and annotations, but they cannot establish that the specified behavior occurs. A solution may contain the expected function declaration while failing to initialize, update, or replicate the corresponding state.

Unit tests executes functions or isolated object instances without creating a full game world. They can verify local computations, constructor defaults, and state transitions whose correctness does not depend on the engine lifecycle. Without a running `UWorld`, however, they cannot observe temporal properties (no lifecycle runs) or distributed properties (no networking).

Integration tests spawn actors in a synthetic `UWorld`, allowing Unreal Engine to manage component registration, subsystem initialization, and other single-world lifecycle behavior. This tier can detect temporal violations, such as a component being registered at the wrong stage (e.g., `BeginPlay`) or an actor querying a subsystem before it is available. However, it cannot observe distributed properties within a single world context.

E2E tests execute a complete, networked PIE session, including the game loop, actor lifecycle, player controllers, gameplay subsystems. This tier is necessary when correctness depends on replicating and updating game states across multiple clients interacting through a shared authoritative server. The server and each of its clients have their own worlds, and the tests verify that client interactions are propagated correctly and the server state maintains consistency. Through the full PIE session, the test suite can observe both types of non-local properties, such as ordering constraints through the full lifecycle and distributed consistency through server-client replication. 17 benchmark tasks in GameEngineBench directly inspect the client world to verify distributed behavior.

The distinction between tiers can be illustrated by a health feature. A static check can identify a replication annotation, a unit test can validate the damage calculation, and an integration test can verify that damage is applied to an actor in a world. Only a networked E2E test, however, can establish that the server produces the correct value, Unreal transmits it through its networking system, and a separate client receives the corresponding state. Our tiered design overcomes the limitations of evaluating specified runtime properties solely on source structure or isolated local behavior.

4.2 Full-Session Verification

Full-Session Verification evaluates agent-generated code inside a complete PIE session under Unreal's listen-server mode (Epic Games, 2026). Unreal is particularly useful for this purpose because networked PIE sessions combine full engine execution with direct access to multiple runtime worlds. When configured to run under one process, PIE creates an authoritative server world and one or more separate client worlds connected through Unreal's built-in networking system. Each world

has its own actors, controllers, game state, and player state. The verifier can inspect corresponding objects across these worlds during the same test, while state must still pass through Unreal's ordinary property-replication and remote-procedure-call mechanisms.

This arrangement provides a stronger oracle than inspecting source code, observing a single game instance, or asking the program to report its own state. FULL-SESSION VERIFICATION can distinguish code that merely marks a specified property for replication from code that updates the property on the server, transmits it to the appropriate clients, and triggers the required client-side behavior. The same environment exposes lifecycle-dependent behavior involving initialization, possession, ticking, callbacks, destruction, and cleanup. Therefore, the pipeline benefits not only from Unreal Engine's production-relevant native C++ environment, but also from PIE's ability to execute the complete gameplay lifecycle and directly observe state across networked worlds within an integrated testing environment. The verification pipeline consists of four stages.

Test injection. After the agent completes its implementation, the benchmark runner validates the editable scope and injects the hidden test files into the project's test directory. We perform an additional contamination check that the agent did not have access to the test files during runtime. The runner then adds the required test-module dependencies and compiles the submitted implementation alongside the hidden tests.

Networked execution. We use Unreal's automation framework to launch the project headlessly. For networking, we define `-ListenServer` and `-NumClients=N` to have the automation run in listen-server mode and create N additional client worlds within the same process, each with its own `UWorld`, `APlayerController`, and local player state, connected to the server via loopback. Tests use controlled fixtures where applicable, including fixed spawn locations, seeded data, scripted actions, and explicit gameplay-state transitions. Controlling certain aspects of the game environment reduces the dependence on manual input and random placement.

Distributed consistency. In 17 tasks, we perform a specified action and wait for the relevant state to replicate to connected clients. Unreal Engine is not a fixed-step deterministic simulator, as frame durations, AI updates, and network delivery may complete on different ticks across runs. Full-Session Verification, therefore, does not require identical execution traces. Instead, latent automation commands yield control to the engine and poll a readiness predicate on successive ticks until the expected state appears or an explicit timeout is reached. This allows gameplay, lifecycle callbacks, and replication to progress without blocking the game thread for testing. Assertions emphasize final-state invariants, event ordering, monotonic progress, and agreement across worlds rather than exact frame counts, leading to state consistency between client and server.

Post-execution. Finally, the runner records compilation status, per-test and per-subcheck outcomes, automation logs, and the resulting workspace for audit. Upon concluding the PIE session, the runner removes injected test files from the saved solver artifact and restores the relevant editor settings to maintain consistent environments across tests.

