

CraftBench-UE: Deterministic Evaluation for Coding Agents in Unreal Engine

Shutong Wu*
University of Rochester

Kevin Calderone, Andy Tsen†
RamenVR

✉ swu85@ur.rochester.edu

🔗 <https://github.com/ramenvr/craftbench-ue>

ABSTRACT

Building gameplay features in a game engine requires more than code, as code that compiles and runs does not necessarily implement the requested gameplay. We introduce CraftBench-UE, an evaluation harness that runs agents in an isolated Unreal Engine environment, reconstructs their saved submissions in fresh projects, and applies deterministic build, asset, and runtime checks without an LLM judge. Based on the harness, we build a benchmark consisting of 70 tasks spanning C++ source, Blueprint assets, and editor scripting. We evaluate seven models under two editor-tool configurations, with a file-and-shell baseline on C++ tasks. We further pair tasks that specify the same gameplay and use the same runtime tests, but require C++ and Blueprint as the deliverables. Across the 10 paired tasks, C++ completion rates exceed Blueprint by 30.0 and 42.9 percentage points in the two tool configurations. Among on-time Blueprint submissions in this paired set that pass asset checks, 42.2% and 50.0% fail explicit runtime assertions. These submissions satisfy asset requirements but fail the required gameplay tests. We will release the harness, task benchmark, and our trajectory findings with the report.

1 INTRODUCTION

Implementing a gameplay feature involves more than producing code that compiles. In Unreal Engine, a feature can span C++ classes, Blueprint graphs, and other features such as the Niagara VFX system. Blueprint is Unreal’s visual programming system, which uses node graphs saved in binary .uasset files. The evaluation question is therefore whether the saved project implements the requested behavior, not simply whether it compiles.

Existing benchmarks evaluate generated functions, repository changes, and tasks in interactive environments (Chen et al., 2021; Jimenez et al., 2024; Merrill et al., 2026; Zhou et al., 2024). Game-development benchmarks extend executable evaluation to game engines such as Godot, Unreal, and Roblox (Chi et al., 2026; La et al., 2026; Zhang et al., 2025). Evaluating projects that combine source code and native assets requires checking both the submitted files and the behavior they produce. The benchmark harness should also differentiate between structural requirements, like a valid asset reference or successful compilation, and behavioral requirements, such as a character decelerating while gliding.

We introduce CraftBench-UE, an evaluation harness for coding agents working with Unreal projects. Each task defines the starting state and the goal deliverables. The harness collects the generated work, then reconstructs it in a fresh project and grades it based on the build, asset, and runtime

*Work done during internship at RamenVR.

†Corresponding author.

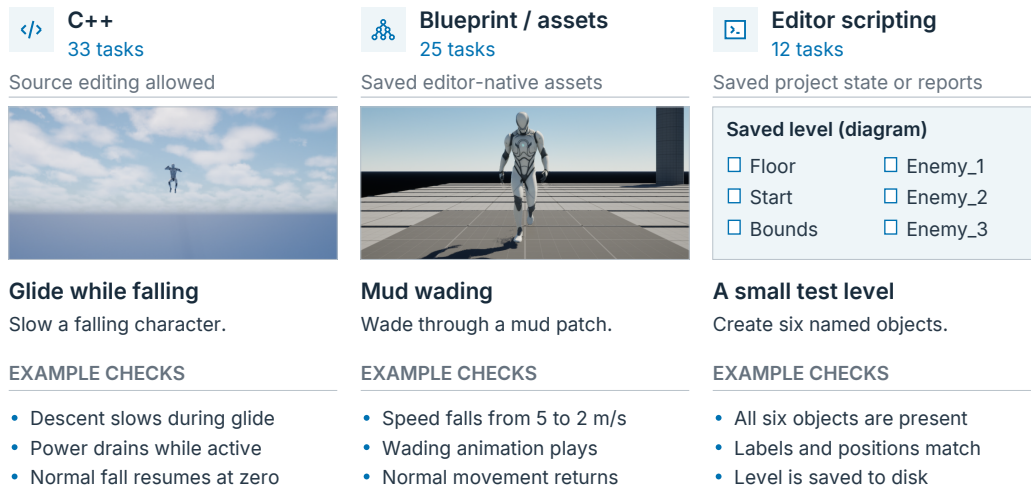


Figure 1: Sample benchmark tasks across the three deliverable groups. The screenshots illustrate the glide task (gp-glide-stamina-cpp) and mud-wading task (t1-mud-wade-bp). The editor-scripting panel shows the kp-spawn-level-actors task, which requires creating and saving a level containing six named objects.

checks. Runtime checks observe behavior in Play-in-Editor (PIE), Unreal’s mode for running a game inside the editor.

Based on the harness, we construct a benchmark of 70 bounded tasks: 33 C++ tasks, 25 Blueprint/native-asset tasks, and 12 editor-scripting tasks. The task suite contains 10 gameplay task pairs, each with both C++ and Blueprint versions. This design enables analysis of how the same agent approaches the same tasks across different deliverable formats.

We evaluate seven models under two editor-tool configurations, MCP1 and MCP2 (§4), with a file-and-shell C++ baseline. In the paired analysis, we found that C++ completion exceeds Blueprint completion by 30.0 and 42.9 percentage points under MCP1 and MCP2 within the time budget. Among on-time Blueprint submissions in this set that satisfy asset checks, 42.2% (19/45) under MCP1 and 50.0% (20/40) under MCP2 fail explicit runtime assertions. The comparison and result analyses address complementary questions: whether agents complete the same gameplay under different deliverable requirements, and what passing an asset check tells us about the final completed feature.

Our contributions are:

1. **An Unreal evaluation harness.** CraftBench-UE runs an agent on a task in an isolated Unreal project, reconstructs its submission and grades it through build, asset, and runtime verifiers.
2. **A benchmark of 70 tasks.** The tasks cover source code, native assets, and editor scripting, including paired gameplay specifications with the same runtime tests.
3. **An empirical study of how coding agents work in game development tasks.** We share our results on how different models work through the benchmark, and how they perform and verify their work during the benchmarking process. Paired comparisons quantify differences under two editor configurations; submission and trajectory analyses show where asset construction and authoring feedback fall short of completed gameplay.

2 RELATED WORK

Execution-based agent evaluation. Tests can reject a valid solution if they impose requirements that the task description does not state. SWE-bench Verified addresses this problem through developer review, screening out unclear tasks and tests that reject valid solutions (OpenAI, 2024). The

checks should match what the agent was asked to do, whether the task involves writing a function, modifying a repository, or working in a terminal or desktop environment (Chen et al., 2021; Jimenez et al., 2024; Merrill et al., 2026; Xie et al., 2024). We document our validation process in §3.

Game-development benchmarks. GameDevBench uses deterministic tests for Godot scripts, scenes, and resources and examines how image and video feedback affect agent performance (Chi et al., 2026). In GameEngineBench, agents make C++ changes in open source Unreal projects, and the changes are examined by both build and runtime tests; an LLM judge then considers the test results and implementation work (La et al., 2026). OpenGameEval evaluates Roblox Studio tasks with executable tests (Zhang et al., 2025). GameXpert-Bench spans generation, repair, and multi-turn optimization of browser-native JavaScript games (Chen et al., 2026). These benchmarks already evaluate the behavior of changes made by agents. GBQA instead scores autonomous defect discovery in running web games by the share of human-verified bugs found (Jiang et al., 2026).

Other work in Unreal covers cinematic assets and complete games. CutsceneBench combines tool and asset checks for Level Sequence assets with evaluation of rendered videos (He et al., 2026). AutoUE focuses on end-to-end Unreal game generation using an LLM judge as the grader (Yin et al., 2026). Our tasks focus on individual gameplay features with outcomes that could be deterministically checked. The harness reconstructs each submission in a fresh Unreal project and applies build, asset, and runtime checks. Appendix G provides a detailed comparison of these benchmarks.

Interfaces and tool use. MCP exposes application operations as tools (Anthropic, 2024), and MCP benchmarks study tool selection and composition (Luo et al., 2025; Wu et al., 2025). Orak likewise standardizes agent access to games through MCP, but its deliverable is a gameplay trajectory scored by game outcomes rather than a saved engine artifact (Park et al., 2026). MCP-Unity connects MCP clients through a Python server to a C# Unity Editor plugin (Wu & Barnett, 2025); Epic’s in-editor Unreal MCP server exposes tools over local HTTP and dispatches calls on the game thread (Epic Games, n.d.). These interfaces provide authoring tool sets; our harness evaluates saved submissions in fresh Unreal projects.

Agent-interface studies show that the surrounding configuration can affect coding outcomes (Yang et al., 2024; Zhang et al., 2026). SkillsBench compares matched conditions with and without curated procedural Skills using deterministic verifiers (Li et al., 2026), whereas we vary editor configurations and required deliverables. We therefore describe editor services, editing permissions, and execution settings alongside the model panel. The observed differences compare these configurations; they do not identify the effect of an individual tool operation.

3 CRAFTBENCH-UE

3.1 EVALUATION SETTING AND TASK SCOPE

CraftBench-UE measures whether an agent can implement a bounded gameplay or editor behavior in an existing Unreal project. The unit of evaluation is an *atomic mechanism task*: an instruction specifies a bounded gameplay or editor behavior, and states which project files or assets the agent may change. Examples include attaching an object to a skeletal socket, enforcing a cooldown, persisting a Blueprint graph, and authoring a mini level (Figure 1). Each task focuses on a small set of engine mechanisms and uses named checks to identify which tested conditions fail. Tasks may still couple subsystems, as a movement behavior can involve input, state, timing, and physics, but they do not combine unrelated features merely to increase difficulty. Each score records whether the submitted project implements that mechanism under the stated task conditions.

