GRACE-BENCH: CAN CODING AGENTS MAKE GAME SYSTEMS WORK TOGETHER?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly used to implement features and repair defects in existing software projects. Game projects make this work especially demanding: a single requirement often spans player input, physics, gameplay rules, object lifecycles, and visual feedback, so meeting it requires these systems to work together through shared runtime state and event sequences. To test whether coding agents can meet such complete game-development requirements, we introduce GRACE-Bench: 60 extension and repair tasks in existing Godot 2D projects, with 1,270 executable checks organized into gate, behavior, and integration layers. The layers cover execution prerequisites, designated mechanisms, and required cross-system interactions of the same task, so each answer is judged against its complete requirement and each failure is located in a layer. Across 12 models, 180 answers validly pass every gate and behavior check, so an evaluation without the integration layer would count all of them as complete; only 65 also pass every integration check. The other 115 (63.9%; task-resampling 95% interval 52.7–74.1%) fail checks of cross-system behaviors required by their task specifications. These failures span 34 tasks and all 12 models, and the proportion stays above one half when any single model or task is left out or when the ten most frequently failed integration checks are removed. These results expose a gap in current coding agents: they often implement the requested mechanisms without making them work together, a gap that checks of individual mechanisms alone would not reveal. Recorded runtime evidence and targeted replay trace selected failures to unmet requirements and their gameplay consequences, showing the diagnostic value of the layered checks. Because every answer is graded on all checks of its task, GRACE-Bench can be rerun to test whether future agents genuinely improve at meeting complete development requirements.

1 INTRODUCTION

Coding agents are increasingly used to implement features and repair defects in existing software projects (Yang et al., 2024; Jimenez et al., 2024; Deng et al., 2025; Zhou et al., 2026). Game projects make this work especially demanding, because a single requirement often spans several systems. Games couple player input, physics, gameplay rules, object lifecycles, and visual feedback through shared state and event sequences. An item pickup may update a resource counter, remove a world object, refresh the interface, and free a quota slot for a later spawn. Correctness depends on these state changes remaining consistent and occurring in the required order. A game-development task therefore requires both the designated mechanisms and the relationships among them, and evaluating it requires checking whether these behaviors work together at runtime.

Recent benchmarks provide several approaches to evaluating game development. GameDevBench (Chi et al., 2026) pairs tutorial-derived Godot tasks with deterministic runtime tests, while GameCraft-Bench (Luo et al., 2026) evaluates complete Godot games through replay and multimodal rubric judging. WorldCoder-Bench (Lu et al., 2026) checks runtime states and transitions through hidden contracts, and GameGen-Verifier (Jia et al., 2026) uses injected states and bounded interactions to verify specification keypoints. We ask: **can coding agents meet complete game-development requirements, in which several game systems must work together?** Answering this requires grading the designated mechanisms and the required interactions among them within the same implementation. An implementation whose mechanisms all pass may still fail a required

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.

interaction; without checks of these interactions it would count as complete, and tracing such failures to the specification and runtime evidence shows which requirement remains unmet.

To study this question, we introduce GRACE-Bench (Game Runtime And Capability Evaluation), a benchmark of 60 extension and repair tasks in existing Godot 2D projects spanning ten gameplay genres. Each task supplies a starter project, a development specification, and any referenced public observations. Its executable checks are organized into three layers: *gate* checks test execution prerequisites and access to the tested objects; *behavior* checks exercise designated mechanisms and their boundary cases; and *integration* checks test the relationships among mechanisms required by the task specification, including state consistency, event ordering, and resource accounting. All three layers assess requirements of the same development task. For each submitted implementation, we retain outcomes across all three layers and quantify integration failures conditional on validly passing every gate and behavior check.

The benchmark contains 1,270 executable checks. Each task includes a public contract specifying how tested objects can be reached, driven, and observed. Private probes, installed after the agent finishes, execute the specified scenarios and record named runtime outcomes; deterministic assertions over those outcomes determine the check results. Reference implementations, deliberately faulty implementations, and recorded calibration outcomes support inspection of the checks and their correspondence to the specification, addressing the test-oracle problem (Barr et al., 2015; Liu et al., 2023; Yu et al., 2025). Seven diagnostic dimensions—state, lifecycle, timing, numeric, visual, physics, and the game's decision rules—group checks by the kind of requirement they assess. Per-check observations and archived agent trajectories provide evidence for examining where an implementation fails and, when the evidence supports it, why.