4.3 EVALUATION PROTOCOL

We score agent performance with *pass@1*. Each configuration attempts every task once, and a run counts as successful only if the submitted implementation compiles and every test in that task's hidden suite passes. No other signal contributes to the benchmark label. Verdicts are read from the per-test and per-subcheck outcomes recorded in the automation log, so a score is determined entirely by executable results and can be recomputed from the archived workspace and test suite with no model in the loop.

Because the label is executable, the validity of the benchmark rests on the test suites rather than on an adjudicating model. Three properties of the suites carry that weight. First, tests are anchored to behavioral requirements rather than to the reference implementation: each task specification enumerates hidden requirements, and every test declares which requirements it observes, so a suite asserts the behavior a requirement names rather than the particular way the original author produced it. Second, every task passes positive and negative controls before release: the reference solution must pass all required tests, and the incomplete start state must fail precisely on the behaviors that were removed. Third, we characterize what each suite is able to detect using seeded faults whose ground truth is fixed by construction (Section 5.5).

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

Execution detail	Setup
Evaluated agents	`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; `DeepSeek 4 Pro` and `Qwen 3.7 Plus` + `qwen-code`; `Kimi for Coding` + `kimi-code`
Task mode	Unreal automation in PIE listen-server mode
Max solver time	3600s
Max compile time	600s
Scoring	pass@1

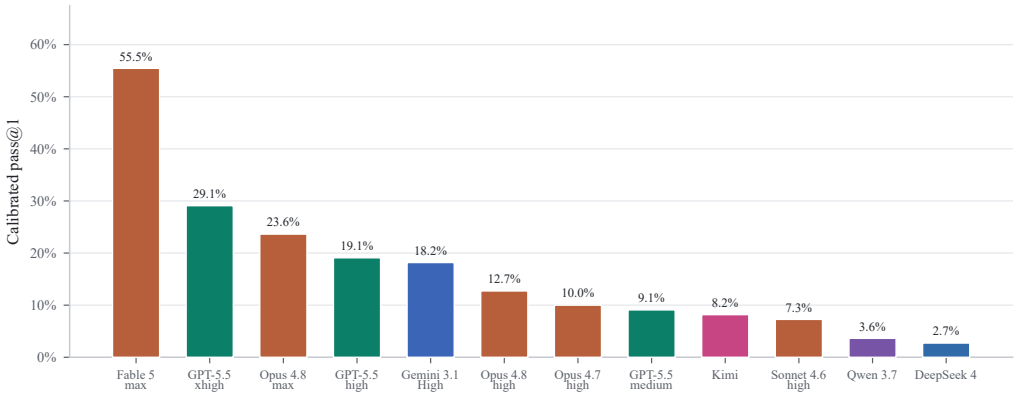


Figure 3: Success rate by model configuration over the active 110-task set. Each label includes the model and reasoning effort where applicable.

5 Results

The results show that the benchmark is not saturated by current coding agents and remains difficult. The strongest coding agent evaluated only solves 55.5% of the tasks. In addition, other agents remain far below that threshold and 31 tasks are not solved by any coding agent tested.

5.1 Success Rates

Figure 3 provides the main model comparison. The best evaluated configuration is `claude-fable-5` with `max` reasoning effort at 55.5% pass@1. The next three strongest configurations are `GPT-5.5` with `xhigh` reasoning effort, `claude-opus-4-8` with `max` effort, and `Gemini 3.1 Pro` with a pass rate of 29.1%, 23.6%, and 18.2% respectively. Notably, we notice a jump in performance with `claude-fable-5`.

If we analyze how the reasoning effort affects the capability of `GPT-5.5`, we see that increasing reasoning effort from `medium` to `high` to `xhigh` raises pass@1 from 9.1% to 19.1% to 29.1%. Additional reasoning helps coding agents, but does not close the gap to the strongest configuration or reduce the difficulty level. This suggests, however, that a portion of the failures are fixed with more thinking and deliberation.

5.2 Efficiency Tradeoffs

We compare the agent success rate against the cost in Appendix Figure 5a. We see that `claude-fable-5`, `GPT-5.5` at `xhigh`, `Gemini 3.1 Pro` have varying capability, cost, and speed. While `claude-fable-5` is the most capable, it is also the most expensive. While `Gemini 3.1 Pro` is the fastest and cheapest, it is also the least capable. Meanwhile, `GPT-5.5` at `xhigh` lies