3.2 CRAFTBENCH-UE EVALUATION HARNESS

Container-oriented frameworks such as Harbor package instructions, environments, and tests (Harbor Framework Team, 2026). Unreal evaluation is comparatively more complex since the framework needs to control a versioned Editor and PIE environment, while preserving binary assets and saved changes. A task package specifies the starting project, prompt, writable area, engine build, verifier, and reference implementation. The harness reconstructs the project and delivers the prompt through

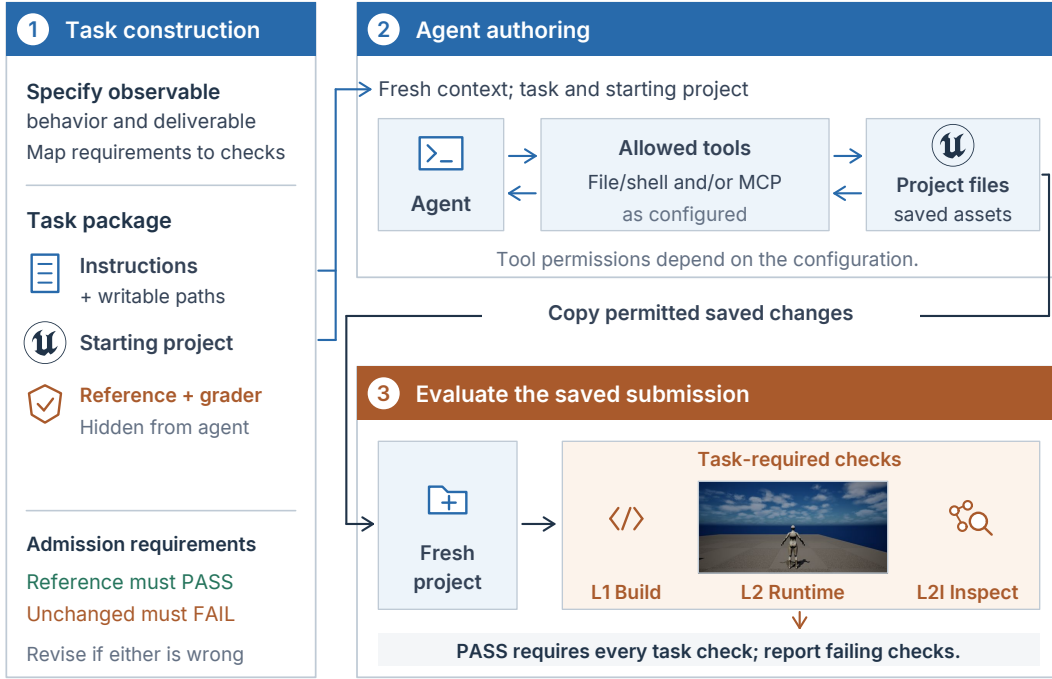


Figure 2: Task construction, agent authoring, and evaluation of the saved submission in a fresh project. Each task declares its required checks. Reference-pass and unchanged-project-fail are admission criteria; §3.5 describes task authoring and validation.

the selected agent configuration. Reference implementations and verifier sources are withheld during authoring; detected exposure excludes a run from scoring (Appendix B.1).

The harness records interactions, timing, and project changes. When authoring ends, it collects the saved changes allowed by the task and applies them to a fresh copy of the starting project. This clean replay separates the submitted artifact from unrelated workspace modifications. It then builds the project, runs behavioral checks, and inspects native assets as declared by the task. Reports include the overall verdict, gate outcomes, and the failing checks identified. Figure 2 summarizes the workflow.

3.3 EXECUTION-BASED DETERMINISTIC GRADING

Each task specifies which of the following checks its submission must pass:

1. **Build (L1).** Compile the relevant editor and game targets. Submission-caused compiler or linker errors are failures.
2. **Behavior (L2).** A fixed-timestep Play-in-Editor (PIE) session will start and evaluate named assertions at certain timesteps using Unreal Functional Tests.¹ Assertions focus on state transitions and variable changes during the session.
3. **Artifact (L2I).** Load the saved assets in the editor without its graphical interface and check the required graph topology, class hierarchy, typed properties, references, or persistence.

For each task, we specify the checks a submission must pass and evaluate them directly in Unreal Engine. The same assertions and checkpoint schedule are applied to every submission for a task. A fixed simulation timestep (1/60 second by default) makes time-dependent checks comparable, while per-task numerical tolerances accommodate the quantities being tested; Appendix B lists the engine switches and the tolerances in force. No LLM reads the submission, trace, screenshot, or test report

¹Unreal’s Functional Testing framework runs scripted checks inside a level; tests can be written in C++ or Blueprint. See Epic’s Functional Testing documentation.

to decide whether it passes or fails. Screenshots may be retained for diagnosis, but never to change the final grade. Structural checks establish required asset properties, while runtime checks establish the tested behavior. This excludes creative tasks whose success depends on visual judgment, a limitation we discuss in Sec. 8.

3.4 THE BENCHMARK AND REQUIRED DELIVERABLES

The task suite contains 33 C++ tasks, 25 Blueprint tasks, and 12 editor-scripting tasks. C++ tasks permit source code changes in a designated folder; BP tasks require saved native assets; scripting tasks are evaluated on their resulting saved project state. The tasks cover domains including gameplay logic, character movement, animation, UI, and editor operations. Of the 70 tasks, 50 start from the Unreal Engine 5.8 Third Person template and 20 from a customized template. Table 1 gives their required checks; Appendix F lists the individual tasks.

Figure 1 shows the three task types in our benchmark. Tasks usually contain multiple checks: glide, for example, must slow a falling character, consume Power, and restore to normal falling state when Power is drained. Editor-scripting tasks instead check the saved output, such as whether a level contains the requested objects. The harness uses each task’s declared checks since not all check types are needed for each task.

Task group	Tasks	Build	Runtime	Assets
C++	33	33	33	0
BP	25	25	15	20
Python	12	12	0	12

Table 1: Required checks in the measured suite. Runtime and asset columns overlap; each paired BP gameplay task requires both.

Ten gameplay specifications have C++ and BP versions that share a starting project template and runtime verifiers. They contain the same interface and gameplay instruction; the only difference is the requested deliverable type (BP and C++). BP additionally requires saved assets. We inspect the submitted files when interpreting the comparison. §4 defines the paired analysis.

3.5 TASK AUTHORING AND VALIDATION

The tasks are designed around common Unreal workflows and informed by the team’s engine expertise and public engine documentation. Team members, including artists, designers, and engineers, propose behaviors that represent the work they use agents to perform. The lead author filters proposals using three admission criteria: the requested outcome must be (i) specific enough to verify with explicit checks, (ii) observable in a clean project without inspecting a solution, and (iii) implementable through a declared deliverable. When a gameplay behavior is suitable for both C++ and Blueprint, it can become a matched pair with different artifact requirements.

The team refines each accepted proposal into a task package that includes instructions, starting project, grading criteria, and other necessary information for agent to start the implementation. We verify that there are at least one reference solution that passes all required checks and that the unmodified starting project fails. Appendix A follows one task from its instruction to the final grading pass.

4 EXPERIMENTAL DESIGN

4.1 MODELS AND TOOL CONFIGURATIONS

We evaluate Claude Sonnet 5, DeepSeek V4 Pro-0813, Gemini 3.7 Flash, GPT-5.6 Luna, GPT-5.6 Sol, Grok 4.6, and GLM-5.3 Flash. The two editor-tool configurations use Unreal MCP (MCP1) and Aura MCP (MCP2). Each model attempts every task under both configurations; C++ also has a file/shell baseline without editor MCP. All configurations use Claude Code with Unreal Engine 5.8 on Windows. Appendix B records model details.

MCP1 exposes catalog discovery and a gateway for editor operations. MCP2 exposes operations directly, including delegated Blueprint and Python tools. The two MCPs provide different editor tools and file-editing permissions, as summarized in Table 2. On BP and Python, their completion differences are therefore not attributable to editor tools alone (Appendix B).

Configuration	C++ tasks	BP and Python tasks
File/shell baseline	Read, write, edit, shell; no editor MCP	Not evaluated
MCP1	File/shell tools and editor MCP	File/shell tools and editor MCP
MCP2	File/shell tools and editor MCP	Read/search and editor MCP; no native write/edit/shell

Table 2: Agent tool permissions. Both editor configurations can perform several kinds of project edits; tool availability is part of the evaluated configuration.

4.2 TASK RUN AND COMPLETION

Each task–model–MCP combination is evaluated with one scored attempt. The agent starts in a fresh project with the task prompt and is prompted to conduct the task. The prompt instructs agents to perform at most one final PIE playtest, which the harness does not enforce. We added this instruction after observing repeated playtest loops in preliminary runs. Agents receive no human feedback during authoring.

We report pass@1 with a 40-minute authoring budget. An attempt is successful if the agent finishes within this budget and its final submission passes all checks. Authoring time is measured from an agent launch to its exit, and the project setup and grading time are not included. Sessions normally have a 60-minute execution ceiling; final submissions returned after the 40-minute budget but under the ceiling are retained for analysis.

4.3 PAIRED COMPARISONS AND OBSERVED WORKFLOWS

For each model, we compare its completion count under the two MCP settings and, on C++, also against the file/shell baseline. We also identify tasks that succeed in one setting and fail in the other, since equal totals can contain successes in different tasks. The ten paired gameplay specifications compare C++ and BP implementations of the same requested behavior. We analyze the interaction logs alongside the final test reports to examine how agents conduct their work. We also measure authoring time and the efficiency of tool calls.

We compare completion using only the gameplay tests to see whether Blueprint’s extra asset checks account for its lower score. Appendix C provides the analysis details.

