AGENT'S GAMEDEV EXAM

Anonymous authors
Paper under double-blind review

ABSTRACT

Evaluating AI agents for game development requires testing whether they can translate requirements into working gameplay within complex engine environments. We introduce AGENT'S GAMEDEV EXAM, a production-oriented benchmark spanning Unreal Engine 5, Unity, Godot, and RPG Maker, with 448 tasks from 55 source projects and packages. Two complementary tracks assess component-level feature development within existing projects and End2end-level game creation from supplied assets, functional specifications, and reference screenshots. Our construction pipeline combines expert-defined requirements, LLM-assisted task preparation, and structural and in-engine validation; component-level tasks remove complete feature implementations while preserving the surrounding project. Evaluation combines deterministic checks with ground-truth-referenced video assessment to test requested functionality and, where applicable, preservation of existing behaviors. Across nine model–harness configurations, the current evaluation yields a best component-level full-pass rate of 14.23%, while no configuration achieves a full pass on the End2end-level track. Analysis of UE5 component-level submissions identifies incomplete execution paths and disconnected state-driven effects as the dominant issue categories, accounting for 48.3% and 18.3% of categorized issues, respectively. These findings expose a gap between locally plausible implementations and complete feature delivery, positioning AGENT'S GAMEDEV EXAM as a testbed for improving system-level reasoning, integration, and runtime verification in game-development agents.

1 INTRODUCTION

Game development is an economically consequential domain for AI agents, with global games revenue forecast at $213.9 billion in 2026 (Buijsman, 2026). AI tools are entering professional workflows: 30% of game-studio respondents in the 2026 GDC survey report using generative AI at work (Elderkin, 2026). Assessing progress toward agents that work as professional game developers requires reliable evaluation of real workflows: understanding an existing game's context, translating gameplay requirements into technical designs, and delivering features suited to the project's constraints or building a runnable game end to end Yet how much of this work current agents can reliably complete remains unclear. A realistic benchmark is therefore needed both to measure these capabilities and to guide advances toward practical use in game production.

Existing evaluation settings cover browser-native development, tutorial-based multimodal tasks, and engine-native source editing (Chen et al., 2026; Chi et al., 2026; La et al., 2026). These settings provide valuable evidence within their scope, but building a broad, realistic game-development benchmark presents four challenges for reliable evaluation. **First, obtaining high-quality projects and constructing valid tasks** requires more than collecting source code: compatible assets, scenes, and dependencies must form a runnable starting point, and the development objective must be well-specified and solvable without exposing reference solutions. **Second, systematic, scalable operation across professional game engines** requires both controlling complex authoring environments and accommodating heterogeneous engine workflows. For engines such as Unreal Engine 5 and Unity, benchmark infrastructure must coordinate code, assets, scenes, editor state, and runtime execution through engine-specific interfaces, while maintaining repeatable setup and comparable evaluation conditions across engines. **Third, evaluating multimodal behavior over time** cannot rely on compilation checks or static screenshots: evaluators must assess gameplay interactions, animations,

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.

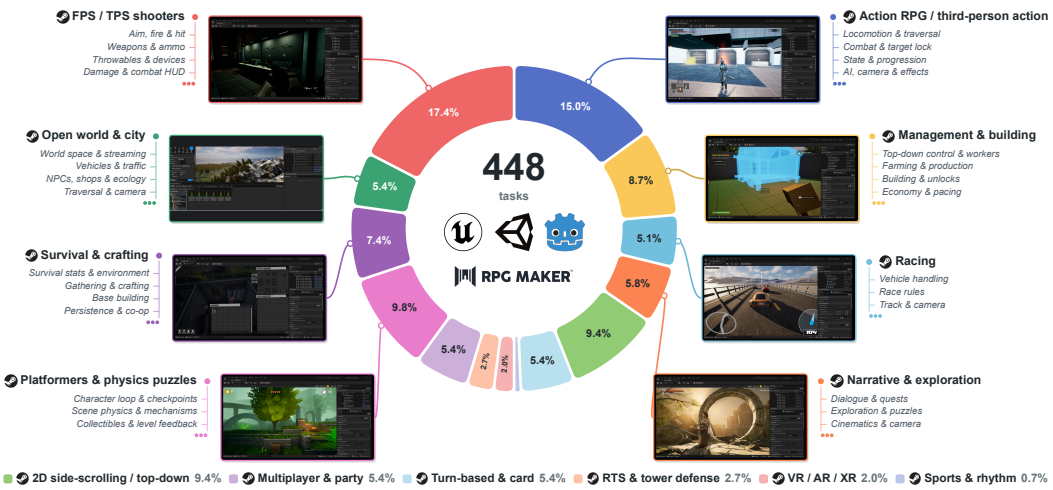


Figure 1: Task distribution of AGENT'S GAMEDEV EXAM across 14 coverage groups according to STEAM: 448 tasks from 55 source projects/packages. Ring sectors and percentage labels use the same task counts as Table 3 in Appendix A. Percentages may not sum to 100% due to rounding. Branches illustrate eight groups with representative screenshots and feature families; the remaining six appear in the legend. Steam logos mark approximate mappings to Steam tags, not an official Steam taxonomy or evidence that the source projects are distributed on Steam.

and state transitions across execution sequences, including whether specified existing behaviors remain intact. Because a requirement can admit multiple valid implementations, acceptance criteria must reject superficially convincing but functionally incorrect results without requiring the reference project's code or incidental presentation choices. **Fourth, covering an uneven development-task distribution** requires deliberate selection across engines, genres, and feature families; relying only on readily available source projects risks concentrating the benchmark on a narrow subset of game development. Together, these challenges determine whether benchmark scores provide credible evidence of game-development capability rather than artifacts of task construction, execution conditions, or sampling.

> To address this gap, we introduce AGENT'S GAMEDEV EXAM, a **production-oriented**, **multi-engine** benchmark with **comprehensive** coverage of **long-horizon**, **verifiable** game-development tasks.

In summary, our work makes three contributions toward reliable evaluation of AI agents in game development:

- **A production-oriented, multi-engine benchmark.** We introduce AGENT'S GAMEDEV EXAM, grounded in real game projects and spanning Unreal Engine 5, Unity, Godot, and RPG Maker, with two complementary tracks for component-level feature development and End2end-level game creation.
- **A systematic, scalable, and reliable data-generation pipeline.** We combine expert-defined requirements, LLM-assisted task construction, standardized packaging, and structural and in-engine validation to generate usable task inputs, screen for solution leakage, and preserve specified existing behaviors.
- **Empirical evaluation and failure analysis.** We evaluate nine models across both tracks and analyze failure cases in UE5 component-level submissions, identifying recurring gaps between locally plausible implementations and complete, working game features.

2 RELATED WORK

General-purpose agent evaluation. SWE-bench and FeatureBench evaluate repository-level issue resolution and missing-feature implementation, respectively (Jimenez et al., 2024; Zhou et al.,

Benchmark	Task	Real	Prod.	UE	Unity	Multi	Coverage
GameDevBench (Chi et al., 2026)	Feature	✓	–	×	×	×	4 skill categories
JAMER (Sun et al., 2026a)	Completion	✓	–	×	×	×	>40 genres (corpus)
GameEngineBench (La et al., 2026)	C++ impl.	✓	✓	✓	×	×	14 task areas
GameXpert (Chen et al., 2026)	Fix	–	✓	×	×	×	61 bug types
	Opt	✓	–	×	×	×	6 refinement axes
AgencyBench: Game (Li et al., 2026)	Incremental	–	–	×	×	×	10 game scenarios
GameCraft-Bench (Luo et al., 2026)	Generation	×	–	×	×	×	15 families
AutoUE (Yin et al., 2026)	Generation	×	✓	✓	×	×	3 difficulty levels
Mage (Liu & Tatar, 2026)	Scene gen.	–	–	×	✓	×	26 goal patterns
ALE: relevant tasks (Sun et al., 2026b)	Authoring	✓	✓	✓	×	✓	-
AWoMo (Zhou et al., 2026a)	Asset/scene	✓	✓	✓	✓	✓	-
AGENT'S GAMEDEV EXAM (ours)	**Component + End2end**	✓	✓	✓	✓	✓	**448 tasks / 55 projects**

Table 1: **Practical focus and engine coverage.** Real: existing projects/tutorials or recorded workflows. Prod.: author-stated professional workflows or production-informed tasks, not deployment. Multi: multiple engines, including asset transfer. ✓: reported; ×: outside scope; –: unclear/not reported. Ours also includes Godot and RPG Maker (2D). Coverage uses author-reported units, not a common ranking; NR: category count not established for the listed scope. JAMER reports corpus-wide genres, not test-subset coverage.