Our analysis covers 720 recorded task–model outcomes from 12 models and shows that cross-system failures are widespread. Among 180 answers that validly pass every gate and behavior check, 115 (63.9%; 95% interval 52.7–74.1% over task resamples) fail at least one integration check, so the number of complete answers falls from 180 to 65 once integration checks are included. These failures span 34 tasks and all 12 models, and removing the ten most frequently failed integration checks still leaves 97 failing answers (53.9%). Within the 180 answers, timing requirements such as cooldowns, timers, and frame-rate independence fail at the highest rate per check (26.9%), and lifecycle requirements such as spawning and removal account for the most failed checks (93 of 291). The failures we trace arise where one system depends on another: a correct rule may receive the wrong state from another system, or the outcome may change when events from two systems fall on the same frame.

Consider a deferred spawn that must coordinate an item quota with the state of a moving vehicle (Task 48; Figure 1). A fuel can cannot spawn while the quota is full. Once a slot opens, the task requires it to appear at the larger of its planned position and 500 pixels ahead of the car. An archived GLM-5.3 answer implements the placement formula correctly and passes every gate and behavior check, yet a separate replay places the can 500 pixels *behind* the car. This 1,000-pixel error arises because the spawner fails to retrieve the car's position and supplies zero to the formula (Section 5.1). Correct placement therefore requires the spawner to apply the rule using the vehicle's current state when the slot becomes available.

This case illustrates what GRACE-Bench measures: whether the requested mechanisms work together, and which requirement fails when they do not.

Contributions. We make three contributions:

- **A benchmark for complete game-development requirements.** GRACE-Bench comprises 60 extension and repair tasks in existing Godot 2D projects across ten gameplay genres, with 1,270 executable checks at three layers: gate checks for execution prerequisites, behavior checks for designated mechanisms, and integration checks for the cross-system interactions each task requires. Retaining outcomes in all three layers for each answer shows which layers fail; seven diagnostic dimensions classify non-gate checks (Section 3).
- **Inspectable checks.** Tasks are built through a staged pipeline with separated roles and human approval gates. Reference and deliberately faulty implementations with calibration records support inspection of the checks against the specification (Section 3.4 and Appendix G).

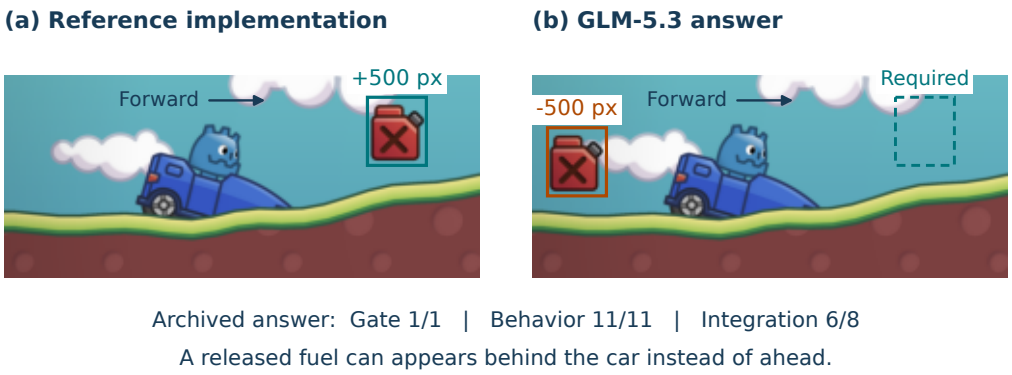


Figure 1: **Behavior checks pass, but a task-required interaction fails.** Separate replays of the evaluated scenario for Task 48's reference implementation (left) and an archived GLM-5.3 answer (right). At release, the car is at $x = 23{,}000$; the required can position is 23,500, but the answer produces 22,500. The dashed box marks the required position, not a rendered object. The layer counts come from the archived full evaluation; the screenshot replay checks only scene readiness and deferred-release placement.