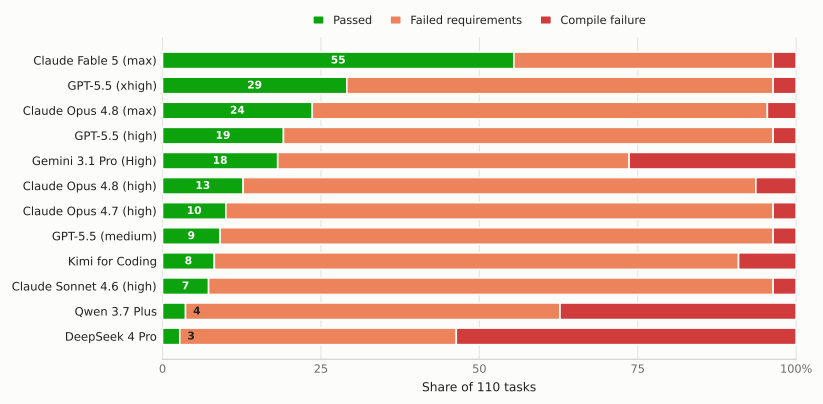


Figure 4: Outcome decomposition per configuration.

Table 3: Per-tier sensitivity: fraction of known-wrong mutants in each property class caught by each tier.

Property class	**n**	**Static**	**Unit**	**Integration**	**E2E**	**E2E only**
Local logic	277	39%	39%	82%	86%	10%
Cross file	127	59%	24%	67%	92%	28%
Temporal	118	47%	18%	70%	91%	25%
Distributed	96	60%	11%	15%	92%	74%
Hollow	150	78%	35%	78%	94%	21%
All	768	54%	29%	68%	90%	26%

in between them. These trade-offs must be considered, as capability is not equivalent to practicality. Cheaper agents can solve simpler tasks and can be much more cost-effective for production.

5.3 Solved Sets Are Complementary

We show that no coding agent dominates all task types in Appendix Figure 6. A stronger agent configuration, one that has an overall higher pass rate is not able to solve all of the tasks a weaker one does, showing that each model and agent configuration is strong at a different subset of the tasks. Some tasks are solved by only one or two configurations, indicating that changing models can recover specific behaviors. This potentially suggests that the best coding agent may be an agent that is handled through a model router, rather than a single coding agent.

5.4 Failure Mechanisms

We break down to see how agents fail the task in Figure 4. Tasks can fail from not compiling or by failing the test suite. We see that the stronger agents usually compile, but fail to satisfy all of the necessary behaviors. For example, agents may recognize to attach an `OnRep` hook, but then not add the necessary replicated state to complete the behavior.

5.5 Verification Strength

We evaluate how well each verification tier detects incorrect code. For 108 tasks, we create a set of *mutants*: solutions with a single bug intentionally injected. Each mutant is run against the task's full test suite for each verification tier, showing which tiers catch the bug. We classify mutants by the property the bug violates (Table 3).

Deeper tiers catch the most bugs. E2E tests catch 90% of the mutants, compared with 68% for integration and 29% for unit tests, and 26% of the mutants evade both unit and integration tests and are only exposed at the E2E tier. This is especially true for distributed-property violations: unit tests

catch only 11% and integration tests only 15%, while E2E tests catch 92%, so 74% are only caught by E2E tests. Temporal bugs show a similar pattern, with E2E at 91% versus unit at 18%. Static checks catch 54% of the mutants because each mutant alters the source they inspect, but as shown below, they also reject many correct solutions. These results show that entire categories of bugs are invisible to shallower verifiers.

Shallower verifiers have higher false alarm rates. The variation across tiers is not limited to strictness, but extends to accuracy. We score human-reviewed 195 solver runs (39 tasks, five configurations) with each tier's suite and use the human-reviewed E2E verdicts as the reference label. Static checks over-reject ($P(\text{fail} \mid \text{E2E pass}) = 33\%$), flagging correct solutions that do not match expected source patterns. Unit tests over-accept ($P(\text{pass} \mid \text{E2E fail}) = 26\%$), passing solutions whose signatures look right but whose behavior is not. Integration tests disagree with E2E tests far less often (3% and 8%).

Rankings based on shallow verifiers mask failure. How a configuration ranks depends heavily on which verifier is used. On the same 39 tasks, four of the five configurations exhibit an 18–38 percentage point swing in success rate across verifiers. For example, `GPT-5.5 (xhigh)` scores 54% under unit tests, close to `claude-opus-4-8 (max)` at 59%, but drops to 28% under E2E tests. A benchmark that relied only on static or unit verification would conclude that this configuration is strong, when in fact it fails to satisfy behavioral requirements under full execution. The exception is `claude-fable-5`, which shows only an 8-percentage-point spread and leads on E2E, suggesting genuine behavioral correctness rather than verifier overfitting. Relying on weaker verifiers would systematically overestimate models that satisfy shallow checks while missing runtime failures, skewing rankings and masking the true capability gap.