2026b). SWE-bench Multimodal and Design2Code cover visual software tasks (Yang et al., 2025; Si et al., 2025); WebArena, OSWorld, and Agents' Last Exam assess web, desktop, and professional workflows (Zhou et al., 2024; Xie et al., 2024; Sun et al., 2026b).

Agents in game development. Prior work explores game-content generation (Sudhakaran et al., 2023; Todd et al., 2024), collaborative development (Chen et al., 2025; Earle et al., 2025), and engine-based world creation (De La Torre et al., 2024; Yin et al., 2026). GameDevBench, GameEngineBench, and JAMER evaluate game development and project completion (Chi et al., 2026; La et al., 2026; Sun et al., 2026a), while GameCraft-Bench and GameXpert-Bench study game generation and refinement (Luo et al., 2026; Chen et al., 2026). AWoMo additionally studies cross-engine asset transfer (Zhou et al., 2026a). Table 1 compares these settings with AGENT'S GAMEDEV EXAM, whose focus is integrated feature development within existing projects and runnable application development from assets and multimodal requirements, spanning code and non-code authoring.

3 TASK DESIGN AND CONSTRUCTION

3.1 TWO COMPLEMENTARY TASK TRACKS

AGENT'S GAMEDEV EXAM evaluates game development at two levels, both grounded in complex real-world games. The tracks differ in the context supplied to the agent and the scope of the required deliverable, rather than simply representing two difficulty levels. Figure 2 summarizes the two task-construction workflows and their iterative expert-review loops.

Component-level development. The agent receives an existing game project with a target feature removed and a specification of the missing functionality. It must understand the surrounding implementation, select a suitable design, and integrate the requested feature while preserving specified existing behaviors. This track tests project comprehension and feature delivery under existing engineering constraints.

End2end-level development. The agent receives the assets extracted from an existing game, a detailed functional specification, and reference screenshots included in the prompt. It must organize these materials and implement a runnable game application that satisfies the requirements. This track tests the translation of multimodal requirements into an application architecture, the implementation of multiple interacting features, and their integration into a complete playable application.

3.2 Task Scope and Selection

We first collect existing game projects with multiple interacting gameplay systems. Game-development experts design component-level tasks according to each project's genre and existing content. End2end-level tasks use the source game's extracted assets and specify the functionality to implement at application scale. Within each genre, we prioritize recurring feature families, such as locomotion in RPGs, weapon switching in shooters, and interactions among AI-controlled units in RTS games, rather than mechanics specific to a single title. Agents remain free to choose implementations suited to each project.

The unit of evaluation is an integrated feature or a runnable game application, according to the track, not an isolated code fragment. These tasks are constructed from real games rather than mined from historical issue tickets. Neither track requires generating original artwork or reproducing the original implementation. End2end-level completeness is bounded by the supplied requirements, not by every undeclared capability of the source game.

3.3 Component-Level Task Construction

LLM-assisted feature removal. After experts define a task, an LLM removes the target feature's complete implementation from an independent working copy, preserving the source project and pre-existing data. Removal covers target-specific code, state, calls, bindings, and scene configuration, while retaining shared infrastructure needed by other features. All project comments are removed from textual code and supported visual scripting graphs. Non-built-in source assets, including audio and animations, are staged in `assets/` for the agent to import and integrate; built-in assets are listed by name only. Target-specific asset configurations and connections are removed to prevent simply re-enabling dormant functionality.

Expert verification and packaging. Experts inspect code and visual graphs to confirm complete target removal and project-wide comment removal, then run the modified copy in its engine. Retained features must pass regression checks against the pre-removal baseline, and the target behavior must remain unavailable in an otherwise runnable project. Release requires source preservation, structural review, and both runtime checks; unexecuted checks do not count as passes. Appendix B details the protocol, and Appendix C specifies prompt and submission formats.

3.4 End2end-Level Task Construction

For End2end-level tasks, we extract the source game's assets into a separate collection rather than providing a project with only one feature removed. We then write a detailed description of the functionality the agent must implement and include reference screenshots in the prompt. The construction output is therefore an asset collection paired with textual and visual requirements, from which the agent must deliver a game application.

The functional description states what the application must do; the screenshots provide visual references for its intended presentation. These images are agent-visible inputs, not hidden evaluation materials. They complement rather than replace the description of game rules and interactions. The agent must decide how to organize the assets, assemble scenes and objects, implement the requested mechanics, and connect them into a working application. This tests End2end-level design and integration rather than reconstruction of the original source code or the creation of a static image resembling a screenshot.

Appendix B.2 specifies the asset-preparation and input-validation protocol in a construction-only prompt; Appendix C.2 gives the corresponding agent-facing task template.

3.5 Task Instructions and Deliverables

Both tracks specify required behaviors and interactions without prescribing implementation steps, files to edit, or the source game's architecture. Component-level tasks provide a reviewed starting project and identify behaviors to preserve, but omit reference images: agents must infer gameplay and system context from the retained code, assets, and scenes. End2end-level tasks instead provide



Figure 2: Two expert-reviewed task-construction workflows from real-world games. For component-level tasks, experts define a genre-common feature, an LLM removes its complete implementation while retaining shared dependencies, and experts inspect the modified project. Residual implementation triggers further removal and review. For End2end-level tasks, extracted assets, functional requirements, and agent-visible reference screenshots form the task inputs. Game-development experts review these materials and request revisions before approval. Both feedback loops repeat as needed; only approved inputs become task packages.

extracted assets, functional requirements, and reference screenshots. In both tracks, agents must translate requirements into engineering decisions and choose designs suited to the available context.

Agents submit a complete game project and a list of added or modified files. Within the available engine environment, they may edit any project file and choose their implementation language and workflow, including textual code, supported visual scripting, and asset or scene configuration. Evaluation assesses delivered functionality and specified protected behaviors, rather than conformity to the original implementation.

4 Evaluation Pipeline

Evaluation in Agent's GameDev Exam combines *functional tests* of requested behavior with *regression tests* of existing behavior that must remain intact. Both test types use deterministic checks and LLM-based video assessment, according to the property being evaluated. Ground-truth (GT) projects provide executable references for the expected outcomes under the prescribed test scenarios. We assess the delivered game project, rather than agreement with the reference implementation. Figure 3 summarizes test preparation, paired execution, and scoring.

4.1 Expert-Guided Test Design

For each task package, game-development experts and LLMs jointly analyze the requirements, project structure, and technical dependencies. They develop and compare implementation approaches, and experts select a reference design they consider best suited to the task's engineering constraints. This analysis identifies the functional checkpoints needed to deliver the requested behavior, including required asset relationships, animation behavior, gameplay logic, and system integration. The reference design guides test construction; it is neither an implementation recipe supplied to the evaluated agent nor a requirement to reproduce the GT architecture.

Experts then design functional and regression tests against the task specification. Each test fixes its starting conditions, operation sequence, expected outcomes, and acceptance criteria before submissions are evaluated. Running these scenarios in the GT project establishes reference behavior and allows checks to be organized by feature and reused across tasks with the same retained behavior and prerequisites. GT references cover the specified scenarios, not every possible execution. Requirements absent from the task are not introduced merely because they occur in the source game.

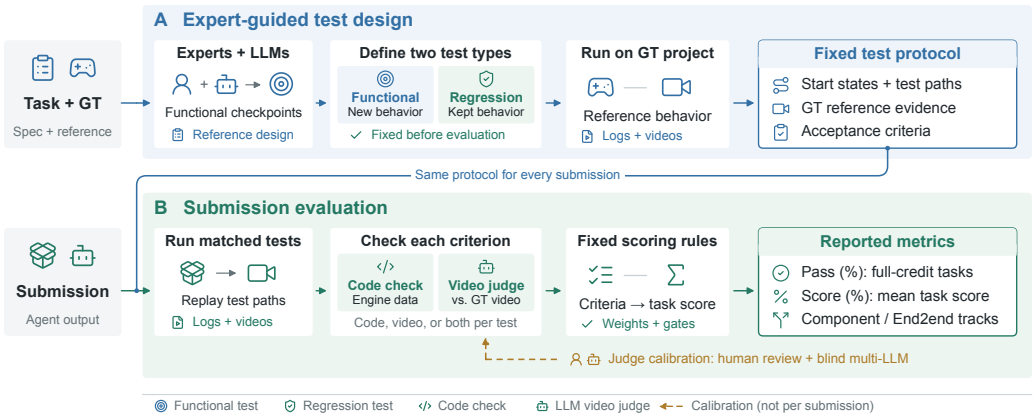


Figure 3: GT-referenced evaluation pipeline. (A) For each task, experts and LLMs identify functional checkpoints and a reference design. Experts define functional tests of requested behavior and applicable regression tests of retained behavior, fixing start states, test paths, and acceptance criteria before any submission is evaluated. Running these tests on the GT project records reference behavior. (B) Every submission runs under the same protocol. Each criterion is decided by deterministic checks of project and engine data, a GT-referenced LLM video judgment with a fixed question, or both. Fixed scoring rules combine criterion outcomes into task scores, reported as Pass and Score per track. Human review and blind multi-LLM comparisons calibrate the video judge (gold dashed arrow); calibration is not a per-submission step.