5 RESULTS

We report task completion for each model and configuration, then examine the saved submissions and tool requests to understand incomplete gameplay and the work performed during authoring. Completion uses the 40-minute budget; trajectory and runtime diagnostics examine the full recorded sessions and final submissions. Appendix E reports authoring time and tool calls.

Appendix Figure 5 shows the full task result. C++ completion is 180/231 (77.9%) with file/shell tools, 199/231 (86.1%) with MCP1, and 203/231 (87.9%) with MCP2. Both MCP configurations complete 110/175 BP runs (62.9%). Python completion is 62/84 (73.8%) and 66/84 (78.6%).

5.1 C++ AND BLUEPRINT IMPLEMENTATIONS OF THE SAME GAMEPLAY

The ten paired gameplay tasks compare the same requested behavior across C++ and BP (Table 3). MCP1 completes 46/70 C++ runs (65.7%) and 25/70 BP runs (35.7%); MCP2 completes 50/70 (71.4%) and 20/70 (28.6%), respectively. Every model completes more C++ versions under both configurations (Figure 3; Table 7). The C++ file/shell baseline already completes 45/70 runs (64.3%), so the higher C++ completion also occurs without editor MCP on the C++ side; most of

Configuration	C++ completion	BP completion
File/shell baseline	45/70 (64.3%)	—
MCP1	46/70 (65.7%)	25/70 (35.7%)
MCP2	50/70 (71.4%)	20/70 (28.6%)

Table 3: Completion within 40 minutes on the same ten gameplay tasks. Seven models run each task, giving 70 runs per evaluated configuration and variant.

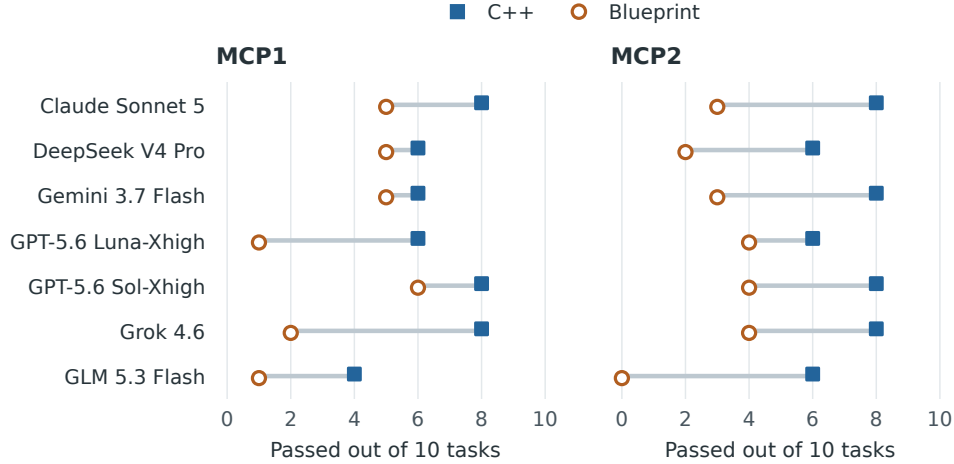


Figure 3: C++ and BP completions within 40 minutes on the ten paired gameplay tasks. Each line connects the same model’s totals under one MCP configuration, with one attempt per task variant.

each MCP configuration’s advantage over that baseline is on C++ tasks without BP counterparts (Table 8).

The C++ completion advantage is 30.0 percentage points under MCP1 and 42.9 under MCP2. Counting submissions that pass the runtime checks without requiring the additional BP asset checks leaves all four completion counts unchanged (Table 9). Within the deadline, no BP submissions pass the runtime checks but fail the asset checks. The BP deficit persists when completion is assessed by gameplay testing. Producing saved native assets remains part of the BP authoring task.

The submissions show how the successful implementations were authored. Although some C++ prompts also permit Blueprint assets, every successful matched C++ submission under either MCP configuration changes C++ source files without changing Blueprint assets. Successful BP submissions change assets without changing the source.

The BP breakdown separates local edits from complete gameplay features (Appendix Figure 6). The three basic graph tasks ask for logging, arithmetic, or delayed movement; two local repair tasks target specified defects in existing behaviors. Both configurations complete 34/35 runs (97.1%) on these five tasks. On ten asset-configuration tasks, completion is 51/70 (72.9%) and 56/70 (80.0%); on the ten gameplay tasks, it falls to 25/70 (35.7%) and 20/70 (28.6%). Agents often complete the included local operations, while fewer submissions implement all parts of a gameplay feature correctly.

5.2 WHERE BLUEPRINT IMPLEMENTATIONS FALL SHORT

On the ten paired gameplay tasks, seven additional MCP1 BP runs and ten additional MCP2 runs yield passing submissions after the 40-minute deadline. Across all final submissions, those that pass asset checks but fail an explicit runtime assertion account for 24/70 (34.3%) under MCP1 and 28/70 (40.0%) under MCP2. Restricted to submissions whose authoring finished within 40 minutes, they

are 19/45 and 20/40 of the on-time submissions that pass asset checks. Appendix D.3 lists the failed checks and the available reports.

The runtime reports provide further observations about these incomplete features. Among submissions that pass asset checks but fail runtime checks, 13 submissions across four BP tasks successfully activate an ability but fail to produce its required effect or meet the required resource check. Seven of them concern the gliding task: the character must descend more slowly while consuming Power. All seven abilities activate, and six consume Power, but none produce the required glide. For example, in three implementations, Power always activates at the start but does not exhaust after gliding for a set time. The other cases comprise four missing periodic burns, one second jump without its Power cost, and one damage operation that leaves Health unchanged. Nine of these runs finish authoring within the scoring budget. Appendix D.3 identifies the 13 submissions and the runtime observations supporting this breakdown.

5.3 HOW AGENTS INSPECT AND VERIFY THEIR WORK

We examine how agents respond to editing feedback and check their work under the instruction to perform at most one PIE playtest. *Asset readback* means retrieving an asset’s current properties or graph from the editor. *PIE state readback* means retrieving values from a running game, such as character position or remaining Power. Both are actions during authoring; the harness applies the task’s runtime checks to the submitted project afterward.

Agents revise both rejected and accepted edits. Of the 175 MCP1 BP runs, 95 directly write Blueprint graphs; 75 read task assets before their first graph write. Among the 58 runs with graph-writing errors, 37 responded in their next tool call by changing the request for the same graph. Agents also revised graphs after requests that returned without an error: this occurred in 30 of the 94 runs with such responses. Before making these later changes, agents often inspected the graph or requested compilation. Appendix D.1 gives the detailed counts and sequence definitions.

Asset readbacks are more frequent than PIE state readbacks. Most BP runs contain asset readback or compilation requests, while PIE state readback appears in 24/175 MCP1 and 30/175 MCP2 runs (Table 4). Agents obtain these values through property queries, scripts that combine actions and measurements, and game-state recordings. C++ agents also use shell commands to request game or engine-test execution, in 57/231 MCP1 and 39/231 MCP2 runs, including unsuccessful attempts. In ten Python MCP2 runs across six tasks and five models, agents find the relevant API or tool, run an editor script that reports success, and read back task-asset state. The analysis follows these different routes through the request arguments and responses (Appendix D.1).

Observed method in BP runs	MCP1	MCP2
Task-asset readback	149/175	165/175
Compilation request	131/175	132/175
PIE start request	38/175	44/175
Image-capture request	15/175	10/175
PIE state readback	24/175	30/175

Table 4: Checking actions in BP trajectories, counted over full authoring sessions. Rows overlap. Request counts include unsuccessful attempts; readback counts require returned values. PIE starts include tool calls and starts within scripts. Image requests can capture either the editor or gameplay. Appendix D.1 defines the categories and their coverage.

Success revisions do not guarantee correct gameplay. In 39 MCP1 Blueprint runs that finished within 40 minutes, every graph-writing error was followed by a changed request for the same graph that returned without an error. Nevertheless, 16 submissions failed the final evaluation, including 15 that failed runtime assertions across six models and eight tasks. A separate inspection of final submissions found five MCP1 Blueprint submissions with resource references that the engine could not resolve; these submissions also failed runtime checks (Appendix D.2).

On the ten paired BP tasks, five MCP1 and ten MCP2 runs contain PIE state readback and pass the asset checks, yet fail the final runtime checks. Reading a value during PIE can leave other required behaviors untested.

6 DISCUSSION

Design tasks with interacting gameplay requirements. A glide feature must slow descent, consume Power, and restore normal falling when Power is exhausted. These requirements need to hold together: checking resource consumption alone would not detect an implementation that leaves descent unchanged. The runtime reports and graph-revision results show why individual editing operations are not sufficient evidence of completion. For bounded gameplay tasks, evaluation should cover each required behavior and the transitions between them within one gameplay mechanism.

Implications for editor-native assets. Our findings motivate evaluating native assets in the engine systems that use them. For animation assets, materials, and scene configuration, the question is not only whether their saved properties are correct, but whether they produce the requested motion, appearance, or interaction. CraftBench-UE examines this connection in Unreal gameplay. Whether the same failure patterns occur in other engines and asset workflows remains a question for future work.

Verification inside the game engine. The observed workflows suggest two complementary forms of feedback. Inspecting asset properties and compiling graphs helps agents check what they have built; running the game and measuring its state helps them check what it does. Asset readbacks are more frequent than PIE state readbacks in the retained trajectories, measured under the prompt’s instruction to perform at most one PIE playtest (§4). The runtime reports show several errors that can be identified using this method, such as unresolved references and missing or incorrect resource usage. Because game development offers many verification methods, such as screenshots, console logs, and PIE tests, how agents choose or combine these methods is worth studying with a larger sample size, especially when many methods are hints or stimuli rather than direct test feedback. For example, console logs in major game engines can be helpful but may also include unrelated information that bloats the agent context. For agent design, this suggests connecting each requested expected behavior to a set of verification methods and expected results, with a verification flow that connects these individual parts and automatically play-tests the implementations.