6 Limitations and Next Steps

While the current benchmark provides a strong foundation for evaluating model capabilities a broad range of gameplay-facing tasks, including combat, inventory, networking, actor lifecycle, player/session flows, subsystem integration, and data/configuration behavior, the task coverage is incomplete. Audio, memory, performance, editor tools, build systems, and security-sensitive systems remains underrepresented. This distribution partly reflects the availability of public Unreal projects, which more commonly expose gameplay code than engine modifications, rendering work, platform integrations, performance infrastructure, or security-sensitive systems. Future versions should expand these areas while preserving the benchmark's emphasis on evaluating engine-managed behavior.

7 Conclusion

GameEngineBench marks a concrete step toward advancing coding-agent evaluation from source-level plausibility to verified behavior within a running, stateful system. The110-task set require agents to integrate missing functionality into existing Unreal Engine projects, while hidden tests evaluate the resulting implementations through progressively stronger verification tiers. At the highest tier, Full-Session Verification executes the complete gameplay lifecycle in PIE and directly inspects engine-managed state. In 17 tasks, the pipeline is used to verify behavior in a separate client world, exposing failures that compilation, source inspection, and single-world tests cannot observe. The results show that current model–harness configurations frequently produce plausible and compilable code without reproducing the required runtime behavior. These findings demonstrate the importance of evaluating generated code within the lifecycle, subsystem, and networking contexts in which it must operate. Although the measured results are specific to Unreal Engine and the evaluated task set, the verification principle applies more broadly: code for stateful frameworks should be tested within the runtime contexts that define its behavior. GameEngineBench provides both a challenging benchmark for native game-engine development and a foundation for evaluating coding agents in other lifecycle-dependent and distributed software systems.

AI use statement

We used large language models to assist in rewording the paper for clarity and grammar. Coding assistants were also used to implement the benchmark harness and test suites. All benchmark tasks,

behavioral specifications, hidden tests, and experimental results were authored and verified by the authors, with slight assistance from large language models. All AI-assisted code was reviewed and executed against the validation controls described in Section 4. All research claims, analyses, or reported numbers that were generated by AI systems underwent human review. We take responsibility for the final content of this work.

REFERENCES

Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software (SPIN 2001), Lecture Notes in Computer Science*, pp. 102–122, 2001. doi: 10.1007/3-540-45139-0_7.

Qian Cheng, Saad Mohammad Rafid Pial, Ruize Tang, Yiming Su, Emilie Ma, Finn Hackett, Ivan Beschastnikh, Yu Huang, and Tianyin Xu. Specula: Scaling formal specifications for autonomous model checking of system code. *arXiv preprint arXiv:2607.25333*, 2026.

Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating agentic capabilities through game development. *arXiv preprint arXiv:2602.11103*, 2026. ICML 2026.

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.

Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 408–419, 2018. doi: 10.1145/3180155.3180222. arXiv:1801.07009.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024. ICML 2024.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

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

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? *arXiv preprint arXiv:2310.06770*, 2023. ICLR 2024.

Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proceedings of the VLDB Endowment*, 14(3):268–280, 2020. doi: 10.14778/3430915.3430918.

Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-321-14306-X.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 329–339, 2008. doi: 10.1145/1346281.1346323.

Shuo Lu, Yinuo Xu, Kecheng Yu, Siru Jiang, Yongcan Yu, Yubin Wang, Haitao Yang, Yuxiang Zhang, Bin Wang, Ran He, and Jian Liang. WorldCoder-Bench: Benchmarking physically grounded 3D world synthesis. *arXiv preprint arXiv:2606.01869*, 2026.

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.

Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-driven protocols and callback control flow. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 1:1–1:29, 2019. doi: 10.4230/LIPIcs.ECOOP.2019.1. Extended version: arXiv:1906.04924.

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. *arXiv preprint arXiv:2601.11868*, 2026.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? *arXiv preprint arXiv:2502.12115*, 2025.

Dezhi Ran, Yuan Cao, Mengzhou Wu, Simin Chen, Yuzhe Guo, Jun Ren, Zihe Song, Hao Yu, Jialei Wei, Linyi Li, Wei Yang, Baishakhi Ray, and Tao Xie. AppForge: From assistant to independent developer – are GPTs ready for software development? *arXiv preprint arXiv:2510.07740*, 2025. ICLR 2026.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986. doi: 10.1109/TSE.1986.6312929.

Jianwen Sun, Chuanhao Li, Zizhen Li, Yukang Feng, Fanrui Zhang, Yifei Huang, Yu Dai, and Kaipeng Zhang. JAMER: Project-level code framework dataset and benchmark on professional game engines. *arXiv preprint arXiv:2606.19830*, 2026.

You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? An empirical study. *arXiv preprint arXiv:2503.15223*, 2025. doi: 10.1145/3744916.3764576. ICSE 2026.