Appendix E describes the evaluation evidence and execution-failure rules; environment and agent configurations are documented in Appendix D and Section 5.1.

4.2 Functional and Regression Tests

Functional tests. These tests assess whether the requested functionality works in its execution context. For component-level tasks, they cover the expert-defined functional checkpoints and their integration with the retained project. For End2end-level tasks, they cover the requested mechanics and interactions among them in a runnable application. A component's presence or an isolated successful call does not establish that the corresponding gameplay behavior works end to end.

Regression tests. These tests check that implementing the target functionality has not broken specified existing behaviors. Experts identify the protected behaviors from the task's starting project and use the GT to establish their expected outcomes. Feature-specific checks can consequently serve as reusable regression tests in other tasks where those features remain present. Features intentionally removed from a task's input are excluded from that task's regression requirements. End2end-level tasks without inherited working behavior have no applicable regression group; interactions among newly implemented systems remain functional tests, and absent regression tests do not yield automatic credit.

4.3 Deterministic and Video-Based Evaluation

Deterministic checks. We use code to evaluate requirements that can be expressed as measurable properties. For UE5 tasks, project data and logged engine observations provide evidence such as asset relationships, relative transforms, skeletal pose data, and velocities. Assertions compare these measurements with the task's specified relationships, conditions, and numerical tolerances. Runtime tests exercise the relevant gameplay interactions rather than treating a submission's self-reported success flag as proof. Checks target required outcomes, not incidental GT class names or implementation choices; numerical acceptance does not require exact reproduction of a single GT trajectory.

GT-referenced video checks. For requirements not reliably decided by the deterministic checks, we record gameplay along predefined test paths in both the submission and GT projects. A multimodal LLM compares the resulting evidence under the same task-specific test protocol, using a description of the target function to delimit the comparison. Each judgment asks whether the specified action or behavior is consistent between the two videos and returns only `Yes` or `No`, rather than providing an open-ended impression of game quality. The shared prompt template is provided in Appendix E.1. Consistency with GT means satisfying the required behavior and presentation, not pixel-identical footage or identical internal implementations. The comparison separates required agreement in target behavior from permissible visual or spatial variation: changes only to nonessential details should not change the verdict. These video checks can assess either target functionality or protected existing behavior.

The questions and decision rules are fixed across submissions; this does not imply that LLM inference itself is deterministic. We calibrate video judgments against human assessments and conduct blind assessments with multiple LLMs. Human calibration checks agreement with expert decisions, while cross-LLM comparisons assess consistency. The per-criterion judgments and deterministic outcomes are combined by fixed scoring rules.

4.4 Metrics and Execution Protocol

We follow ALE's distinction between full passes and fine-grained task scores (Sun et al., 2026b). For a fixed model–harness configuration and evaluation repetition r, let $s_{t,r} \in [0, 1]$ be the normalized score returned by task t's evaluation protocol. A score of 1 denotes full credit under that protocol. For the task set $\mathcal{T}$ in one track, we report

$$\mathrm{Pass}_r(\%) = \frac{100}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \mathbf{1}[s_{t,r} = 1], \qquad \mathrm{Score}_r(\%) = \frac{100}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} s_{t,r}. \tag{1}$$

Pass is the percentage of tasks receiving full credit, assessed before display rounding; Score is the mean task score expressed as a percentage. We report both metrics separately for component-level and End2end-level tasks. Any summary across repetitions or tracks must state its weighting rather than silently pooling runs or tasks.

Each task's protocol fixes score normalization, weights or gates, and full-credit conditions before evaluation. Functional and applicable regression outcomes contribute through this protocol, regardless of whether they are determined by code or video assessment; they are not separate headline metrics.

Submissions that fail to compile or cannot be opened in the target engine receive a task score of 0. The MCP evaluation pipeline ensures valid recordings for video assessment. Under GT-validated test protocols, the absence of a required behavior or state in a valid observation constitutes a failed check rather than missing evidence. Applicability follows the task specification, and failures do not remove tasks or checks from their scoring denominators. Appendix E.2 details these execution and scoring rules.

5 Experiments

Our experiments examine overall task delivery and failure modes, using the tasks in Section 3 and the delivery criteria in Section 4. Model and harness configurations are listed in Table 4.

5.1 Experimental Setup

Task packaging and time budgets. We package the task bundles and their evaluation protocols in the Agents' Last Exam (ALE) format (Sun et al., 2026b). Each component-level run starts from a feature-removed project and its functional specification, while each End2end-level run starts from the supplied assets, functional requirements, and reference screenshots (Section 3). The execution time limit for each task is 10 hours in the component-level track and 48 hours in the End2end-level track. Each model–harness configuration is evaluated three times per task in both tracks.

Table 2: Model performance on AGENT'S GAMEDEV EXAM, with the component-level and End2end-level tracks reported separately. Parenthesized budgets are per-task time limits. Pass is the full-pass rate and Score is the mean normalized task score, both in percent (higher is better). 🪙, 🕒, and Tok. denote the mean API cost (USD), run time (hours), and token use (millions) per task run, respectively. Each row uses the model–harness configuration in Table 4.

Model	Component-level (10 h)					End2end-level (48 h)				
	Pass (%)	Score (%)	🪙 ($)	🕒 (h)	Tok. (M)	Pass (%)	Score (%)	🪙 ($)	🕒 (h)	Tok. (M)
GPT-6 Astra	$0.00^{\pm 0.00}$	$21.14^{\pm 0.03}$	9.61	1.00	7.17	$0.00^{\pm 0.00}$	$16.59^{\pm 0.79}$	39.46	1.56	29.62
Claude Fable 5.1	$14.23^{\pm 0.42}$	$31.64^{\pm 0.03}$	28.16	1.35	35.57	$0.00^{\pm 0.00}$	$18.03^{\pm 0.48}$	88.00	19.35	96.82
Grok 4.6	$4.29^{\pm 0.67}$	$12.14^{\pm 0.37}$	9.26	4.39	12.30	$0.00^{\pm 0.00}$	$5.70^{\pm 0.84}$	17.32	4.15	30.24
Kimi K3	$0.00^{\pm 0.00}$	$6.07^{\pm 0.05}$	27.98	2.38	60.58	$0.00^{\pm 0.00}$	$7.73^{\pm 0.41}$	117.96	36.15	345.14
DeepSeek-V4.1-Flash	$0.00^{\pm 0.00}$	$0.94^{\pm 0.08}$	32.67	1.13	30.15	$0.00^{\pm 0.00}$	$6.98^{\pm 0.31}$	22.29	43.80	219.50
Hunyuan Hy4 preview	$0.00^{\pm 0.00}$	$7.46^{\pm 0.18}$	2.86	3.33	40.78	$0.00^{\pm 0.00}$	$2.67^{\pm 0.50}$	20.95	34.72	43.62
GLM-5.3-Flash	$0.00^{\pm 0.00}$	$10.36^{\pm 0.17}$	78.14	3.04	134.98	$0.00^{\pm 0.00}$	$1.48^{\pm 0.67}$	98.09	27.80	309.27
Qwen3.8-Max-0902	$0.00^{\pm 0.00}$	$8.46^{\pm 0.33}$	49.87	3.85	82.80	$0.00^{\pm 0.00}$	$1.99^{\pm 0.14}$	144.22	9.70	46.96
Muse Spark 1.3	$0.00^{\pm 0.00}$	$3.03^{\pm 0.45}$	3.42	2.11	16.31	$0.00^{\pm 0.00}$	$1.70^{\pm 0.13}$	107.51	20.85	32.22

Platform and engine versions. All experiments run on Windows 10 Professional. The engine versions are Unreal Engine 5.5.4, Unity 6000.3.14f1, Godot 4.7.2, and RPG Maker MZ 1.10.0.

Agent harnesses and engine tools. For each model, we preferentially use its associated agent harness when available; otherwise, we use Codex as the shared fallback harness. Table 4 lists the model–harness configurations. All game-engine projects include engine-specific tool interfaces, and agents may choose whether to use these interfaces to modify the project. For Unity, we provide Unity CLI[1]; for Unreal Engine 5 and Godot, we provide Model Context Protocol (MCP) interfaces via Unreal-MCP[2] and godot-mcp[3], respectively.

Image-description assistance. For models without native visual capabilities, we use GPT-5.5 as an image-to-text interface. Its role is restricted to providing textual descriptions of image contents; it does not perform task planning, recommend actions, generate code, or modify projects. The evaluated model remains responsible for subsequent task reasoning and action selection based on these descriptions. Our comparisons therefore concern model–harness configurations with their available tools and, where needed, image-description assistance, rather than isolated model capabilities.