- **Evaluation of 12 models and failure analysis.** Cross-system failures are widespread: gate and behavior checks alone would count 180 answers as complete, of which 65 pass every integration check; the other 115 span 34 tasks and all 12 models. Runtime evidence and targeted replay show how selected failures appear in gameplay and, where possible, trace them to their causes (Sections 4 and 5).

2 Related Work

From code generation to project modification. HumanEval (Chen et al., 2021) evaluates function synthesis with executable tests. SWE-bench and SWE-bench Pro (Jimenez et al., 2024; Deng et al., 2025) evaluate issue resolution in existing repositories, while FeatureBench (Zhou et al., 2026) covers feature implementation and reconstruction. SWE-agent (Yang et al., 2024) studies the agent–computer interface for these workflows. GRACE-Bench evaluates edits to existing game projects, where task completion requires specified mechanisms and their runtime interactions to hold within the same implementation.

Game development and runtime verification. GameDevBench (Chi et al., 2026) pairs tutorial-derived Godot tasks with deterministic runtime tests, verified references, and human review. GameCraft-Bench (Luo et al., 2026) evaluates complete Godot games through replay and multimodal rubric judging. WebGameBench (Zhang et al., 2026) retains per-requirement pass/fail/unverified evidence from browser interaction, and GameXpert-Bench (Chen et al., 2026) evaluates generation, repair, and optimization with runtime, integration, and regression checks. WorldCoder-Bench (Lu et al., 2026) verifies Three.js worlds through hidden, mutation-hardened state and transition contracts. GameGen-Verifier (Jia et al., 2026) uses specification keypoints, injected states, and bounded interactions, combining visual judgments with applicable programmatic assertions. GRACE-Bench retains check outcomes across its three layers and examines integration failures *conditional on validly passing every gate and behavior check*. This analysis identifies additional failed checks in the same implementation and links them to the task requirements. Appendix I compares the evaluation objects, evidence, and units of judgment of these protocols.

Test adequacy and grading evidence. EvalPlus (Liu et al., 2023) shows that additional tests expose errors missed by original code-generation benchmarks; UTBoost (Yu et al., 2025) examines insufficient tests and test-log parsing in SWE-bench. The test-oracle literature (Barr et al., 2015) distinguishes executing a program from deciding whether its outcomes are correct. GRACE-Bench connects requirements, runtime scenarios, and deterministic assertions, with reference and delib-

Table 1: Operational meanings of the three layers, illustrated with requirements from Task 48.

Layer	Intended scope	Task 48 example
Gate	Runtime readiness and access through the declared interfaces.	The scene loads and the car, terrain, spawner, and public contract are available.
Behavior	Designated mechanisms and their boundary cases.	A collected object occupies its quota slot until it leaves the scene after the specified delay.
Integration	Required relationships, event ordering, and consistency across conditions.	Deferred release respects the freed quota and computes placement using the car's current position.

erately faulty implementations providing calibration evidence (Section 3.4). The supplementary workbench study examines feedback during development while retaining executable checks for final grading (Appendix F).

3 GRACE-BENCH

3.1 TASKS AND DESIGN RATIONALE

The evaluation set contains 60 extension and repair tasks using 34 configured Godot 2D template paths. The tasks span ten primary gameplay genres (Figure 8 in Appendix H); platformers and runners account for 18 tasks and puzzle games for 8. Each task supplies a starter project, a `TASK.md` specification, and any referenced public observations. The agent modifies the project without access to the grading probe or reference solution. Tasks can share a game base, and the 34 paths are grouping identifiers rather than independent codebases. Appendix G describes how the tasks were constructed.

The tasks require consistent behavior across interacting game mechanisms. In Task 48, for example, a pickup can change resource accounting, remain in the scene while occupying a quota slot, and subsequently free that slot for a deferred spawn whose placement depends on the vehicle's current position. The development requirement covers this sequence as well as the individual mechanisms. Some tasks also require inferring rules from observations: Task 3 supplies lantern brightness and pickup records while withholding the underlying oil-accounting rules. Its implementation must reproduce these observations through shared state. The task set thus covers both explicit interaction requirements and requirements that combine rule recovery with implementation.