Jie Yang, Honglin Guo, Li Ji, Jiazheng Zhou, Rui Zheng, Zhikai Lei, Shuo Zhang, Zhiheng Xi, Shichun Liu, Yuxin Wang, Bo Wang, Yining Zheng, Tao Gui, and Xipeng Qiu. ABC-Bench: Benchmarking agentic backend coding in real-world development. *arXiv preprint arXiv:2601.11077*, 2026. Findings of ACL 2026.

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. *arXiv preprint arXiv:2603.07106*, 2026.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on swe-bench. *arXiv preprint arXiv:2506.09289*, 2025. ACL 2025.

Boxi Yu, Yang Cao, Yuzhong Zhang, Liting Lin, Junjielong Xu, Zhiqing Zhong, Qinghua Xu, Guancheng Wang, Jialun Cao, Shing-Chi Cheung, Pinjia He, and Lionel Briand. SWE-ABS: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark. *arXiv preprint arXiv:2603.00520*, 2026.

Bingchen Zhao, Dhruv Srikanth, Yuxiang Wu, and Zhengyao Jiang. SpecBench: Measuring reward hacking in long-horizon coding agents. *arXiv preprint arXiv:2605.21384*, 2026.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270*, 2025.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, Fazl Barez, Rahul Gupta, Jwala Dhamala, Jacob Merizian, Mario Giulianelli, Harry Coppock, Cozmin Ududec, Jasjeet Sekhon, Jacob Steinhardt, Antony Kellermann, Sarah Schwettmann, Matei Zaharia, Ion Stoica, Percy Liang, and Daniel Kang. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint arXiv:2507.02825*, 2025. NeurIPS 2025 Datasets and Benchmarks.

A SAMPLE TASKS

We show two sample tasks that agents cannot solve to provide a better idea of what a task entails. Each example demonstrates a failure mode that coding agents struggle with.

Task `1` (Zombie System) is a multiplayer task in HordeTemplate. It requires coordinated edits across eight source files to implement zombie AI within the game. The agent must consider the existing code on the round handler and player/actor states when implementing the zombie spawn logic as well as the player-zombie interaction logic. A correct implementation must also properly handle replication so that all necessary information is consistent.

Task `19` (Generated Map Orchestrator) is narrower in file count but still integration-heavy. An actor is supposed to populate a procedural grid when the round starts. Once local generation is complete, the actor fires a ready signal. We then should replicate map-size changes, and keep all peers synchronized. Similarly, this task is difficult because the agent must coordinate generation, actor pooling, and timing while handling proper syncing.

B TEST CONSTRUCTION AND FULL-SESSION EXECUTION

This section describes how the benchmark derives tasks, constructs behavioral oracles, executes full Play-in-Editor (PIE) sessions, and controls sources of mutation.

B.1 TASK DERIVATION AND EDITABLE SCOPE

Each task is derived from a feature in a working Unreal Engine 5 project. Task authors identify an existing behavior, preserve its original implementation as the reference solution, and remove the implementation required to produce that behavior while retaining a buildable starting project. The public task specification describes externally observable requirements rather than prescribing a particular implementation.

Each task package defines an editable scope, ordinarily restricted to designated C++ headers and source files. Project levels, Blueprints, materials, animations, configuration, and other assets remain fixed fixtures unless a task explicitly places them in scope. This boundary prevents an agent from satisfying an assertion by rewriting the test environment instead of implementing the requested behavior.

Hidden tests are stored outside the solver workspace and are injected only after the agent completes its implementation. Before evaluation, the runner checks the workspace against the editable-scope policy and verifies that the hidden tests were unavailable during the solve. The agent implementation and injected tests are then compiled together using the project's Unreal build configuration. Compilation is a measured stage because it exposes errors involving C++, headers, reflection macros, generated code, module boundaries, and engine API dependencies.

B.2 REQUIREMENT-TO-ORACLE CONSTRUCTION

Task authors decompose each behavioral specification into observable requirements and construct at least one oracle for each requirement. Each oracle is assigned to the lowest-fidelity verifier capable of observing the required property. This rule prevents a source-level proxy from standing in for behavior that can only be established during engine execution.

Static checks can establish that a required declaration or annotation exists, but they cannot establish that the associated behavior executes correctly. Unit tests evaluate local computations without a complete game world. Integration tests create engine-managed objects in a synthetic `UWorld`, exposing initialization and lifecycle behavior within one context. End-to-end tests execute a complete PIE session and are required when correctness depends on the game loop, authority, remote procedure calls, replication, or state in a separate client world. A task may use several tiers when its specification combines local, lifecycle-dependent, and distributed requirements.