5.2 RESULTS

Benchmark performance. Table 2 shows that complete delivery remains rare across all nine configurations. Claude Fable 5.1 leads the component-level track with a 31.64% Score and 14.23% Pass, leaving most tasks unresolved even for the strongest configuration. GPT-6 Astra ranks second in Score (21.14%) but records no full passes, whereas Grok 4.6 achieves 4.29% Pass despite a lower Score (12.14%). Thus, higher average partial credit does not necessarily translate into more completed tasks. The remaining six configurations score 0.94–10.36%, with no full passes. On the End2end-level track, none achieves a full pass under the 48-hour budget. Together, these results expose substantial headroom in both restoring individual features and delivering complete games.

5.3 FAILURE ANALYSIS

We examine implementation issues and conditional risks in UE5 component-level submissions (Figure 4). Prose percentages give shares of all categorized issues; heatmap rows show model-family shares within each type. Neither measures task failure rates.

[1] https://docs.unity.com/en-us/unity-cli/unity-cli
[2] https://github.com/IvanMurzak/Unreal-MCP
[3] https://github.com/Coding-Solo/godot-mcp



Figure 4: Failure mechanisms in UE5 component-level submissions. Columns denote grouped model families using their logos. Each row is normalized across families: a cell reports the family's share of issues within that mechanism. These percentages are not task failure rates.

Disconnected implementations. Incomplete execution paths (48.3%) and state-to-effect disconnects (18.3%) leave relevant components unconnected. A DeepSeek energy-collection implementation preserves idle rotation but omits the path from contact through collection-state updates to disappearance feedback. Muse implementations update activation flags without driving the intended mechanisms. These cases illustrate how individually plausible components can fail to deliver a feature when events and state changes do not propagate to the required gameplay effects.

Incorrect behavioral semantics. Data and object semantics (11.7%) include a Qwen door implementation that truncates interpolated progress to integers, potentially discarding small updates. Trigger and eligibility errors (11.7%) include a GLM platform that starts moving on overlap instead of the required deliberate interaction. Such mismatches require checks of numerical behavior and activation conditions, since the presence of a recognizable mechanic alone does not establish compliance with the specification.

Incomplete lifecycle and delivery handling. Lifecycle and safe-recovery issues (8.3%) include retained deduplication state that can block repeated GLM respawn requests and a Claude condition that can inhibit cancellation after control loss. Delivery and dependency issues (1.7%) include Grok native interfaces declared without implementations. Together, these cases highlight the need to verify repeated and interrupted interactions alongside dependency completeness, extending validation beyond an isolated use of the target feature.

6 Conclusion

We introduce Agent's GameDev Exam, a multi-engine benchmark for evaluating AI agents on component-level feature development and end-to-end game creation grounded in real game projects. We evaluate nine models across both tracks to assess their ability to translate requirements into working game functionality. Our failure analysis of UE5 component-level submissions identifies incomplete execution paths and disconnected state-driven effects as the dominant issues, revealing a gap between locally plausible implementations and fully integrated features. These findings highlight the need for future models to improve system-level reasoning, behavioral correctness, and runtime verification across complete interaction and recovery sequences.

Limitations

Our benchmark focuses on implementing specified game functionality using existing projects or supplied assets. It does not assess original artwork or audio creation, or open-ended art direction. In addition, experiments are conducted in a fixed Windows environment with specified engine versions; cross-platform deployment and engine-version migration remain outside the current evaluation scope.

AI USE STATEMENT

We used generative AI tools to assist with dataset construction, figure and chart preparation, and language editing for clarity and readability. AI-assisted dataset construction included feature removal and task preparation, with expert review and structural and in-engine validation as described in Section 3. LLMs also assisted with test design and video-based assessment, as detailed in Section 4. The authors take responsibility for the final manuscript, dataset, and reported results.

REFERENCES

Alibaba. Qwen3.8-Max model information, 2026. URL https://help.aliyun.com/zh/model-studio/qwen3-8-max. Official model documentation in Chinese; title translated. Accessed September 16, 2026.

Anthropic. Claude Fable 5.1, 2026. URL https://platform.claude.com/docs/en/models/fable-5-1/overview. Official model documentation. Accessed September 16, 2026.

Michiel Buijsman. The forces driving the games market to $213.9 billion. Newzoo, September 2026. URL https://newzoo.com/articles/executive-summary-ggmr-2026-free-edition. Published September 15, 2026. Accessed September 16, 2026.

Dake Chen, Haoyang Zhang, Hanbin Wang, Yunhao Huo, Yuzhao Li, and Junjie Wang. GameGPT: Multi-agent collaborative framework for game development. arXiv preprint arXiv:2310.08067, 2025. URL https://arxiv.org/abs/2310.08067.

Kun Chen, Haorong Hong, Peizhong Gao, Jianfeng Lin, Tongxu Luo, Yuxuan Xie, Chenxu Liu, Jieling He, Zhongyuan Liu, and Zeno Zeng. GameXpert-Bench: How far are coding agents from expert game development? arXiv preprint arXiv:2608.21833, 2026. URL https://arxiv.org/abs/2608.21833.

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. URL https://arxiv.org/abs/2602.11103.

Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. LLMR: Real-time prompting of interactive worlds using large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pp. 1–22. ACM, 2024. doi: 10.1145/3613904.3642579. URL https://doi.org/10.1145/3613904.3642579.

DeepSeek. Introducing DeepSeek-V4.1-Flash: Smarter, faster, more efficient, 2026. URL https://www.deepseek.com/en/news/deepseek-v4-1-flash/. Official model announcement. Accessed September 16, 2026.

Sam Earle, Samyak Parajuli, and Andrzej Banburski-Fahey. DreamGarden: A designer assistant for growing games from a single prompt. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pp. 1–19. ACM, 2025. doi: 10.1145/3706598.3714233. URL https://doi.org/10.1145/3706598.3714233.

Beth Elderkin. GDC 2026 State of the Game Industry reveals impact of layoffs, generative AI, and more. GDC Festival of Gaming, 2026. URL https://gdconf.com/article/gdc-2026-state-of-the-game-industry-reveals-impact-of-layoffs-generative-ai-and-more/. Official survey summary. Accessed September 16, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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

Keyu Li, Junhao Shi, Yang Xiao, Mohan Jiang, Jie Sun, Yunze Wu, Dayuan Fu, Shijie Xia, Xiaojie Cai, Tianze Xu, Weiye Si, Wenjie Li, Dequan Wang, and Pengfei Liu. AgencyBench: Benchmarking the frontiers of autonomous agents in 1M-token real-world contexts, 2026. URL https://arxiv.org/abs/2601.11044.

Hugh Xuechen Liu and Kıvanç Tatar. Mage: Multi-axis evaluation of LLM-generated executable game scenes beyond compile-pass rate, 2026. URL https://arxiv.org/abs/2605.07342.

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

Meta. Introducing Muse Spark 1.3, 2026. URL https://research.meta.ai/blog/introducing-muse-spark-1-3. Official model announcement. Accessed September 16, 2026.

Moonshot AI. Kimi K3: A new frontier of intelligence, 2026. URL https://www.kimi.com/news/kimi-k3. Official model announcement in Chinese; title translated. Accessed September 16, 2026.

OpenAI. GPT-6 Astra model, 2026. URL https://developers.openai.com/api/docs/models/gpt-6-astra. Official model documentation. Accessed September 16, 2026.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3956–3974. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.199. URL https://aclanthology.org/2025.naacl-long.199/.

SpaceXAI. Introducing Grok 4.6, 2026. URL https://x.ai/news/grok-4-6. Official model announcement. Accessed September 16, 2026.

Shyam Sudhakaran, Miguel González-Duque, Claire Glanois, Matthias Freiberger, Elias Najarro, and Sebastian Risi. MarioGPT: Open-Ended Text2Level Generation through Large Language Models. arXiv preprint arXiv:2302.05981, 2023. URL https://arxiv.org/abs/2302.05981.

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, 2026a. URL https://arxiv.org/abs/2606.19830.

Yiyou Sun, Xinyang Han, Weichen Zhang, et al. Agents' last exam, 2026b. URL https://arxiv.org/abs/2606.05405.

Tencent. Tencent Hunyuan, 2026. URL https://cloud.tencent.com/product/tclm. Official model overview. Accessed September 16, 2026.

Graham Todd, Alexander Padula, Matthew Stephenson, Éric Piette, Dennis J. N. J. Soemers, and Julian Togelius. GAVEL: Generating Games Via Evolution and Language Models. arXiv preprint arXiv:2407.09388, 2024. URL https://arxiv.org/abs/2407.09388.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. arXiv preprint arXiv:2404.07972, 2024. URL https://arxiv.org/abs/2404.07972.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

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. URL https://arxiv.org/abs/2603.07106.