3.2 FROM TASK REQUIREMENTS TO EXECUTABLE CHECKS

Each task's public contract specifies how tested objects can be reached, driven, and observed, together with the runtime environment, timing, tolerances, and determinism conditions. This contract can name resource paths, node groups, properties, or methods. These declared interfaces allow the same evaluation scenario to be applied to different implementations; implementation choices remain open within the specified interfaces.

The grading artifacts connect specification clauses to scenarios and named check outcomes. A private probe, installed after the agent finishes, initializes the required state, drives the tested objects, and records observations. Deterministic assertions compare those observations with the task's requirements. For example, Task 48's deferred-release check fills the quota, triggers a deferred spawn, frees a slot, and observes the released item's position against both the planned position and the car's current position. The expected placement and timing come from Section 4.4 of the public task specification.

The 1,270 checks comprise 100 gate, 642 behavior, and 528 integration checks. All three layers assess the same development task (Table 1). Behavior checks can already involve multiple systems; integration labels select additional required relationships and combined conditions. Retaining the outcomes of all three layers within each implementation supports analysis of which requirements still fail after all gate and behavior checks pass.

Table 2: The seven diagnostic dimensions and what their checks read.

Dimension	What its checks read
Numeric	Quantities and curves: difficulty ramps, resource growth, and parameter schedules.
Timing	Event timing: cooldowns, timers, and frame-rate independence.
State	Modes and transitions: win and loss decisions, scoring, and state-machine switches.
Lifecycle	Creation and removal: spawning, object pooling, and off-screen cleanup.
Physics	Motion and collisions: collision layers, triggers, and rigid-body response.
Decision	The game's own decision rules, such as targeting, pursuit, and patrol; not the evaluated model's intelligence.
Visual	Objective graphical properties: asset specifications and runtime presentation, not aesthetic judgment.

An absent, unexecuted, or explicitly unscored check is an unavailable measurement. We retain valid passes, valid failures, and unresolved outcomes separately; an all-pass layer requires a valid pass for every expected check. Appendix A gives the full rules.

3.3 Scores and diagnostic dimensions

The primary leaderboard uses *dimension-complete weighted scoring*: each dimension first receives a binary score, and these scores are then combined into a weighted task score. Let D_t denote the set of non-gate dimensions present in task t. The score $s_{t,d}$ for dimension d is

$$s_{t,d} = \begin{cases} 1, & \text{if every behavior and integration check in dimension } d \text{ validly passes,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

A failed or unmeasured check therefore gives its dimension zero credit.

A failed or unmeasured gate sets the task score S_t to zero. If every gate check validly passes, we weight each dimension by $1 + n_{t,d}$, where $n_{t,d}$ is the number of integration checks assigned to dimension d in task t. These counts are fixed by the task's check set. The weighted task score is

$$S_t = \frac{\sum_{d \in D_t} (1 + n_{t,d})\, s_{t,d}}{\sum_{d \in D_t} (1 + n_{t,d})}. \tag{2}$$

The denominator normalizes the weights, so the task score lies between zero and one. The rule has two consequences. A dimension earns credit only when it works in every tested case, so an implementation that fails one boundary case of a mechanism receives no credit for that dimension, however many of its other checks pass. Dimensions with more integration checks receive more weight, so the score depends more on the dimensions that carry more of the task's required cross-system relationships. The leaderboard averages task scores equally. Layer-wise check rates are separate diagnostics, not the leaderboard score; Section 4.2 compares the resulting ranking with unweighted and per-check alternatives.

Non-gate checks carry one of seven diagnostic dimensions (Table 2); the dimensions present in a task form the set D_t scored above. They are overlapping categories, not isolated abilities.

3.4 Grader calibration and supporting evidence

Calibration links the proposed checks to implementations with known intended behavior. We inspect stored reference and deliberately faulty implementation records for all 60 tasks. These records provide evidence about the grader.

Recorded acceptance and rejection. All 60 recorded reference runs contain the complete set of named rubric checks with passing Boolean outcomes and no checks marked unscored. All 275 stored faulty-implementation records contain at least one definite measured failure. Reference acceptance establishes agreement with the intended solution, while a faulty implementation's rejection shows that at least one check detects a failure in that record.