Tier	Execution context	Representative observations	Properties not established
Static	Source and class metadata inspection	Declarations, signatures, macro annotations, and constructor-defined metadata	Executed behavior, lifecycle, and networking
Unit	Direct calls on functions or isolated objects	Pure logic, defaults, conversions, and local state transitions	World lifecycle and distributed state
Integration	Actors and components in a synthetic `UWorld`	Registration, initialization, subsystem interaction, and single-world lifecycle	Cross-client delivery and authority consistency
End-to-end	Complete PIE session	Tick-dependent behavior, possession, authority, RPCs, replication, and client-visible effects	Deployment properties not represented by PIE

Table 4: Verification tiers and the properties observable at each level of execution fidelity.

B.3 Networked PIE Topology

PIE starts a live game session from the project loaded by the Unreal Editor (Epic Games, 2026). For networked evaluation, the harness configures a listen server and the required client worlds and enables execution under one process. The server world is authoritative and also contains the listen server's local player. Each additional client has a distinct `UWorld`, actor set, `APlayerController`, `AGameState`, and local `APlayerState`.

The worlds communicate over loopback through Unreal's built-in networking system, including `UNetDriver`, `UNetConnection`, replicated properties, and remote procedure calls. The single-process topology simplifies observation but does not replace replication: state reaches a client only if the implementation produces it on the authority and Unreal transmits it to that client. Because all worlds are accessible within one process, a test running on the game thread can locate corresponding server and client objects and compare their state without log scraping or a custom inter-process protocol.

The harness uses an invocation of the following form, with task-specific values substituted for the project, test filter, and client count:

```
UnrealEditor <project>.uproject
  -ExecCmds="Automation RunTests <filter>; Quit"
  -Unattended -CrashForUAT -NullRHI -NoSplash
  -NumClients=N -ListenServer
```

The exact command line and PIE settings are recorded with each task configuration. PIE exercises the engine's gameplay lifecycle and networking implementation, but it is not identical to a packaged build, a dedicated-server deployment, or communication over a wide-area network.

B.4 Full-Session Verification Pipeline

The verification pipeline consists of four stages.

Test injection and compilation. After the agent completes its implementation, the benchmark runner copies the hidden tests from the task package into the project's test directory and updates `Build.cs` to include the required `AutomationController` dependency. The modified project and injected tests are compiled together. A compilation failure is recorded as an evaluation outcome rather than bypassed.

Session initialization. The runner launches the project through Unreal's automation framework. For networked tasks, it starts PIE in listen-server mode with the required client count and same-process execution enabled. The test waits until the required worlds, controllers, actors, and subsystems are available instead of assuming that they exist on the first frame.

Stimulus, convergence, and assertion. The test establishes controlled initial conditions and performs the action specified by the task, such as applying damage on the authority, beginning a round, possessing an actor, invoking an RPC, or destroying an object. A latent automation command yields control to the engine between polls so that ticking, lifecycle callbacks, and networking can progress. The command evaluates an explicit readiness predicate until the required state appears or a time budget expires. Once the state has converged, the test evaluates task-specific invariants in the relevant worlds.

In 17 tasks, assertions directly inspect a separate client world. A server-side value therefore does not pass merely because it was computed correctly on the authority; the test also requires the corresponding client state or client-side effect to appear. A schematic replication oracle is:

```
start networked PIE
wait until corresponding server and client actors exist
apply the specified action to the authoritative actor
while client state has not converged and time remains: yield to the next engine tick
assert client state equals the expected state
```

Result collection and cleanup. After execution, the runner parses Unreal's automation log to recover per-test verdicts and subcheck outcomes. The complete log, test results, and final workspace snapshot are retained for auditing. The runner then terminates PIE, restores the relevant settings, and removes the injected tests from the saved solver artifact.

B.5 Controls for Verdict Stability

Unreal Engine is not a fixed-step deterministic simulator. Frame deltas, AI updates, physics, navigation, and loopback delivery may complete on different ticks across repeated executions. We therefore target verdict stability rather than identical execution traces.

Tests use controlled initial conditions, including fixed transforms, seeded fixtures, scripted actions, and explicit gameplay-state transitions where applicable. They avoid fixed sleeps and uncontrolled dependence on elapsed wall-clock time. Instead, latent commands poll state predicates with explicit time budgets while yielding to the engine between ticks. Assertions emphasize final-state invariants, event ordering, monotonic progress, and cross-world agreement rather than exact frame counts. For example, a replication test checks that the expected client state eventually agrees with the server-produced state, not that delivery occurs after a fixed number of frames.

The harness also constrains the execution environment. Automation runs unattended with a standardized command line; relevant PIE settings are snapshotted and restored; and session startup is delayed until required compilation work has completed. Headless tests normally use `-NullRHI` and `-NoSplash`. Command-line changes are restricted by an allowlist, with task-specific exceptions recorded in the task configuration.