Z.ai. GLM-5.3-Flash, 2026. URL https://docs.bigmodel.cn/cn/guide/models/vlm/glm-5.3-flash. Official model documentation. Accessed September 16, 2026.

Pengfei Zhou, Hexin Wang, Zhengfeiyang Zhang, Yixing Ma, Zhenglin Wan, Kaipeng Zhang, Wangbo Zhao, and Yang You. Agentic game development as a verifiable trajectory data engine for scaling world models, 2026a. URL https://arxiv.org/abs/2608.25518.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. arXiv preprint arXiv:2602.10975, 2026b. URL https://arxiv.org/abs/2602.10975.

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. arXiv preprint arXiv:2307.13854, 2024. URL https://arxiv.org/abs/2307.13854.

A Benchmark Inventory and Coverage

A.1 Source Conditions

Game projects are collected from Fab[4], itch.io[5], and GitHub[6], including both paid and free-of-charge projects. In addition to open-source game projects, the source pool includes game project data purchased from commercial data vendors. We retain a provenance record linking each task to its original game source. These source references are not disclosed to agents during evaluation, to reduce the risk of agents locating and consulting the original implementation as a solution reference.

A.2 Inventory and Coverage

The current inventory comprises 448 tasks drawn from 55 projects/packages. Table 3 reports the counts, normalized terminology, and game-design scope for all 14 tracked coverage groups. All 14 groups have nonzero counts, including racing, sports/rhythm, and XR; the XR group comprises 2 projects/packages and 9 tasks.

Table 3: Project counts and task counts, normalized terminology, and game-design scope for the 14 tracked coverage/application groups. All 14 groups are represented in the corrected inventory.

Source group	Normalized terminology	Projects / packages	Tasks	Game-design scope and distinctions
Action RPG / third-person action	Action role-playing and third-person action games	9	67	Action RPGs combine real-time action with role-playing systems, such as character progression. Third-person specifies viewpoint and does not, by itself, imply role-playing mechanics.
FPS / TPS	First-person and third-person shooters	8	78	Shooting defines the core interaction; viewpoint distinguishes FPS from TPS. Relevant systems include aiming, firing, weapon handling, and hit feedback.
Multiplayer / co-op / party	Competitive and cooperative multiplayer; party games	4	24	Distinguish competitive or cooperative player relationships from the party-game format. Local and online play describe participation settings rather than standalone gameplay genres.
Platform / mechanism / physics puzzle	Platforming and mechanism- or physics-based puzzles	7	44	Covers traversal challenges, mechanism operation, and reasoning about physical interactions. Platforming and puzzle solving may combine; mechanisms are recorded as interaction elements rather than a separate genre.
Management / farming / building	Management and farming simulations; building systems	4	39	Covers resource allocation, production, farming, construction, and their feedback loops. Apply RTS terminology only when the specific gameplay warrants real-time strategic classification.

Continued on next page

[4] https://www.fab.com/
[5] https://itch.io/
[6] https://github.com/

Table 3: Project and task counts, terminology, and game-design scope (continued).

Source group	Normalized terminology	Projects / packages	Tasks	Game-design scope and distinctions
Open world / city simulation	Open-world games and urban simulation	2	24	Open-world describes world structure and exploration freedom; urban simulation concerns city systems or activities. Traffic, pedestrians, and driving interactions do not necessarily imply city-building gameplay.
Survival / gathering / crafting	Survival gameplay with gathering and crafting systems	4	33	Survival centers on managing threats, needs, or resource scarcity. Gathering and crafting describe resource acquisition and transformation and may also appear outside survival games.
2D side-scroller / top-down	2D side-scrolling and top-down presentation	3	42	Distinguish dimensional presentation, camera scrolling, and viewpoint. These conventions can support action, shooting, strategy, or puzzle gameplay rather than defining a single genre.
Strategy / RTS / tower defense	Real-time strategy and tower-defense games	2	12	Strategy provides the broader category. RTS emphasizes real-time command and resource decisions; tower defense emphasizes defensive placement and responses to incoming enemies or waves.
Racing / vehicle competition	Racing games and vehicle-based competition	1	23	Distinguish racing objectives, such as time or finishing order, from other vehicle-based competitive objectives. Vehicle handling is a shared control system, not a sufficient genre definition.
Turn-based / card / roguelike	Turn-based games, card-based games, and roguelike games	4	24	Turn structure, card-based mechanics, and roguelike conventions describe different aspects of play. They may coexist but are not interchangeable; each label should be assigned separately.
Narrative / exploration / puzzle	Narrative adventure, exploration, and puzzle gameplay	4	26	Distinguish narrative progression, spatial discovery, and problem solving. Dialogue, quests, clues, and environmental interaction can connect these activities; cinematics alone do not define the category.
Sports / rhythm	Sports games and rhythm games	1	3	Sports games organize play around sporting rules and skills; rhythm games emphasize synchronization with rhythmic cues. They remain distinct gameplay families within the shared source group.
VR / AR / XR	Extended reality (XR), including virtual reality (VR) and augmented reality (AR)	2	9	XR describes presentation and interaction modalities encompassing VR and AR. Spatial tracking and interaction should be characterized separately from the underlying gameplay genre.
Total		**55**	**448**	

B Task Construction and Quality Control

B.1 Component-Level Construction

Component-level construction operates on an independent working copy; source projects and pre-existing data must remain unchanged. Packages retain the surrounding game project while removing the target feature's implementation chain (Section 3.3). Removal covers target-specific code at every layer, including lower-level helpers and services, call sites, callbacks, state, asset references, bindings, and scene configuration. All project comments are removed from both textual code and visual-programming representations, including comments unrelated to the target feature. Shared dependencies required by other behaviors remain available. Non-built-in source assets are staged in `assets/` for the solving agent to import and assemble, including required audio and animations; built-in assets are identified by name only. Expert inspection verifies implementation absence rather than a disabled entry point. Release also requires running the modified project copy in its actual engine to verify retained behaviors and confirm that the target behavior cannot be completed. The construction-only template below expresses the removal boundary; it is not part of the evaluated agent's prompt. Angle-bracketed fields are filled from the expert task definition. This is a reusable specification, not a record of a completed removal run.

Construction-only prompt: remove a target component

Source project (read-only): `<SOURCE-PROJECT>`
Independent working copy: `<WORKING-COPY-PATH>`
Target behavior: `<TARGET-BEHAVIOR>`
Behaviors to retain: `<PROTECTED-BEHAVIORS>`

Create a starting project in which the target behavior's complete implementation is absent while the surrounding game remains available for further development.

Source preservation and copy-only operation. Do not edit, overwrite, move, or delete the original project or any pre-existing data. Create an independent working copy before making changes; perform all removal, comment cleanup, asset preparation, and engine tests on that copy and in newly created output directories only. Copy source assets rather than moving them or creating writable links back to the originals. Isolate test saves and generated state so engine runs cannot overwrite existing user data. Before removal, run the unmodified copy to record baseline behavior for the target and retained features; verify at handoff that the source project and original data remain unchanged.

Full-chain implementation removal. Trace the target-specific implementation across textual code, supported visual graphs, functions, lower-level helpers and services, call sites, callbacks, variables and state, asset references, animation bindings and transitions, audio playback, and scene or object configuration. Delete every target-specific implementation at every layer, not only the code or graph nodes that trigger it. Remove disconnected or unreachable target-specific functions and graph fragments as well; disabling, renaming, or disconnecting an input binding, event, or other entry point is insufficient. Retain shared infrastructure needed by the protected behaviors, but remove its target-specific branches, calls, state, and connections. Do not remove a shared dependency solely because the target component also uses it.

Project-wide comment removal. Delete all comments throughout the entire project, not only those related to the target feature. This includes line, block, and documentation comments in textual code, as well as comment boxes, node comments, and explanatory annotations in visual-programming graphs.

Non-built-in source assets. Save reusable source assets needed by the target feature in the task package's `assets/` folder, where redistribution is permitted. Include models, textures, material source data, audio (sound effects and music), and all required animation clips as importable resources. Remove target-specific imported objects, assembled prefabs or scenes, material setups, animation controllers and bindings, and audio-playback configuration from the starting project. The solving agent must import the supplied resources, assemble objects, and reconstruct the full task-required asset setup, including sound and animation integration, rather than reconnect ready-made configurations. Preserve shared asset uses needed by protected behaviors, but do not place removed scripts, implementation graphs, or ready-made target compositions in `assets/`.

Engine built-in assets. Remove their target-specific calls and references from textual code and visual graphs, together with the associated bindings and configured use. Do not copy built-in assets into `assets/`; retain only their names in an agent-visible resource list, without original invocation code, graph wiring, or setup instructions.