7 CONCLUSION

CraftBench-UE evaluates coding agents through their saved source and assets and the behavior those deliverables produce in Unreal Engine. Its harness reconstructs submissions in fresh projects and applies task-specific build, asset, and runtime checks. The 70-task benchmark supports comparisons across three deliverable groups. On 10 matched gameplay specifications, C++ completion exceeds BP completion under both editor configurations. Submissions that pass asset checks can still fail runtime assertions, demonstrating the value of evaluating structural requirements and gameplay behavior together.

8 LIMITATIONS

Task and check scope. The 70 tasks are selected for explicit, checkable outcomes rather than sampled representatively from game development. They do not measure visual appeal or the integration of many features into a complete game. Future work could extend the task set to long-horizon projects and creative requirements with separately validated evaluation criteria. Meanwhile, our current checks include three necessary types of tests in Unreal Engine, while future work could explore custom tests to verify generated features, potentially providing more coverage than this harness.

Experimental scope. Each task–model–configuration run has one attempt, with at most one PIE playtest during authoring. Results do not estimate repeated-attempt variability, long-horizon task performance, or performance with unrestricted testing. Future work could explore more about how agents verify themselves to produce the expected outcome.

Environment coverage. All experiments use Unreal Engine 5.8 on Windows through Claude Code. We have not tested whether the observed completion patterns persist across other engine versions, operating systems, or agent runtimes.

ACKNOWLEDGMENTS

We thank the entire Aura team for their support and contributions to the benchmark tasks.

ETHICS STATEMENT

The authors’ organization develops MCP2, whose implementation remains proprietary. Appendix B describes its exposed tool surface. The benchmark team authors the tasks and verifiers. Both editor configurations are evaluated on the same task suite using the same verifiers, and their results are reported separately.

REPRODUCIBILITY STATEMENT

The harness, benchmark materials, and analysis code will be released upon publication at <https://github.com/ramenvr/craftbench-ue>. Task instructions, starting-project configurations, verifier, and reference implementations are included as well. Appendix D defines the trajectory analyses. Appendix B.2 lists the release contents. Users will need to install Unreal Engine and provide it in the environment; engine binaries and proprietary MCP internals are not redistributed.

AI USE STATEMENT

Generative-AI assistants supported task ideation, methodological critique, data analysis, and figure brainstorming. The rule-based pipeline in §3 produced benchmark verdicts. The authors remain responsible for the analysis, claims, and final manuscript.

REFERENCES

- Anthropic. Introducing the Model Context Protocol. industry blog, 2024. URL <https://www.anthropic.com/news/model-context-protocol>.
- Kun Chen, Haorong Hong, Peizhong Gao, Jianfeng Lin, Tongxu Luo, Yuxuan Xie, Chenxu Liu, Jiuling He, Zhongyuan Liu, and Zeno Zeng. GameXpert-Bench: How Far Are Coding Agents from Expert Game Development?, 2026. URL <https://arxiv.org/abs/2608.21833>.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, et al. Evaluating Large Language Models Trained on Code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qihong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating Agentic Capabilities Through Game Development. In *ICML 2026*, 2026. URL <https://arxiv.org/abs/2602.11103>.
- Epic Games. Unreal MCP in Unreal Editor. Unreal Engine 5.8 official documentation, n.d. URL <https://dev.epicgames.com/documentation/unreal-engine/unreal-mcp-in-unreal-editor>. Accessed September 14, 2026.
- Harbor Framework Team. Harbor: A Framework for Evaluating and Optimizing Agents and Models in Container Environments. Software, 2026. URL <https://github.com/harbor-framework/harbor>.
- Lanshan He, Haozhou Pang, Qi Gan, Xin Shen, Ziwei Zhang, Yibo Liu, Gang Fang, Bo Liu, Kai Sheng, Shengfeng Zeng, Chaofan Li, Zhen Hui, Keer Zhou, Lan Zhou, and Shujun Dai. Cutscene Agent: An LLM Agent Framework for Automated 3D Cutscene Generation, 2026. URL <https://arxiv.org/abs/2604.25318>.
- Shufan Jiang, Chios Chen, and Zhiyang Chen. GBQA: A Game Benchmark for Evaluating LLMs as Quality Assurance Engineers. ICLR 2026 workshop paper, 2026. URL <https://arxiv.org/abs/2604.02648>.

- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *ICLR 2024 (oral)*, 2024. URL <https://arxiv.org/abs/2310.06770>.
- Brian La, Sejoon Chang, Ben Kim, Junyoung Bae, Aamish Ahmad Beg, Sei Chang, Gonzalo Gonzalez-Pumariaga, and Kanav Goyal. GameEngineBench: Evaluating Coding Agents on Real C++ Runtime Environments, 2026. URL <https://arxiv.org/abs/2607.03525>.
- Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, et al. SkillsBench: Benchmarking How Well Agent Skills Work Across Diverse Tasks, 2026. URL <https://arxiv.org/abs/2602.12670>.
- Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. MCP-Universe: Benchmarking Large Language Models with Real-World Model Context Protocol Servers, 2025. URL <https://arxiv.org/abs/2508.14704>.
- Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, et al. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. In *ICLR 2026 (poster)*, 2026. URL <https://arxiv.org/abs/2601.11868>.
- OpenAI. Introducing SWE-bench Verified. industry blog, 2024. URL <https://openai.com/index/introducing-swe-bench-verified/>.
- Dongmin Park, Minkyu Kim, Beongjun Choi, Junhyuck Kim, Keon Lee, Jonghyun Lee, Inkyu Park, Byeong-Uk Lee, Jaeyoung Hwang, Jaewoo Ahn, et al. Orak: A Foundational Benchmark for Training and Evaluating LLM Agents on Diverse Video Games. In *ICLR 2026 poster*, 2026. URL <https://arxiv.org/abs/2506.03610>.
- Shutong Wu and Justin P. Barnett. MCP-Unity: Protocol-Driven Framework for Interactive 3D Authoring. In *SIGGRAPH Asia 2025 Technical Communications*. Association for Computing Machinery, 2025. doi: 10.1145/3757376.3771417. URL <https://doi.org/10.1145/3757376.3771417>.
- Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, et al. MCPMark: A Benchmark for Stress-Testing Realistic and Comprehensive MCP Use, 2025. URL <https://arxiv.org/abs/2509.24002>.
- Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, et al. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *NeurIPS 2024 (Datasets and Benchmarks Track)*, 2024. URL <https://arxiv.org/abs/2404.07972>.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *NeurIPS 2024 (poster)*, 2024. URL <https://arxiv.org/abs/2405.15793>.
- 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. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2341–2364, 2026. URL <https://aclanthology.org/2026.findings-acl.111/>.
- Tiantian Zhang, Kartik Ayyar, Mengsha Sun, and Lynn Gong. Using OpenGameEval to Benchmark Agentic AI Assistants for Roblox Studio. Roblox newsroom, 2025. URL <https://about.roblox.com/newsroom/2025/12/opengameeval-benchmark-agentic-ai-assistants-roblox-studio>.
- Yunbei Zhang, Janet Wang, Yingqiang Ge, Weijie Xu, Jihun Hamm, and Chandan K. Reddy. Stop Comparing LLM Agents Without Disclosing the Harness, 2026. URL <https://arxiv.org/abs/2605.23950>.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *ICLR 2024*, 2024. URL <https://arxiv.org/abs/2307.13854>.

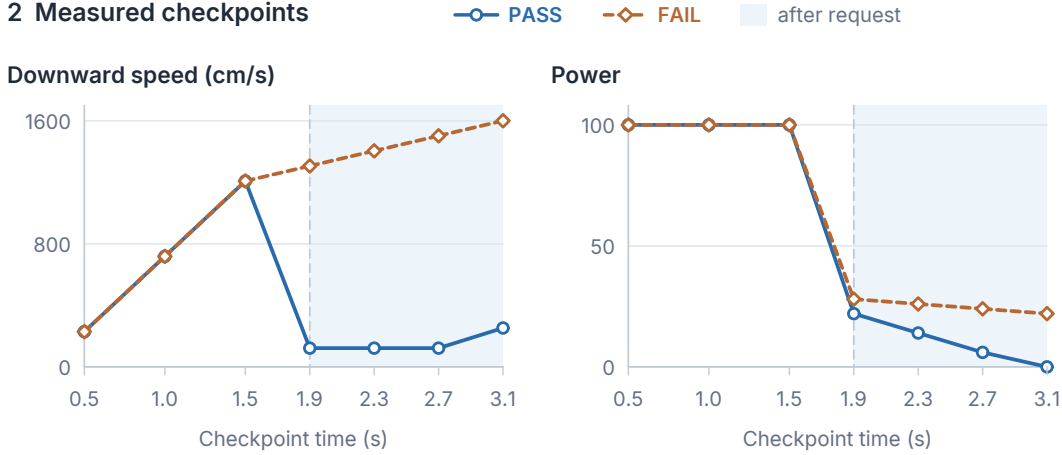
A A WORKED EXAMPLE OF TASK VALIDATION

The glide task in Figure 4 asks the agent to slow a falling character while consuming Power, then restore normal falling when Power runs out. The runtime fixture requests ability activation and observes descent and Power at scheduled checkpoints. Checking Power alone would miss an implementation that consumes the resource without slowing the character.

1 Task requirements

Slow descent while glide is active	Consume Power during the glide	Resume falling when Power reaches zero
--	--	--

2 Measured checkpoints



3 Task view and grading outcomes

PASS: all required behaviors

Descent slows, Power drains, falling resumes.