`-NullRHI` disables frame rasterization but preserves the game loop, actor lifecycle, gameplay logic, and networking. It is therefore appropriate for behavioral tests that do not require rendered output. Tasks whose specifications require visual rendering must use an explicitly declared rendering-enabled configuration.

B.6 Task Validation and Calibration

Each task is subjected to positive and negative controls before release. The task package must load, the starting project must compile, the reference solution must pass every required test, and the incomplete starting state must fail on the behaviors intentionally removed. A failure caused only by an invalid fixture, environment configuration, or unrelated project defect does not validate the oracle. Test authors inspect subcheck outcomes and execution logs to confirm that failures correspond to the requirements named by the tests.

We additionally evaluate verifier sensitivity using deliberately incorrect implementations, or mutants. Each mutant introduces a localized defect into an otherwise valid solution and is labeled by the affected property class. Running mutants through the verification tiers identifies the first tier that detects each

defect and measures which errors evade shallower verification. This analysis characterizes verifier depth; it does not establish that the test suite captures every possible incorrect implementation.

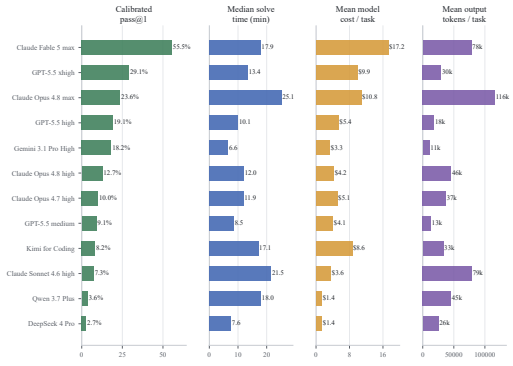
B.7 Recorded Outputs and Scope

For each evaluated run, the runner records compilation status, per-test and per-subcheck outcomes, automation logs, and the final workspace snapshot. A run passes only when compilation succeeds and all checks required by the benchmark's stated scoring rule pass.

Full-Session Verification establishes behavior within the tested PIE configuration. It does not by itself establish rendering correctness under `-NullRHI`, performance at production scale, equivalence to packaged builds or dedicated-server fleets, resilience to packet loss or wide-area latency, platform-specific behavior, or correctness for interaction paths absent from the test suite. These limitations bound the experimental claim: the method evaluates whether an implementation satisfies specified invariants during a controlled and auditable engine session.

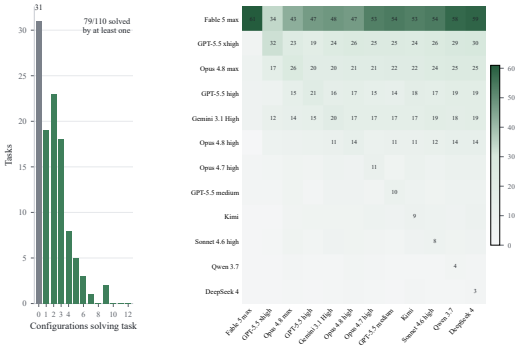
C Supplemental Result Figures

Below are extra figures for analysis.

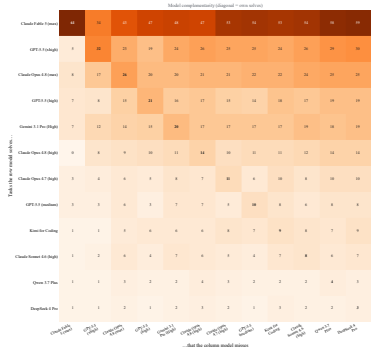


(a) Pass@1 against median time, cost, and output tokens.

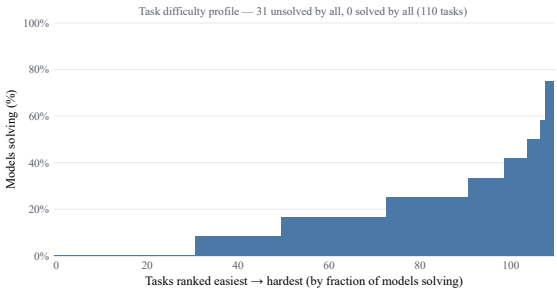
Figure 5: Supplemental verifier and efficiency views.



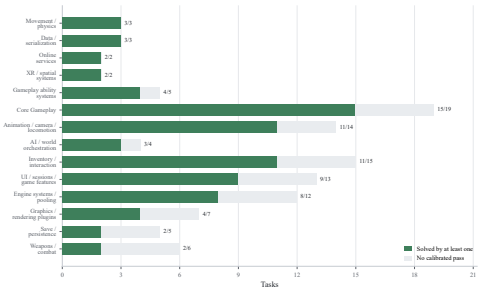
(a) Configurations per task (left); tasks solved by row but missed by column (right).