Scenario coverage. Scenario manifests link requirements to evaluation scenarios, while witness manifests identify deliberately faulty implementations for critical scenarios. We cross-check these assignments against the recorded calibration outcomes and the scenario-to-check mappings. All 432 critical scenarios have at least one designated faulty implementation with a definite failure on a check mapped to that scenario. The corresponding implementations also pass every gate check.

4 Empirical Evaluation

4.1 Evaluation setup

We evaluate 12 models on 60 tasks each, yielding 720 task–model answers, all of which enter the leaderboard. In 707 runs, the agent submits its project, indicating that it has finished editing; correctness is determined by the checks. The other 13 runs stop before submission because the model repeatedly fails to produce an executable action in the required format; in 12 of them, the rejected responses include output truncated at the maximum response length. These 13 projects are still graded for the leaderboard by the same rule (11 receive zero and two receive partial credit), but check-level diagnostics exclude them because an unsubmitted project may be an unfinished edit. None of the 13 passes every gate and behavior check, so their exclusion does not change the set of answers that pass every gate and behavior check (Section 4.3). The diagnostics retain every valid pass/fail observation from the 707 submitted answers, including observations from runs with other checks unavailable. Appendix A lists the checks that could not be measured.

All runs use the same prompt templates and the mini-swe-agent 2.4.5 text-action protocol in isolated Docker projects with two CPU cores, 2 GB memory, no network access, and a 300-second timeout for each command; grading is private and follows the solving session (Appendix B). No limit on steps or total time affected any run: the prompts state no such budget, and every run ended either with the agent's submission or, for the 13 runs above, after repeated format failures.

4.2 Overall performance

Table 3 reports the 12-model leaderboard, ranked by dimension-weighted mean (DW). GPT-6 Astra leads with 61.1% on this metric and 30.0% on the Resolved measure. GPT-6 Astra leads Claude Opus 5 by 8.8 points, with a paired 95% interval of -0.9 to 18.7, so the two are not separated; among adjacent pairs, only GLM-5.2 and Qwen3.7-Max are. Larger gaps are separated: GPT-6 Astra leads every model ranked third or lower, Claude Opus 5 every model ranked fourth or lower, and Claude Fable 5 every model ranked fifth or lower. Alternative aggregations give Kendall's τ_b with the DW ranking of 0.97 for an unweighted mean of dimension-complete scores and 0.97 for the mean per-task fraction of valid non-gate checks passed; both keep the same top three, and the largest rank change is 1. For Resolved it is 0.74: Qwen3.8-Max takes second place. In these results, each model's Resolved count equals its answers that pass every gate and behavior check minus those among them that fail an integration check (the last two columns of Table 3).

4.3 Integration failures after gate and behavior success

Of the 707 submitted answers, 627 validly pass every gate check, and 180 of these also validly pass every behavior check. Of these 180, 115 (63.9%) have at least one valid failing integration check, with a 95% task-resampling interval of 52.7–74.1%. Without the integration layer, all 180 would count as complete; with it, 65 do. These failures span 34 tasks and all 12 models, and Table 3 gives the split for each model. All 180 answers have valid outcomes for every expected check, so restricting the analysis to the 704 submitted answers with complete check records leaves every count in this subset unchanged.

Reverse comparison. The layers can also be conditioned in the other order. Of the 119 answers that validly pass every gate and integration check, 54 (45.4%) fail at least one behavior check. The two answer-level proportions are not directly comparable: an answer in the first subset carries 8.8 integration checks on average and one in the second carries 12.0 behavior checks, and the subsets cover different tasks (42 and 38, with 29 in common). Per check, integration checks fail more often

Table 3: The 12-model, 60-task leaderboard, ranked by dimension-weighted mean (DW; Section 3.3). DW and Resolved are percentages; the interval resamples the 60 tasks. Resolved is the fraction of tasks where every check validly passes. The last two columns count a model's answers that validly pass every gate and behavior check, and those among them with a valid failing integration check. Table 5 reports the Almost resolved rate, rounds, time, and text volume.