At 2.3 s: speed 122 cm/s; Power 14.

FAIL: descent never slows

Power drains, but the glide has no effect.

At 2.3 s: speed 1405 cm/s; Power 26.

Figure 4: Archived development/validation traces for the glide task, separate from the study’s agent runs. In the shaded post-request interval, the passing trace slows descent while Power drains; the failing trace drains Power without slowing descent. The plotted values and verdict labels are transcribed from a retained development capture. The Manny screenshot is a separate scene illustration, not a frame from either trace.

This example illustrates why task authors check behavior after integration into the engine. Figure 1 in the main paper illustrates tasks across the three deliverable groups.

B EXECUTION CONFIGURATION AND SUBMISSION HANDLING

The experiments use Unreal Engine 5.8 on Windows with Claude Code as the agent driver. Table 5 records the requested outer-agent model identifiers. Provider-qualified identifiers are served via the same Claude Code CLI, with its Anthropic base URL pointing to OpenRouter’s Anthropic-compatible endpoint and the OpenRouter key supplied as the CLI’s bearer token; `claude-sonnet-5` is requested directly from Anthropic. The analysis manifest retains available model and driver metadata.

Display label	Requested model identifier
Claude Sonnet 5	<code>claude-sonnet-5</code>
DeepSeek V4 Pro	<code>deepseek/deepseek-v4-pro-0813</code>
Gemini 3.7 Flash	<code>google/gemini-3.7-flash</code>
GPT-5.6 Luna	<code>openai/gpt-5.6-luna</code>
GPT-5.6 Sol	<code>openai/gpt-5.6-sol</code>
Grok 4.6	<code>x-ai/grok-4.6</code>
GLM-5.3 Flash	<code>z-ai/glm-5.3-flash</code>

Table 5: Requested agent model identifiers.

Tool exposure. MCP1 exposes three entry points: catalog discovery, tool description, and a gateway that calls a named operation. MCP2 exposes 176 entry points, including direct editing and delegated tools. These names include asset and Blueprint inspection, Blueprint graph editing and compilation, asset creation and editing, text and C++ file editing with in-editor Python execution, play-in-editor control, and more. A gateway can access many operations, so entry-point counts do not measure tool coverage. Table 2 gives file and shell permissions. Initialization failures count as failed attempts.

Timing and comparison scope. The runner used a 60-minute execution ceiling. Completion required the agent to finish within 40 minutes and its final submission to pass all required checks. The submitted project is graded afterward. The C++ baseline uses two hosts, whereas its MCP comparisons use one. Both hosts use the same hardware configuration, each with an NVIDIA GeForce RTX 5080 GPU. On BP and Python tasks, MCP1 also retains the native write, edit, and shell tools that MCP2 does not (Table 2), and some MCP1 runs on these surfaces issue native file or shell writes, so this permission difference is exercised in the recorded sessions. Driver versions vary within and across task groups, and some records lack version metadata. These differences limit attribution of completion or timing differences to MCP access alone.

Behavior (L2) checks launch the editor with `-deterministic -FPS=60`, which fixes the simulation step at 1/60 second from the first frame; a task may declare further legs at other fixed steps, which are replayed in separate sessions and must all pass. Checkpoints are evaluated against the world’s game time, counted from when play begins rather than from when the test fixture starts.

Agent inputs and testing. Each run supplies the task prompt and starting project, specifies where to save changes, and instructs the agent to perform at most one PIE playtest. The limit is prompt-level only: the harness does not intercept or refuse play-start requests during authoring, so additional playtests consume the run’s authoring budget rather than being blocked; however, there is no such request in the current trajectories. Play-start requests are instead recovered from the trajectories after the run (Appendix D.1). The agent receives no human feedback or grading reports during authoring. The harness grades its saved submission afterward.

Saved submissions. The harness collects changed or new files accepted by the starting project’s submission manifest and applies them to a fresh starting project. The grader applies the task’s additional file and configuration constraints before evaluating the reconstructed project. This procedure makes the saved submission, rather than unsaved editor state, the object of evaluation.

B.1 ACCESS TO EVALUATION MATERIAL

Before authoring, the harness withholds reference implementations, verifier sources, and repository metadata that could expose them. It also removes other tasks’ scaffolds and replaces runtime test bodies with stubs while retaining the build definitions needed to compile the authoring project. Withheld material is restored for grading in the clean project.

The harness monitors the withheld paths and excludes an attempt from scoring if it detects exposed evaluation material. Task instructions and permitted starting content remain available during authoring.

B.2 RELEASE MATERIALS

The release includes the harness, task instructions, starting-project setup, verifiers, reference implementations, retained submissions, and analysis code. Run identifiers and source hashes connect the reported panel, check results, and trajectory classifications to their underlying records. Public trajectories omit credentials and private content while retaining the feedback and submission evidence needed to inspect the analyses. Users provide an Unreal Engine installation; engine binaries and proprietary MCP2 internals are not redistributed.

C DETAILED RESULTS AND SCORING CHECKS

This appendix gives detailed results for the 70-task suite shown in Figure 5 and its ten paired gameplay specifications. Each record is identified by task, model, and configuration. The resulting panel has no missing or duplicate combinations. Completion requires `overall=PASS` and meets the 40-minute time budget. Build checks are L1, runtime checks of gameplay are L2, and asset checks of saved structure are L2I.

BP task groups. We group BP tasks by their instructions: three basic graph tasks (logging, arithmetic, and delayed movement), two local repairs (door hitch and inventory stacking), ten asset-configuration tasks, and ten paired gameplay tasks. Asset-configuration tasks check saved structure; gameplay tasks also check runtime behavior.

Counting runtime passes. The ten C++/BP gameplay pairs share runtime fixtures; BP also requires asset checks. Counting recorded runtime passes within 40 minutes leaves all four completion counts unchanged (Table 9). Every counted runtime pass also passes the build checks.

All 20 BP submissions that fail asset checks have runtime reports, including five MCP1 and seven MCP2 submissions returned within 40 minutes. Five attempts have `FAIL_NO_EDITS` and no runtime outcome; they remain failures in the denominator. One MCP1 submission passes runtime but fails asset checks after 52.1 minutes. Runtime automation failures remain failures even when individual messages report successful behavior.

Python input bindings. The input-binding task checks action types, six key bindings, and their modifiers across four input assets. Scores use the engine’s active `DefaultKeyMappings.Mappings` array in the retained submissions. These checks assess saved input configuration without simulating keypresses.

D TRAJECTORY ANALYSIS

D.1 INSPECTION AND VERIFICATION IN AGENT TRAJECTORIES

The trajectory analysis covers 693 C++ transcripts, 350 BP transcripts, and 168 Python transcripts, using the run membership and reported outcomes in Appendix C. Counts cover full authoring sessions unless a time limit is stated. For MCP1, the operation is read from the gateway request’s toolset and operation arguments. Requests are matched to returns by tool-use identifier and occurrence; seven requests lack retained returns, and six nested calls in the baseline C++ traces are

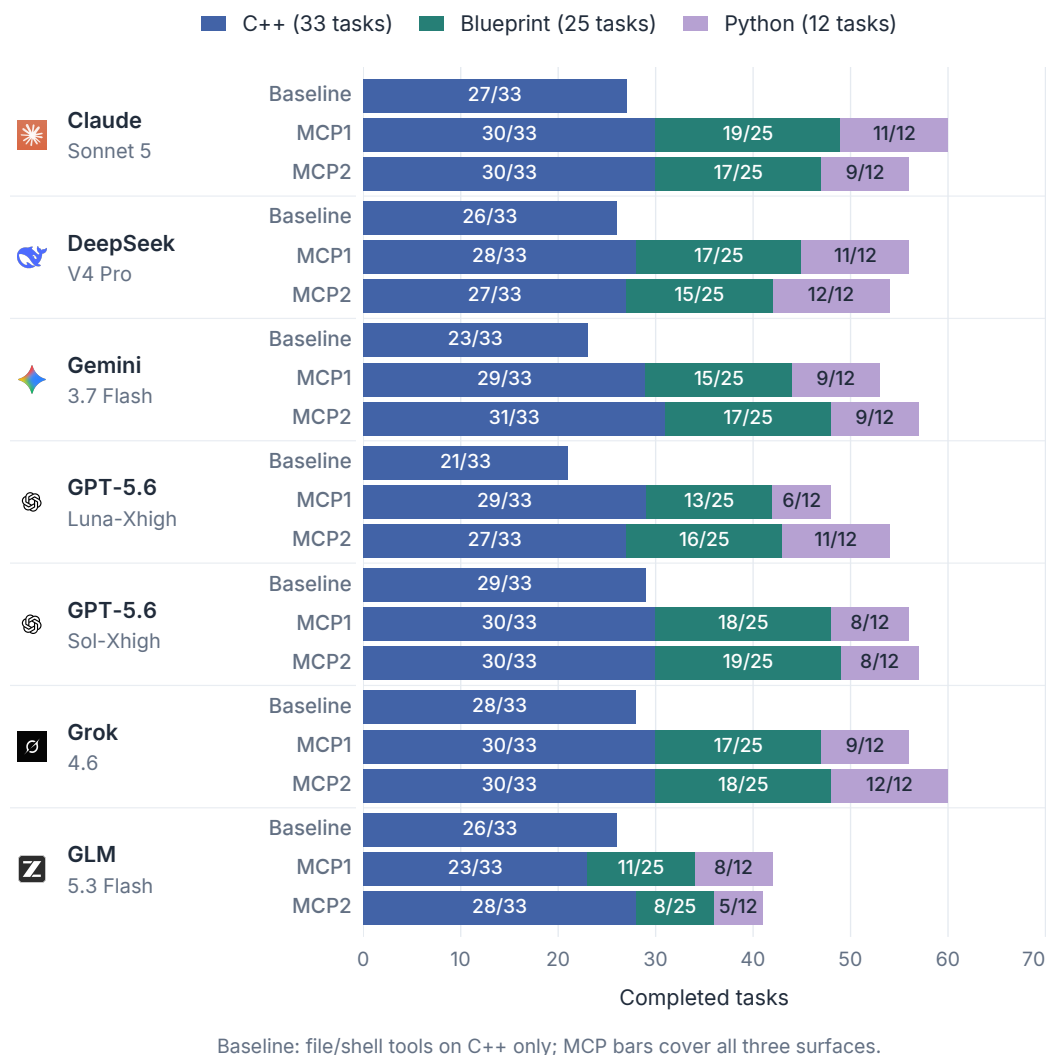


Figure 5: Task completion within 40 minutes, by model and configuration. Each model has three stacked bars: Baseline, MCP1, and MCP2. Colored segment lengths indicate the counts of completed C++, Blueprint, and Python tasks. Labels inside each segment give completed tasks out of 33 C++, 25 Blueprint, or 12 Python tasks. The baseline uses file/shell tools and was evaluated only on C++, so its total bar length is not comparable with those of the MCP configurations. Table 6 gives the same counts.