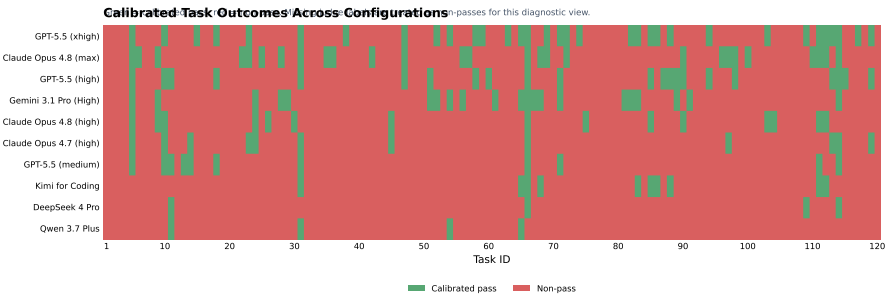
(b) Pairwise task complementarity.



(c) Tasks ranked by fraction of configurations that solve them.

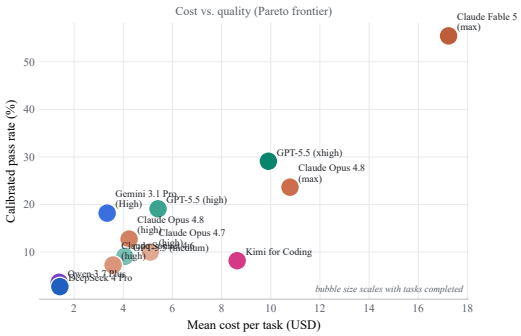


(d) Tasks solved by at least one configuration (green) vs. none (gray), by area.

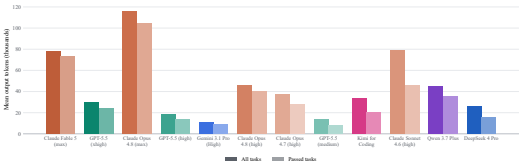


(e) Per-task outcome matrix over the active 110-task set; green is pass.

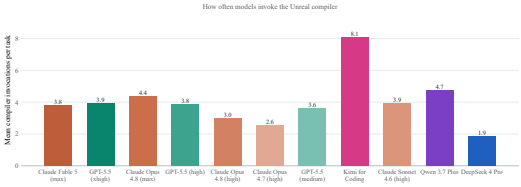
Figure 6: Task coverage and model complementarity. Panel (e) is the task-level view behind the aggregate pass@1 ranking.



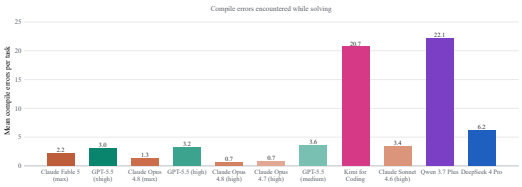
(a) Cost vs. pass rate, where cost data is available.



(b) Output tokens per task.



(c) Mean compiler invocations per task.



(d) Mean compile errors during solving.

Figure 7: Efficiency and resource-use diagnostics. Cost and compile accounting are only as complete as each wrapper's trajectory exposes, so (b)–(e) are available-data diagnostics rather than a pricing comparison.

D COMPLEMENTARITY EXAMPLES

Figure 6 show that model capability is not completely nested. This is important since it suggests that lower-ranked models are still useful in terms of additional capability even if higher ranked ones are available. The task-level matrix supports this view: some tasks are completely missed by overall stronger models and yet end up being completed by purportedly weaker models. Such complementarity also suggests that the benchmark is testing a variety of reasoning axes beyond a straightforward notion of C++ generation skill.

It is useful to present two tasks for this comparison, both of which were solved only by one configuration: Task `15` and Task `19`. Task `15` shows a great amount of dependence on initialization ordering, which only `gpt-5.5` at `xhigh` is able to clearly fulfill. Task `19`, on the other hand, needs multiple subsystems being handled correctly: replication, object lifecycle, pooling and async generation, each interacting with the rest. Only the `claude-fable-5` at `max` is able to juggle all this together.

Both these tasks, solved only for one run, justify the off-diagonal entries in the complementarity matrix, since they fail in completely different ways. This presents pockets of capability that the models hold that would be difficult or more homogeneous to achieve.

E LICENSING AND AVAILABILITY

GameEngineBench is MIT licensed. However, the nine source projects carry a mix of licenses, affecting the distribution of the tasks. While some offer permissive licenses, others have licenses that restrict redistribution. More detail on the license of each project is listed in the manifest. The benchmark and harness will be made available.