#	Model	DW ↑	95% interval	Resolved ↑	All gate and behavior pass	Integration failure
1	GPT-6 Astra	61.1	52.5–69.5	30.0	28	10
2	Claude Opus 5	52.3	44.1–60.3	13.3	23	15
3	Claude Fable 5	48.6	40.2–56.9	13.3	22	14
4	Qwen3.8-Max	39.7	31.0–48.6	15.0	16	7
5	GPT-5.6 Sol	37.5	29.8–45.4	8.3	20	15
6	GPT-5.5	34.2	26.8–42.0	5.0	12	9
7	GLM-5.3	32.7	24.3–41.4	10.0	16	10
8	Kimi K3	26.4	19.7–33.5	1.7	10	9
9	Claude Opus 4.8	23.1	17.1–29.7	3.3	9	7
10	DeepSeek-V4-Flash	22.6	15.1–30.7	6.7	15	11
11	GLM-5.2	21.3	15.3–27.7	1.7	8	7
12	Qwen3.7-Max	8.3	4.8–12.2	0.0	1	1

in the first subset than behavior checks do in the second: 291 of 1,585 (18.4%) against 113 of 1,426 (7.9%).

Concentration and robustness. The 180 answers come from 42 tasks, and the five tasks with the most failing answers account for 40 of the 115 failures. Most failing answers fail more than one integration check: 46 fail exactly one, 21 fail two, and 48 fail three or more. Removing the most frequently failed integration checks lowers the proportion to 59.4% after three checks, 53.9% after ten, and 47.2% after twenty (Appendix A defines the order). Leaving out one model at a time gives 62.4–69.1%, and leaving out one task at a time gives 62.0–65.7%.

4.4 Layer and dimension profiles

For context, the following profiles describe all submitted answers unless stated otherwise. Valid check-pass rates are 91.3% for gate, 70.1% for behavior, and 57.8% for integration. Figure 2 shows lower integration than behavior rates for all 12 models. These rates pool valid checks within each model; they use different denominators and weighting from the leaderboard.

Across integration checks, aggregate pass rates are lowest for timing requirements (51.2%) and numeric requirements (51.2%), compared with 67.6% for game decision-rule checks. These profiles describe which designated requirements pass less often across the submitted population. Appendix D provides the complete model-by-dimension matrices and exact counts.

Within the 180 answers that pass every gate and behavior check, integration failures occur in every dimension. Timing has the highest failure rate per valid check, 26.9% (64 of 238, in 47 answers), followed by state (19.7%), visual (17.8%), lifecycle (17.1%), numeric (16.3%), and decision rules (13.9%); physics has 3 failures in 64 checks. Lifecycle checks contribute the most failures, 93, spread over 59 answers.

We also average the behavior-minus-integration difference within each task and weight the 60 tasks equally. The resulting mean difference is 12.0 percentage points, with a task-bootstrap 95% interval of 8.7–15.3 points; 51 tasks have a positive difference. Appendix E reports template-path weighting and comparisons grouped by the number of diagnostic labels attached to a check.

5 Failure Analysis

To understand what the additional failed checks mean for task completion, we connect the public requirement, evaluated scenario, observed outcome, and supporting implementation evidence for two answers that pass every gate and behavior check in the archived evaluation. The GLM-5.3