C++			
Model	File/shell baseline	MCP1	MCP2
Claude Sonnet 5	27/33 (81.8%)	30/33 (90.9%)	30/33 (90.9%)
DeepSeek V4 Pro	26/33 (78.8%)	28/33 (84.8%)	27/33 (81.8%)
Gemini 3.7 Flash	23/33 (69.7%)	29/33 (87.9%)	31/33 (93.9%)
GPT-5.6 Luna	21/33 (63.6%)	29/33 (87.9%)	27/33 (81.8%)
GPT-5.6 Sol	29/33 (87.9%)	30/33 (90.9%)	30/33 (90.9%)
Grok 4.6	28/33 (84.8%)	30/33 (90.9%)	30/33 (90.9%)
GLM 5.3 Flash	26/33 (78.8%)	23/33 (69.7%)	28/33 (84.8%)

BP		
Model	MCP1	MCP2
Claude Sonnet 5	19/25 (76.0%)	17/25 (68.0%)
DeepSeek V4 Pro	17/25 (68.0%)	15/25 (60.0%)
Gemini 3.7 Flash	15/25 (60.0%)	17/25 (68.0%)
GPT-5.6 Luna	13/25 (52.0%)	16/25 (64.0%)
GPT-5.6 Sol	18/25 (72.0%)	19/25 (76.0%)
Grok 4.6	17/25 (68.0%)	18/25 (72.0%)
GLM 5.3 Flash	11/25 (44.0%)	8/25 (32.0%)

Python		
Model	MCP1	MCP2
Claude Sonnet 5	11/12 (91.7%)	9/12 (75.0%)
DeepSeek V4 Pro	11/12 (91.7%)	12/12 (100.0%)
Gemini 3.7 Flash	9/12 (75.0%)	9/12 (75.0%)
GPT-5.6 Luna	6/12 (50.0%)	11/12 (91.7%)
GPT-5.6 Sol	8/12 (66.7%)	8/12 (66.7%)
Grok 4.6	9/12 (75.0%)	12/12 (100.0%)
GLM 5.3 Flash	8/12 (66.7%)	5/12 (41.7%)

Table 6: Full-task completions within 40 minutes, with counts and percentages shown separately for each surface. The file/shell baseline is evaluated only on C++.

Model	MCP1		MCP2	
	C++	BP	C++	BP
Claude Sonnet 5	8/10	5/10	8/10	3/10
DeepSeek V4 Pro	6/10	5/10	6/10	2/10
Gemini 3.7 Flash	6/10	5/10	8/10	3/10
GPT-5.6 Luna	6/10	1/10	6/10	4/10
GPT-5.6 Sol	8/10	6/10	8/10	4/10
Grok 4.6	8/10	2/10	8/10	4/10
GLM 5.3 Flash	4/10	1/10	6/10	0/10

Table 7: Per-model completion within 40 minutes on the ten paired gameplay specifications.

C++ task subset	Baseline	MCP1	MCP2
Ten paired gameplay tasks	45/70	46/70	50/70
Other 23 C++ tasks	135/161	153/161	153/161

Table 8: C++ completion within 40 minutes, split by whether the specification also has a BP version. The split follows the task definitions. It is not selected from outcomes.

excluded. The counts describe visible agent requests and returns; unobserved work inside tool services is outside this analysis.

Asset and PIE state readback. An *asset readback* is a returned value from a task-owned asset, excluding errors, search results, schema descriptions, and object handles without state. A *PIE state*

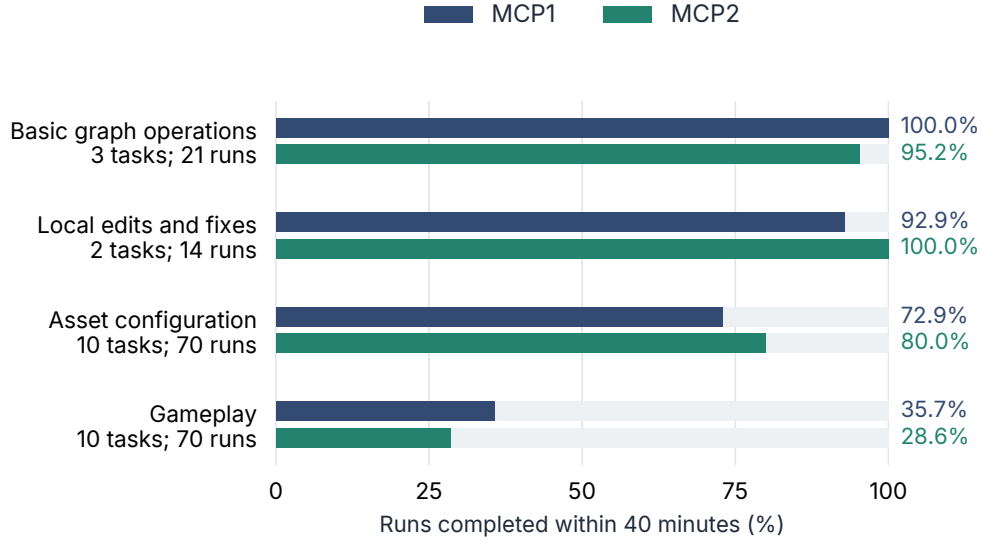


Figure 6: BP completion by task type. Row labels give the number of tasks and runs per configuration (seven models per task). Groups follow the task instructions and differ in content, starting state, and required checks.

Configuration	C++ full	BP full	BP runtime pass	Gap (pp)
MCP1	46/70	25/70	25/70	30.0
MCP2	50/70	20/70	20/70	42.9

Table 9: Completion on the ten paired gameplay tasks, with and without the BP asset checks. Each task group has 70 attempts per configuration. All counts require authoring to finish within 40 minutes. Counting recorded BP runtime passes leaves the full-task counts and the C++-BP gap unchanged; the original submissions are used in both counts. Gap is measured in percentage points.

readback contains a value measured while the game runs in Play-in-Editor (PIE). It may come from a dedicated getter, a property or transform query targeting a UEDPIE object, an inspected script, a combined action sequence, or a recording. A *PIE start request* counts the request to start the game, including starts within identified scripts. A start request alone is insufficient for a state readback.

Table 4 reports these methods for all BP runs; Table 10 separates them by model. Images count separately and require an image block in the agent’s conversation: seven and eight runs qualify, respectively. Script-based readbacks require returned measurements rather than a printed success statement. Compilation counts requests; a return without an error, including a null return, does not establish successful compilation.

On C++, 151 reviewed shell requests ask for game or engine-test execution, covering 57/231 MCP1 and 39/231 MCP2 runs. Two further requests only list available tests and are excluded. The counts include failed starts and background requests whose completion is unconfirmed. These shell operations are separate from PIE start requests.

On Python, ten MCP2 runs across six tasks and five models contain tool/API discovery, a task-directed modification script, an explicit `bSuccess:true` return, and subsequent asset readback in the task’s namespace. The readback need not concern the modified field. MCP1 has a different return protocol, so this sequence is not used to rank script success between configurations.

Graph revisions. All 175 MCP1 BP trajectories are screened for direct graph writing. The 95 qualifying runs contain 454 writes with complete request–return matching; 75 read task assets before their first graph write. An *accepted graph write* returns without a reported tool error. A revision

Model	PIE start request		PIE state readback		Image return	
	MCP1	MCP2	MCP1	MCP2	MCP1	MCP2
Sonnet	17	18	12	11	1	5
DeepSeek	4	6	2	4	2	0
Gemini	2	4	1	3	0	0
Luna	0	2	0	1	0	0
Sol	0	0	0	0	0	0
Grok	1	1	1	1	0	0
GLM	14	13	8	10	4	3

Table 10: BP inspection methods by model, out of 25 runs per configuration. PIE start requests include identified scripts. PIE state readbacks require returned measurements from the running game. An image must appear in the agent’s conversation. Rows use the definitions in Appendix D.1.

changes the arguments for the same graph after the previous write returns. Six malformed writes have no recoverable graph identity and cannot be paired. *Immediate* means that the revision is the next agent tool request, rather than a short elapsed time.

Table 11 distinguishes revisions after accepted and rejected writes, together with intervening readbacks and compilation returns. Revisions can extend functionality as well as correct errors.