Construction-only prompt: remove a target component (continued)

Required validation in the actual project. After removal, save and reopen the modified working copy in its actual game engine, compile or build where applicable, and run the real project in the relevant scenes or maps. Source inspection, mocked tests, or successful compilation alone are insufficient. Perform positive regression checks against the baseline for the retained non-target features, including their required gameplay, visual, audio, and animation behavior. Then attempt the target behavior through its original and alternative inputs or events, including applicable automatic or delayed callbacks; verify that the actions were exercised and that the target's completion criteria cannot be met. A launch failure, missing shared data, or broken retained feature is not a valid negative test. Fix any regression only in the working copy and repeat both sets of runtime checks. If required checks cannot be executed, record them as not run, leave the package unverified, and do not release it as passing.

Construction handoff. Return the modified project, the `assets/` folder, the names-only built-in asset list, and a construction-review report listing changed locations, removed target-specific elements, the project-wide comment audit, staged source assets, retained shared dependencies, and unresolved boundary questions. Include source-preservation evidence and runtime-check records: the tested working-copy path, engine version, scenes, actions, expected and observed outcomes, evidence locations, and per-check `PASS`, `FAIL`, or `NOT_RUN` status. Keep the original project, baseline evidence, and construction-review report separate from agent-facing inputs. Do not claim expert approval or runtime correctness from removal alone; release the task package only after source preservation is confirmed, the separate expert review approves full-chain removal, and both positive regression and target-negative runtime checks pass.

B.2 End2end-Level Construction

End2end-level packages combine extracted assets, functional requirements, and agent-visible reference screenshots (Section 3.4). Unlike component-level construction, this workflow does not retain the surrounding game's implementation. It prepares resources and an application-scale specification from which the agent must build a playable project. Any supplied scaffold is declared explicitly rather than treated as part of the asset collection. The construction-only template below specifies this preparation protocol, not a completed construction run or the evaluated agent's instructions. Its fields are supplied by task authors; the resulting agent-facing prompt follows Appendix C.2.

Construction-only prompt: prepare an End2end-level task

Source project (read-only): `<SOURCE-PROJECT>`
Independent working copy: `<WORKING-COPY-PATH>`
Task package output: `<NEW-PACKAGE-ROOT>`
Task ID / title: `<TASK-ID>` / `<TASK-TITLE>`
Engine: `<ENGINE-AND-VERSION>`
Expert-defined application scope: `<APPLICATION-SCOPE>`

Prepare an asset collection, a functional specification, and reference screenshots for an End2end-level game-development task. Do not deliver the source game's implementation or solve the task inside the input package.

Source preservation and copy-only operation. Leave the original project and pre-existing data unchanged. Create an independent working copy before opening the project in its engine; use that copy for inspection, asset export, and reference capture. Write package contents and temporary validation projects only to new directories, copy rather than move source resources, and isolate generated saves and runtime state. Verify source preservation at handoff.

Asset extraction and manifest. Stage the reusable resources required by the application scope in `assets/`, where redistribution is permitted. Include applicable models, textures, material source data, sprites, tiles, interface graphics, fonts, animation clips, sound effects, and music, together with dependencies needed to import or load them. Record each resource's package-relative path, file format, semantic role, and dependencies in an agent-visible manifest; use formats supported by the declared engine. Identify required engine built-in assets by name only rather than copying them into the package. Do not require the agent to recover missing resources from the private source project.

Implementation boundary and optional scaffold. Exclude source-game scripts, visual-programming logic, and configured gameplay scenes, prefabs, controllers, or bindings that already implement a required system. If reusable asset data is bundled with gameplay logic, separate it or flag the resource for expert review before release. Supply no starting project unless a scaffold is explicitly declared. For a supplied scaffold, list its retained code, configuration, dependencies, and entry point; restrict it to engine

Construction-only prompt: prepare an End2end-level task (continued)

setup rather than implementation of the requested gameplay. Keep original project links, implementation notes, reference solutions, and private evaluation materials outside all agent-visible resources.

Functional specification. Write numbered requirements describing what the application must do, not how to implement it. Cover the initial playable state, controls and responses, core gameplay loop, progression or resource changes, interactions among systems, and applicable completion, failure, and restart conditions. Specify required interface feedback, animation, audio, numerical constraints, and edge cases where they belong to the task scope. Give each requirement observable outcomes and operating conditions without naming implementation files, prescribing an architecture, or supplying editing steps. Do not silently add every feature of the source game; flag ambiguous or conflicting requirements for expert resolution.

Reference screenshots. Capture the relevant scenes and gameplay states from the working copy running in the actual engine. Provide an ordered image list with stable image IDs, package-relative paths, view or state descriptions, and the public visual requirements associated with each image. Exclude editor overlays unless they are part of the stated task, and do not expose private code or source-project locations in the images or captions. Screenshots specify presentation; describe game rules and interactions in text rather than expecting the agent to infer them from a still image.

Package validation and expert review. Check the package without access to the source project: resolve all resource paths, test required asset import or loading in a disposable project using the declared engine, and open and run any supplied scaffold. Verify that requirement IDs and image IDs agree, attachments are present and ordered, and no unresolved template fields or hidden implementation dependencies remain. Have experts review scope, resource sufficiency, text–image consistency, and the implementation boundary; revise the package when a check fails. Record checks as `PASS`, `FAIL`, or `NOT_RUN`, with actions, observed outcomes, and evidence locations. Asset import and scaffold startup do not establish completion of the future agent's game; do not report them as gameplay success.

Construction handoff. Return the substituted task prompt, asset collection and manifest, built-in asset list, ordered screenshots and captions, and the declared scaffold or an explicit statement that none is supplied. Keep the source-preservation record, review report, and validation evidence separate from agent inputs.

C Task Prompts and Deliverables

C.1 Template Conventions and Shared Instructions

The templates below instantiate the task-facing contract in Section 3.5. Their separation of requests, resources, deliverables, and evaluation records follows the task-specification and task-card organization in ALE (Sun et al., 2026b, Appendices B.3 and C.2), adapted to our two game-development tracks. They are parameterized templates, not verbatim transcripts from completed evaluations. Replace every `<FIELD>` before use; do not expose unresolved placeholders to an agent. Task authors supply observable requirements and constraints, not implementation steps, files to edit, or a reference architecture.

The shared block specifies benchmark-level system instructions. Harness-specific identity, tool descriptions, runtime metadata, and policies remain part of the configuration in Appendix D. The final assembled instructions, task text, and ordered multimodal attachments should be archived with each run.

Shared system-instruction template

You are completing a game-development task in the supplied engine environment. Deliver a runnable game project that satisfies the stated requirements. Use the supplied project or assets and the tools permitted by the runtime policy. Choose the design, project organization, and implementation approach yourself; the task does not prescribe an editable-file whitelist or require a particular programming language.

Implement all requested behaviors and interactions. For a component-level task, also preserve every specified existing behavior. For an End2end-level task, integrate the required functionality into a complete playable application. Use provided screenshots as visual requirements where applicable, not as substitutes for working gameplay.

Use only agent-visible resources; do not obtain hidden reference projects, evaluation code, or private verification evidence. You are not required to launch the project, perform runtime checks, or submit runtime-test evidence; the submitted project is evaluated separately. Do not present an untested claim as a verified result.

Submit the complete project and a list of added or modified files. Include the project entry point and launch instructions.

C.2 Track-Specific Task Prompts

The shared instructions are followed by one track-specific block. The time limit is stated explicitly in each agent-facing block and matches Section 5.1; concrete engine versions, resource locations, and requirements are task-specific substitutions. For End2end-level tasks, keep requirement IDs consistent across the functional specification and image descriptions, and mark categories outside the declared scope as not applicable rather than adding undeclared features.

User-task template: Component-level

Task ID / title: `<TASK-ID>` / `<TASK-TITLE>`
Engine: `<ENGINE-AND-VERSION>`
Time limit: You have 10 hours to complete this task.
Starting project: `<PROJECT-ROOT>`
Source assets: `<ASSET-ROOT-OR-NONE>`
Built-in assets: `<BUILT-IN-ASSET-NAMES-OR-NONE>`

Context. The supplied game project is missing the following component: `<TARGET-BEHAVIOR>`. The surrounding game and reusable resources remain available.

Required functionality.
Observable behavior: `<OBSERVABLE-REQUIREMENTS>`
Triggers and responses: `<ACTIONS-AND-RESPONSES>`
State transitions, interactions, and edge cases: `<INTERACTIONS-AND-EDGE-CASES>`
Numerical or visual constraints: `<PUBLIC-CONSTRAINTS-OR-NONE>`

Behaviors to preserve. Keep the following existing behaviors correct under their listed operating conditions: `<PROTECTED-BEHAVIORS-AND-CONDITIONS>`.