Observed sequence	All (95)	Same tasks (49)	≤ 40 min (79)
Accepted write, then changed same-graph write	30	16	20
Readback between those writes	27	15	17
Compilation return between those writes	21	11	12
Both readback and compilation return	18	10	9
Error, then immediate same-graph change	37	20	26
Acceptance, then immediate same-graph change	4	1	4

Table 11: Graph revisions in MCP1 BP runs. Columns cover all direct graph-writing runs, the seven tasks on which every model directly writes graphs, and sessions ending within 40 minutes. Rows overlap. A compilation return need not report successful compilation. Immediate means the next agent tool request after the preceding write returns; 58 runs contain errors, and 30 revise accepted writes.

Table 12 links graph-writing errors and subsequent accepted revisions to submission outcomes. Of the 16 on-time failures after every reported error receives an accepted revision, 15 fail explicit run-time checks, spanning six models and eight gameplay tasks.

Table 12: Graph-writing errors and subsequent revisions in MCP1 BP runs. Rows successively restrict the 175-run panel. Accepted revisions return without a reported tool error. On-time passes finish authoring within 40 minutes; final passes include later submissions.

Sequence	Runs	On-time pass	Final pass
Uses direct graph writing	95	55	62
Receives graph-writing error	58	27	34
Changed same-graph request accepted	55	26	32
Every error has a later accepted change	50	23	29
Previous row, session ends within 40 min	39	23	23

D.2 UNRESOLVED RESOURCE REFERENCES

Five MCP1 BP submissions across five tasks and two models contain both an unresolved-resource warning and a failed runtime check. Two omit the resource’s owning class; three use a class-qualified name that the engine cannot resolve. The warnings establish unresolved references, but need not explain every failed check: the healing warning concerns Health, while the failed check concerns MaxHealth initialization. The census covers retained runtime reports from 101/105 MCP1 and

103/105 MCP2 runs on the 15 BP tasks that declare runtime checks, including 66/70 and 69/70 paired gameplay runs.

UE 5.8 source inspection identifies two supported lookup routes: an owning class plus a member name, or a valid full field path. No submission was modified to test whether resolving its reference would complete the task.

D.3 RUNTIME FAILURES AFTER PASSING ASSET CHECKS

Of the 140 BP attempts on matched gameplay tasks, runtime reports are retained for 66/70 under MCP1 and 69/70 under MCP2. The five attempts without reports have FAIL_NO_EDITS outcomes. We select submissions that pass asset checks and fail an explicit runtime assertion: 24/70 and 28/70. One additional MCP1 automation failure has no such assertion and is excluded from this category.

Failure after passing asset checks	MCP1	MCP2
Initial Health or Power	3	4
Required effect or resource use	8	6
Pose, display, or object indicator	5	8
Restoration or preserved behavior	5	6
Poison damage ratio	3	4
Total attempts	24/70	28/70
Authoring finished within 40 minutes	19/70	20/70

Table 13: BP submissions that pass asset checks but fail a runtime check. Each of the 52 selected attempts has one observed failure category; subsequent checks may not execute if a previous check failed.

Selected BP submissions	MCP1	MCP2
On-time submissions passing asset checks	45	40
Runtime checks pass	25	20
Explicit runtime assertion failure	19/45 (42.2%)	20/40 (50.0%)
Runtime automation failure	1	0

Table 14: Runtime outcomes among BP submissions on the ten gameplay specifications that pass asset checks and finish authoring within 40 minutes. The last three rows partition this selected subset; automation failures are separate from failed assertions.

Categories group the state measured by failed assertions, with duplicate messages removed within each attempt. Each of these 52 attempts has one observed category; later checks may not execute. The reports evaluate the final saved submissions.

Thirteen effect or resource failures outside poison stacking also show the ability got activated successfully: seven under MCP1 and six under MCP2, across area burn, double jump, glide, and Health operations. Seven concern gliding; six consume Power before any verifier-induced exhaustion. Some reports contain no valid glide sample or no forced-exhaustion stage, so these observations do not establish a common failure mechanism across all seven submissions, but show how agents might fail when implementing features. Eight versioned verifier source objects establish how activation and resource use were measured. C++ failure selection requires passing build checks; BP selection requires passing asset checks.

E AUTHORING TIME AND COST

For all seven models, median C++ authoring time is longer with either MCP configuration than with the Baseline (Figure 7). Durations include unsuccessful runs and sessions that exceed the 40-minute deadline; completion still requires finishing within that deadline.

Input context and billed cost. Of 231 C++ MCP task-model pairs, 229 have identical retained task prompts and 185 have both first requests linked to billing records. In these 185 pairs, median

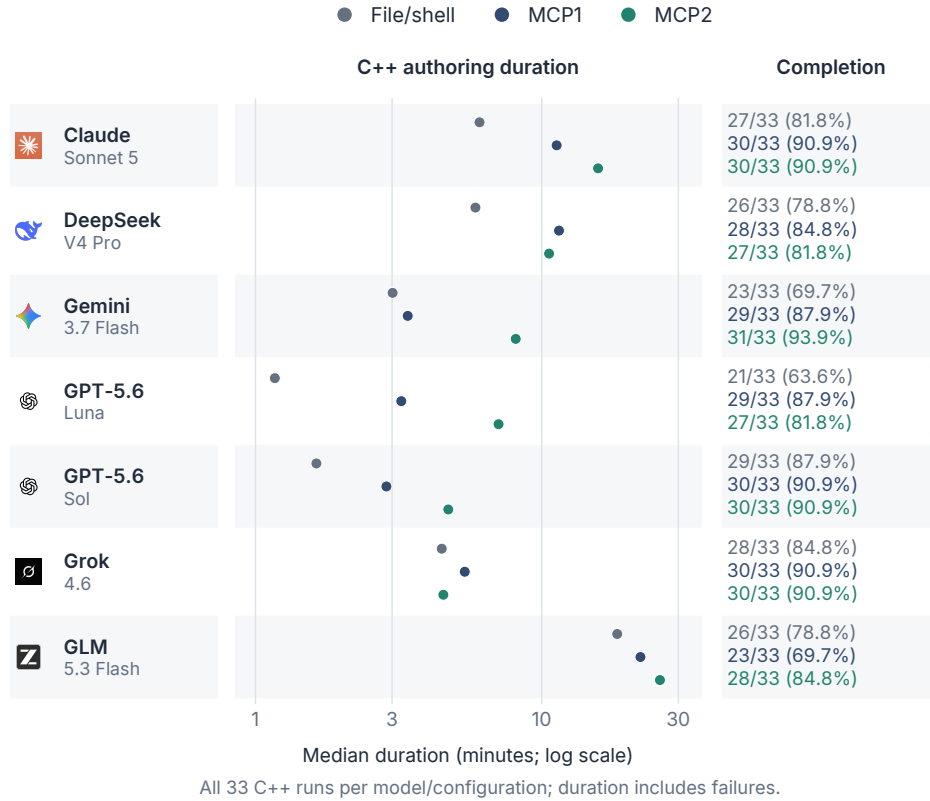


Figure 7: C++ completion and median authoring time on the same 33 tasks per configuration. Markers show median duration on a logarithmic scale; labels give completion counts. All runs enter the duration calculation.

first-request input is 3.99 times as large under MCP2 as under MCP1, a median difference of 82,354 tokens. Each configuration retains its own system instructions and tool descriptions.

Full-session billing is available for 162 prompt-matched pairs (Table 15). The median paired MCP2/MCP1 charge ratio is 0.21 for DeepSeek and 1.80–2.69 for the other five covered models. Cache use also differs by model: among 19 DeepSeek pairs with complete token records, the median share of input served from cache is 3.47% under MCP1 and 98.21% under MCP2; for Sol’s 33 pairs, it is 92.12% and 93.17%.

Model	Pairs	MCP1	MCP2	Charge ratio	Higher
Claude Sonnet 5	0/33	—	—	—	—
DeepSeek V4 Pro	20/33	18/20	19/20	0.21×	6/20
Gemini 3.7 Flash	32/33	28/32	31/32	1.86×	23/32
GPT-5.6 Luna	29/33	27/29	25/29	2.48×	29/29
GPT-5.6 Sol	33/33	30/33	30/33	2.69×	33/33
Grok 4.6	31/33	28/31	29/31	1.80×	29/31
GLM-5.3 Flash	17/33	15/17	16/17	2.09×	17/17
C++ linked subset	162/231	146/162	150/162	2.21×	137/162

Table 15: C++ completion and billed inference on the matched subset. Completion columns count passes within 40 minutes. *Charge ratio* is the median paired MCP2/MCP1 ratio; *Higher* counts pairs with larger MCP2 charges. Pair counts give billing coverage out of 33 tasks per model.

Accounting. We match generation IDs from CLI summaries and assistant messages to OpenRouter billing records, require the requested model and a numeric charge for every observed generation, deduplicate IDs across four ledgers, and sum `cost_details.upstream_inference_cost`. This excludes unrelated projects in the same ledgers. On fully linked C++ runs, CLI cost estimates exceed charges by model-dependent median factors of 1.25–54.85; we use the billed amounts directly, without a conversion factor. Charges cover full agent sessions, including requests after the deadline, and exclude engine compute and unobserved inference inside tools. BP, Python, and direct-Anthropic Sonnet runs have no usable billing linkage and are omitted from cost comparisons.

Input totals add the ledger’s separate uncached, cache-read, and cache-write fields; cache shares use that total as the denominator. The result-summary input field already includes cached tokens and is never added to its cache fields.

F TASK CATALOG AND PAIRED GAMEPLAY SPECIFICATIONS

Table 16 lists every task in the primary study, its starting project, and its required checks. The 50 Third Person tasks use Unreal’s standard template; the other 20 use the CraftBench template. Task names are directory identifiers so that a reader can locate the instructions and verifier directly. Table 17 gives the requested behaviors in the 10 C++/BP pairs. Their task definitions name the same interface and verifier, while BP additionally requires saved BP graphs rather than C++.

The task bank also contains 47 additional tasks outside the primary study that were excluded due to the time and budget limit. They do not enter the reported completion rates or trajectory counts.

Table 16: The 70 study tasks, with their starting projects and required grading gates. IDs refer to task directories. Build, runtime, and inspect denote L1, L2, and L2I; ✓ marks a required check and — marks one not required. The C++, BP, and Python groups contain 33, 25, and 12 tasks.

Row	Task ID	Starting project	Build	Runtime	Inspect
C++ source tasks					
C01	gp-crafting-queue	CraftBench template	✓	✓	—
C02	gp-dot-aoe-burn-cpp	Third Person	✓	✓	—
C03	gp-double-jump-stamina-cpp	Third Person	✓	✓	—
C04	gp-glide-stamina-cpp	Third Person	✓	✓	—
C05	gp-harvestable-regrow	CraftBench template	✓	✓	—
C06	gp-heal-over-time-cpp	Third Person	✓	✓	—
C07	gp-health-attribute-ops-cpp	Third Person	✓	✓	—
C08	gp-inventory-stacking	CraftBench template	✓	✓	—
C09	gp-poison-dot-stack-cpp	Third Person	✓	✓	—
C10	gp-spawner-population	CraftBench template	✓	✓	—
C11	t0-sanity-log-on-beginplay	CraftBench template	✓	✓	—
C12	t1-data-asset-drives-speed	CraftBench template	✓	✓	—
C13	t1-datatable-drives-value	CraftBench template	✓	✓	—
C14	t1-default-cube-mesh-actor	CraftBench template	✓	✓	—
C15	t1-extraction-volume-per-actor-trigger	CraftBench template	✓	✓	—
C16	t1-gameplay-tag-gate	CraftBench template	✓	✓	—
C17	t1-movement-component-drives-actor	CraftBench template	✓	✓	—
C18	t1-mud-wade-cpp	Third Person	✓	✓	—

Continued on next page

Table 16 (continued)

Row	Task ID	Starting project	Build	Runtime	Inspect
C19	t1-overlap-logs-once	CraftBench template	✓	✓	—
C20	t1-overlap-teleport-portal	Third Person	✓	✓	—
C21	t1-physics-drop-and-rest	CraftBench template	✓	✓	—
C22	t1-screen-tint-cpp	Third Person	✓	✓	—
C23	t2-gravity-floating-pawn-movement	CraftBench template	✓	✓	—
C24	t2-homing-projectile	CraftBench template	✓	✓	—
C25	t2-hud-layout-and-countdown	Third Person	✓	✓	—
C26	t2-ladder-climb-volume	Third Person	✓	✓	—
C27	t2-melee-ability-with-cooldown	CraftBench template	✓	✓	—
C28	t2-npc-follows-player	Third Person	✓	✓	—
C29	t2-race-clock-cpp	Third Person	✓	✓	—
C30	t2-timeline-color-cycle	CraftBench template	✓	✓	—
C31	t2-weapon-fire-animation-on-trigger	Third Person	✓	✓	—
C32	t3-gate-and-door-cpp	Third Person	✓	✓	—
C33	tp2-sprint-stamina	Third Person	✓	✓	—
Blueprint/native-asset tasks					
B01	gp-additem-stack-fix-bp	Third Person	✓	✓	—
B02	gp-door-hitch-fix-bp	Third Person	✓	✓	—
B03	gp-dot-aoe-burn-bp	Third Person	✓	✓	✓
B04	gp-double-jump-stamina-bp	Third Person	✓	✓	✓
B05	gp-glide-stamina-bp	Third Person	✓	✓	✓
B06	gp-heal-over-time-bp	Third Person	✓	✓	✓
B07	gp-health-attribute-ops-bp	Third Person	✓	✓	✓
B08	gp-poison-dot-stack-bp	Third Person	✓	✓	✓
B09	t0-sanity-bp-log-on-beginplay	CraftBench template	✓	✓	—
B10	t1-blueprint-event-to-action	CraftBench template	✓	✓	—
B11	t1-blueprint-graph-on-beginplay	CraftBench template	✓	✓	—
B12	t1-dawn-fog-lighting-rig	Third Person	✓	—	✓
B13	t1-hero-blueprint-copy-with-flashlight	Third Person	✓	—	✓
B14	t1-mud-wade-bp	Third Person	✓	✓	✓
B15	t1-playable-level-bootstrap	Third Person	✓	—	✓
B16	t1-screen-tint-bp	Third Person	✓	✓	✓
B17	t1-third-person-chase-camera	Third Person	✓	—	✓
B18	t1-walk-animation-footstep-cues	Third Person	✓	—	✓
B19	t2-consistent-enum-names	Third Person	✓	—	✓
B20	t2-cutscene-camera-push-and-hero-rise	Third Person	✓	—	✓
B21	t2-datatable-csv-export	Third Person	✓	—	✓
B22	t2-race-clock-bp	Third Person	✓	✓	✓
B23	t2-weapon-held-in-right-hand	Third Person	✓	—	✓
B24	t3-gate-and-door-bp	Third Person	✓	✓	✓
B25	t3-piercing-projectile	Third Person	✓	—	✓
Editor-scripting tasks					
P01	kp-anim-track-bake	Third Person	✓	—	✓
P02	kp-blueprint-actor-audit-report	Third Person	✓	—	✓
P03	kp-character-boom-and-movement	Third Person	✓	—	✓
P04	kp-config-source-audit	Third Person	✓	—	✓
P05	kp-derived-class-search	Third Person	✓	—	✓
P06	kp-engine-source-search	Third Person	✓	—	✓
P07	kp-fog-and-postprocess-rig	Third Person	✓	—	✓
P08	kp-motion-set-shares-one-rig	Third Person	✓	—	✓

Continued on next page

Table 16 (continued)

Row	Task ID	Starting project	Build	Runtime	Inspect
P09	kp-retarget-maps-two-rigs	Third Person	✓	—	✓
P10	kp-routine-usage-search	Third Person	✓	—	✓
P11	kp-spawn-level-actors	Third Person	✓	—	✓
P12	kp-typed-input-bindings	Third Person	✓	—	✓

Task family	Requested behavior	Catalog rows
Area burn	Apply periodic damage inside a bounded area and stop after its duration.	C02 / B03
Double jump	Reverse a fall with an upward impulse; debit Power once and reject insufficient Power.	C03 / B04
Glide	Slow descent while draining Power and restore normal falling when Power is exhausted.	C04 / B05
Healing over time	Restore Health repeatedly for a limited duration without exceeding MaxHealth.	C06 / B06
Health operations	Initialize writable Health and provide repeatable damage and healing abilities.	C07 / B07
Stacking poison	Apply periodic damage with up to three stacks and refresh the duration on reapplication.	C09 / B08
Mud wading	Slow movement and change animation in mud, restore both afterward, and preserve the unaffected character.	C18 / B14
Speed-driven tint	Track movement speed relative to the current maximum while preserving an unrelated screen setting.	C22 / B16
Race clock	Run a timed coin-collection round with live readouts, final scoring, and restart behavior.	C29 / B22
Gates and door	Open each new gate for its required pair of named crates while preserving the existing door behavior.	C32 / B24

Table 17: The ten gameplay specifications with C++ and Blueprint counterparts. Each pair names the same starting project, runtime map, and test class in its task definitions. Both counterparts require build and runtime checks; the Blueprint counterpart also requires artifact inspection. These summaries describe the requested behavior; the task prompts specify the acceptance tolerances and deliverables.

G COMPARISON WITH GAME-DEVELOPMENT BENCHMARKS

Table 18 compares what agents are asked to produce and the evidence used to evaluate it. Different task scopes support different claims: atomic gameplay checks, visual cutscene quality, and complete game generation are not interchangeable outcomes.

Benchmark	Setting and task source	Agent’s deliverable	Grading evidence
GameDevBench (Chi et al., 2026)	Godot; 333 tasks from web/video tutorials	Scripts, scenes, resources	In-engine tests of behavior and scene state
GameEngineBench (La et al., 2026)	Unreal; 110 tasks from nine projects	C++ changes in existing projects	Build and runtime tests, followed by an LLM behavioral judge
OpenGameEval (Zhang et al., 2025)	Roblox; expert-authored tasks (47 in the initial release)	Scripts and changes to existing experiences	Executable tests in Roblox Studio
GBQA (Jiang et al., 2026)	Web games; 124 human-verified injected bugs in 30 agent-built games	Bug reports from interactive exploration	LLM critic matches reports to ground truth; recall of verified bugs
CutsceneBench (He et al., 2026)	Unreal; 65 authored cutscene scenarios	Level Sequence assets	Tool and sequence checks; model-judged quality on 25 sampled videos per model
AutoUE (Yin et al., 2026)	Unreal; complete game generation	Game projects with scenes and gameplay	Automated play-testing; LLM-judged game quality
GameXpert-Bench (Chen et al., 2026)	No prescribed engine, browser JavaScript; 97 generation tasks, 100 repair tasks from 50 human-verified levels, 102 optimization turns	All game source files, written from a blank workspace; source patches for repair	Deterministic headless-browser probes (repair); evaluation-agent rubric with human rating (generation); hidden-rubric judging (optimization)
CraftBench-UE	Unreal; 70 bounded gameplay/editor tasks	Source changes, native assets, editor-script outputs	Task-specific build, runtime, and saved-asset checks; no LLM judge

Table 18: Benchmark designs, using the cited releases. The OpenGameEval count refers to its initial release; its task bank has since expanded. CraftBench-UE also includes 10 C++/BP gameplay pairs and evaluates two editor MCP configurations.