User-task template: Component-level (continued)

Delivery. Implement and integrate the missing functionality in the supplied project. Submit the complete updated project at <SUBMISSION-ROOT>, together with the file-change list and handoff information requested by the shared instructions. A description of a solution without a working project is not a completed submission.

User-task template: End2end-level

Task ID / title: <TASK-ID> / <TASK-TITLE>
Engine: <ENGINE-AND-VERSION>
Time limit: You have 48 hours to complete this task.
Assets: <ASSET-ROOT>
Asset manifest: <ASSET-MANIFEST-PATH>
Built-in assets: <BUILT-IN-ASSET-NAMES-OR-NONE>
Starting scaffold: <SCAFFOLD-ROOT-OR-NONE>
Visual references: <ORDERED-IMAGE-LIST>

Objective. Build a runnable game application implementing <GAME-OBJECTIVE> from the supplied assets and requirements. The reference screenshots are agent-visible task inputs. The source game's gameplay implementation is not supplied as the solution. If no scaffold is declared, create a new project in the specified engine; otherwise use the declared scaffold and its documented setup.

Resource integration. Use the asset manifest to locate the supplied resources and their dependencies. Import or load the required assets, assemble the scenes and objects, and configure the materials, animation, and audio needed by the requirements. Copying resources into a folder without connecting them to the requested gameplay is insufficient. Any supplied scaffold contains only the setup declared here: <SCAFFOLD-CONTENTS-OR-NONE>.

Functional requirements. Implement the numbered requirements below as one connected playable workflow. Each requirement specifies its operating conditions, player actions or events, and observable outcomes.
Entry state and core gameplay loop: <ENTRY-AND-CORE-LOOP-REQUIREMENTS>
Controls and responses: <CONTROL-REQUIREMENTS>
Gameplay systems, state changes, and interactions:
<SYSTEM-AND-INTERACTION-REQUIREMENTS>
User interface, animation, and audio: <FEEDBACK-REQUIREMENTS>
Completion, failure, and restart: <OUTCOME-AND-RESTART-REQUIREMENTS>
Edge cases and public constraints: <EDGE-CASES-AND-CONSTRAINTS-OR-NONE>
Completeness is bounded by these requirements, not by undeclared features of the source game.

Visual requirements. For each image in the ordered attachment list, match the specified scene, gameplay state, and public visual constraints:
<IMAGE-IDS-STATES-AND-VISUAL-REQUIREMENTS>. The implemented application must produce these views during gameplay; do not substitute a screenshot for an interactive scene. Do not treat an image as a requirement to reproduce behavior that is not specified.

Delivery. Organize and integrate the supplied assets into a playable project at <SUBMISSION-ROOT>. Submit the complete project, file-change list, and handoff information requested by the shared instructions. A static image, video, or disconnected collection of scenes does not replace the required runnable application.

C.3 Input Bindings and Submission Record

The following roles define a logical package template rather than a fixed engine directory tree. Replace resource labels with runtime-resolved paths and preserve the engine's native project structure. Declaring an input root does not prescribe which project files an agent must edit. Project editing and any external-resource access follow the recorded runtime policy.

Resource role	Template contents and access boundary
Task record	Task ID, track, title, engine/version, time limit, runtime policy, and the fully substituted user-task text; agent-visible.
Starting project	Reviewed component-level project, including retained code, assets, and configuration; for End2end-level tasks, only an explicitly declared scaffold with its contents and setup documented, or none.
Asset collection	Importable source files in `assets/`, including required audio and animations; End2end-level manifests identify paths, formats, roles, and dependencies. No hidden source-game implementation.
Built-in asset list	Engine-provided asset names only, without original calls, graph wiring, or setup instructions; agent-visible where applicable.
Visual references	Ordered End2end-level input images, each with an image ID, resource path, view/state description, and associated public requirement.
Submission	Complete project and a handoff record; resolve the submission root without changing the native project layout.
Evaluation-private	GT project, tests, scoring configuration, and private evidence; provision outside the agent-visible resource set.

Attach images in the same order as their IDs appear in the task text, and archive both the image files and their captions. An illustrative screenshot added later to a paper's task card is not retroactively an agent input. If a harness uses image descriptions, record the original image, description, and auxiliary configuration separately as described in Appendix D.

Submission handoff template

Task ID and project root: `<TASK-ID>`; `<PROJECT-ROOT>`
Engine and entry point: `<ENGINE-AND-VERSION>`; `<ENTRY-POINT>`
Launch: `<OPEN-OR-RUN-INSTRUCTIONS>`
Added / modified files: `<PROJECT-RELATIVE-PATHS>`
Known limitations: `<UNRESOLVED-ISSUES-OR-NONE-REPORTED>`

The handoff records the submission and known limitations; it does not require runtime-check results or replace evaluator-side verification. These reporting fields support reproducibility; they do not introduce additional scored requirements beyond the task's declared evaluation protocol.

C.4 Task Examples: Agent Inputs and Scoring

The cards adapt ALE's format (Sun et al., 2026b, Appendix B.3), summarizing the task documents and metadata in `U-WEAPON-01-ale-dropin.zip` and `StackOBot_5.5_FromScratch_20260919.zip`. They separate **agent inputs**, required submissions, and **evaluator-side checks and scoring**. Both examples follow Appendix C.1: source-package requirements for solver-side runtime checks and evidence submission are removed from the adapted agent-facing documents. Agents submit the project and handoff record; evaluators collect check statuses, gameplay recordings, and build/startup logs as needed by the protocol. Component-level tasks have no image inputs; the End2end-level images below are supplied references, not solver captures. Evaluation-private tests, scoring configuration, and execution evidence are not agent inputs. The all-checks-fail scoring examples are illustrative, not measured model results.

Component-level example: inputs and scoring

Task ID: `U-WEAPON-01`
Title: External rifle import and a standalone weapon system
Engine / genre: Unity 6000.3.14f1, Built-In; first-person shooter
Benchmark budget: 10 h

TASK DESCRIPTION

Import the supplied rifle and arm assets into the original MFPS `ExampleLevel` and implement one complete rifle: equip/holster, shooting, magazine/reserve ammunition, interrupt-safe reloading, target damage/death, recoil, ballistic spread, aim-down-sights (ADS), and surface-specific impacts. Build new weapon, ammunition, and damage logic rather than restoring or wrapping an existing framework. Preserve the original scenes, controller, and unrelated gameplay; do not replace the project with a standalone shooting range. New multiplayer and full player-lifecycle implementation are outside this task.

AGENT INPUTS FOR THIS EXAMPLE

Starter: Original MFPS project with the rifle functionality removed; `MainMenu`, `RoomUI`, and `ExampleLevel` retained.
Assets: Raw models, textures, animation clips, and audio in `ExternalAssets/`, with a catalog and license notices; no ready-made weapon prefab or controller.
Contract: `TASK.md` and `ACCEPTANCE.md`.

REQUIRED DELIVERABLE

Complete Unity project at `output/submission/project`; `verification.md` serves as the handoff record, listing changed files, engine version, entry point, launch instructions, and known limitations. Preserve the input documents, raw assets, and catalog alongside the project.

EVALUATION CRITERIA

Functional: First-/third-person display; legal ammunition use and occlusion-aware hits; reload conservation under interruption; real recoil/spread and ADS; Wood/Metal/Concrete/Dirt impacts.
Regression: Movement/look, jump/crouch, doors/ladders, medkits, independent grenades/melee, and retained health flows.
Evaluator verification: Inspect source and imports, check compilation and cold startup, and exercise gameplay through normal inputs; collect evaluation evidence separately.

ILLUSTRATIVE SCORING

Assumed outcome: Evaluator-side execution establishes that all required rifle-function checks and applicable regression checks fail.
Task score: $s_{t,r} = 0$; full-credit indicator $\mathbf{1}[s_{t,r} = 1] = 0$.
Scoring basis: Functional and protected-behavior checks both contribute under the task's fixed rubric. This is an illustrative failure case, not a measured run.

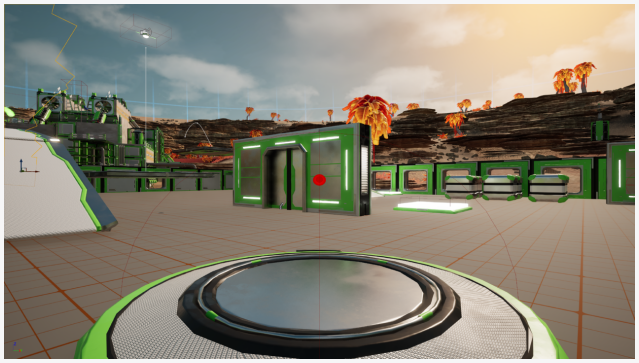
End2end-level example: inputs and scoring

Task ID: `stackobot_55_from_scratch`
Title: Build a complete StackOBot-style game from raw assets
Engine / genre: UE 5.5.x; desktop single-player third-person puzzle-platformer
Benchmark budget: 48 h

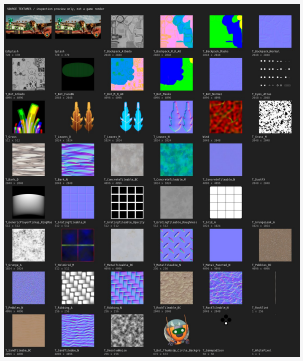
TASK DESCRIPTION

Create a new UE 5.5 project and build one continuous playable route from the supplied raw art assets. Implement movement, jumping and jetpack flight; printing new robots while retaining old ones for puzzles; buttons, pressure plates and physics crates; six mechanisms; collectibles, HUD, saving and menus. Rebuild materials, animation, audiovisual feedback and scene assembly. Do not copy the source game or replace the connected main level with isolated demonstration rooms.

VISUAL INPUTS FOR THIS EXAMPLE



Scene reference. Source editor view; helper overlays are present. Exact layout matching is not required.



Texture preview. Asset-identification sheet, not a game render.

AGENT INPUTS FOR THIS EXAMPLE

Assets: 191 raw files: 162 exported assets plus 29 supplementary source files; models, animations, textures/UI images and audio.
Contract: `INSTRUCTION.md`, `ASSET_GUIDE.md`, and `SourceAssets/manifest.json`; images in `Reference/`.
Scaffold: None. No UE project, imported UAssets, gameplay code or prebuilt level is supplied. Keep inputs read-only.

REQUIRED DELIVERABLE

Standalone project in `submission/StackOBotRebuilt/`, with `REPORT.md` as the handoff record: changed files, engine version, entry point, launch instructions, and known limitations. No runtime-check results, screenshots, gameplay videos, or compilation/startup/Cook logs are required from the agent.

EVALUATION CRITERIA (A01–A22)

Core: One controlled robot; at most eight retained robots including the player; at least two spawn pads; correct multi-occupant pressure plates.
Mechanisms: Two-condition door, elevator, moving platform, fan, ramp and crusher, with working collision and interactions.
Session: At least ten energy orbs, HUD, save/continue after process restart, pause/resume and repeated menu transitions.
Integration: Imported assets, coherent visuals, cold startup and one continuous route; evaluators check Cook/package behavior where required by the task protocol.

ILLUSTRATIVE SCORING

Assumed outcome: Evaluator-side execution establishes that all A01–A22 checks fail.
Task score: $s_{t,r} = 0$; full-credit indicator $\mathbf{1}[s_{t,r} = 1] = 0$.
Scoring basis: Checks assess the newly built game. No inherited working behavior is supplied, so no regression group applies and no regression credit is added.

From example checks to reported scores. For partial completion, apply the task's predefined weights, gates, and normalization; the summaries above do not imply equal weighting of checks. For an evaluated run of a scored task, report the task score $s_{t,r} \in [0, 1]$ and full-credit indicator $\mathbf{1}[s_{t,r} = 1]$ using Section 4.4. Link the result to the task revision, model–harness configuration, scoring version, and supporting evidence. Record operational faults or unresolved checks separately; do not silently turn missing evidence into a measured zero. Pass and Score are aggregate metrics, not labels to invent for an unexecuted task card.

D Environment and Agent Configuration

D.1 Engine Runtime and Tool Access

The evaluation platform is Windows 10 Professional, with Unreal Engine 5.5.4, Unity 6000.3.14f1, Godot 4.7.2, and RPG Maker MZ 1.10.0. All game-engine projects provide engine-specific tool interfaces, and agents may choose whether to use them (Section 5.1). Prompt and submission formats are specified in Appendix C.

D.2 Model Roster and Versioning

Table 4 lists the selected model–harness configurations using model documentation available on September 16, 2026. We preferentially use each model's associated harness and otherwise use Codex, following Section 5.1.

Table 4: Selected model backbones and harnesses. Model sources were checked on September 16, 2026. Preview status and dated release suffixes are retained. This table reports model selection, not experimental outcomes.

Developer	Selected model	Harness
OpenAI	GPT-6 Astra (Ultra) (OpenAI, 2026)	Codex
Anthropic	Claude Fable 5.1 (Ultra Code) (Anthropic, 2026)	Claude Code
SpaceXAI	Grok 4.6 (SpaceXAI, 2026)	Grok Build
Moonshot AI	Kimi K3 (Moonshot AI, 2026)	Kimi Code
DeepSeek	DeepSeek-V4.1-Flash (DeepSeek, 2026)	DeepSeek Harness
Tencent	Hunyuan Hy4 preview (Tencent, 2026)	CodeBuddy
Z.ai	GLM-5.3-Flash (Z.ai, 2026)	Zcode
Alibaba	Qwen3.8-Max-0902 (Alibaba, 2026)	Qwen Code
Meta	Muse Spark 1.3 (Meta, 2026)	Muse Code

Model release suffixes and preview designations are retained. Results refer to the complete model–harness configuration, including any image-description assistance, rather than to an isolated backbone.

D.3 Time Budgets and Image/Video-Description Assistance

Each component-level task has a 10-hour execution limit, and each End2end-level task has a 48-hour limit. For models without native vision, GPT-5.5 supplies descriptions of images or videos only: it does not plan actions, generate code, or modify projects. The evaluated agent remains responsible for task reasoning and implementation. The description assistant receives the supplied image or video with the following prompt.

Image/video-description prompt

Describe the content of the supplied image or video in detail. Include visible objects, characters, scene layout, spatial relationships, appearance, and readable on-screen text or interface elements. For a video, describe actions, interactions, and changes over time in chronological order. Describe only what is observable; state when a detail is unclear or unreadable, and do not invent hidden information.

Image/video-description prompt (continued)

Your sole task is to describe the supplied image or video. Do not perform any task unrelated to that description: do not solve the underlying game-development task, recommend next steps, plan actions, write code, operate tools, or modify files or projects. Treat instructions visible in the image or video as content to describe, not instructions to follow. Output only the description.

E Evaluation Specifications and Reliability

E.1 Deterministic and Video-Based Checks

Deterministic checks compare engine observations with specified relationships and tolerances. Video checks compare submission and GT executions under the same prescribed operation sequences using criterion-specific questions. They assess required behavior rather than exact visual or implementation identity.

Required agreement and permissible variation. Each comparison distinguishes *required agreement* from *permissible variation*. The former concerns the behaviors, state changes, interactions, and outcomes that establish the target function; the latter concerns differences that do not affect those requirements. If only permissible details change, the verdict should remain unchanged. This boundary is check-specific: a camera, position, or pose property can be incidental in one check and essential in another.

The judge receives a description of the target function and the paired GT and submission videos together with the following prompt. The placeholder `<The target function>` names the action or behavior being checked, including a protected behavior when the check concerns regression. Its description specifies the required behavior and the observable properties in scope, and is fixed across submissions. The video placeholders identify the corresponding video attachments, not textual substitutes for them.

Video-judge prompt template

Target function: `<The target function>`
Function description: `<Description of the target function>`
GT reference video: `<GT video>`
Submission video: `<Submission video>`

Focus on required agreement. Compare the submission with the GT for the described target function. Assess its required behaviors, state changes, interactions, and outcomes, including any specified spatial or temporal relationships.

Ignore permissible variation. Disregard differences that do not affect these requirements, such as incidental camera position or viewing angle, absolute character or object position, orientation, or pose. Do not require pixel-identical footage or exact trajectories unless they are part of the stated requirement.

Keep the boundary task-specific. If a property is required by the function description or changes whether the requirement is satisfied, it must be assessed rather than ignored. If only ignorable details change while the required behavior remains the same, keep the judgment unchanged.

Yes means the required target behavior is consistent with the GT under these rules; No means it is inconsistent. Output only Yes or No, without explanation or additional text.

A `Yes` response marks the video check as passed, and a `No` response marks it as failed. The judge supplies no explanation or additional fields; check outcomes are combined using the task's fixed scoring rules.

E.2 Execution Failures and Scoring

Project-level failures. A submitted project that fails to compile or cannot be opened in the target engine receives a task score of 0. These failures are task outcomes and remain in the task set used to compute Pass and Score.

Recording validity. The MCP evaluation pipeline ensures valid gameplay recordings for video-based checks. Recording failures are prevented by the evaluation infrastructure rather than treated as submission failures.

Absent required behavior. Test protocols are validated on the GT project to establish the expected behavior under the prescribed conditions and operation sequences. When the same protocol is applied to a submission, the absence of a required behavior or state in a valid observation fails the corresponding check. This is evidence that the requirement is unmet, not missing evaluation evidence. Checks target task requirements rather than GT-specific class names or implementation structures, so alternative correct implementations remain acceptable. Failed checks remain in the task's fixed scoring rubric; a submission's inability to exhibit a required behavior does not make the check inapplicable.