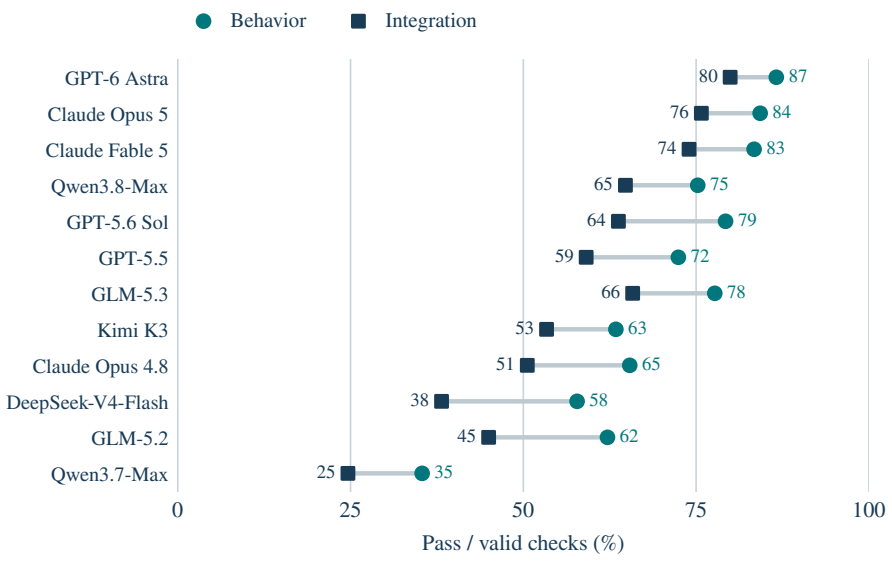


Figure 2: Behavior and integration check-pass rates by model, ordered by the leaderboard. Each endpoint pools valid checks over that model's submitted answers; unresolved checks are excluded. This diagnostic view does not use the leaderboard's gate veto or dimension-complete weighting.

answer to Task 48 (fuel pickup) passes 1/1 gate, 11/11 behavior, and 6/8 integration checks; it has a targeted runtime replay and an identifiable code defect. The GPT-6 Astra answer to Task 3 (lantern) passes 2/2, 8/8, and 6/7; it has an archived boundary mismatch whose cause remains unresolved, and it was not rerun.

5.1 Deferred spawning fails to find the vehicle

Requirement and scenario. Task 48's public specification requires a deferred item to spawn at

$$x_{\text{release}} = \max(x_{\text{planned}},\, x_{\text{car}} + 500)$$

within a tolerance of ± 4 pixels after a quota slot becomes available. The probe drives quota deferral and release under two vehicle positions. In the condition shown in Figure 1, the planned position is $22{,}500$ and the car is at $23{,}000$, so the required release position is $23{,}500$.

Observed failure and cause. The archived GLM-5.3 answer passes every gate and behavior check but releases the can at $22{,}500$. A separate replay of this evaluated scenario reproduces the position mismatch, while the reference implementation releases it at $23{,}500$. The answer's can is therefore 500 pixels behind the car, where the task requires it to be 500 pixels ahead. Release timing and terrain-relative height meet the checked requirements in this scenario.

The submitted placement formula is correct. Its vehicle lookup, however, searches the spawner's ancestor nodes while the car is a sibling. The lookup fails and defaults the car position to zero, yielding $\max(22{,}500,\, 0 + 500) = 22{,}500$. This case connects a public requirement to a concrete defect in the state supplied across systems. The recorded earlier-layer passes coexist with that defect; they do not establish that the formula receives the correct runtime inputs. Appendix C.1 records the replay scope and evidence provenance.

5.2 Contact on the torch-expiry boundary

Task 3 requires contact exactly on a torch's burnout frame to refill nothing: the lantern timeline must match an untouched run, with no flare. The archived GPT-6 Astra answer passes all gate and behavior checks but fails the integration check for this boundary.

The saved probe aligns contact with the torch's departure frame. At the final sample, the untouched run has lantern tier 1 and alpha 0.60, while the contact run has tier 2 and alpha 0.78. Neither reports

a flare. The observed lighting states therefore differ where the specification requires matching timelines. The submitted code explicitly checks expiry before returning fuel, so an absent expiry check does not explain the record. The evidence locates a mismatch involving torch lifetime, contact, and lantern state; it does not establish the internal event order that caused it (Appendix C.2).

5.3 Diagnostic value for task evaluation

These examples show what the check records add to an overall score. The layer outcomes identify submissions whose designated prerequisites and behaviors pass evaluation; the named failures identify the remaining requirements to inspect. Scenario observations then specify the discrepancy, such as an incorrect release position or a changed lantern state at a forbidden refill boundary. Code and replay evidence can establish a cause, as in Task 48, or delimit what remains unknown, as in Task 3.

What these failures indicate. The integration failures are not isolated slips. Of the 115 answers that pass every gate and behavior check but fail an integration check, 69 fail two or more integration checks. They come from 34 of the 42 tasks that contribute such answers and from all 12 models. The failed requirements concentrate where one system depends on another. Timing requirements fail at the highest rate per check (26.9%), and lifecycle requirements such as spawning and removal account for the most failed checks (93 of 291). An answer can therefore implement every designated mechanism and still break a required interaction between them. Section 5 shows this directly. In Task 48, a deferred fuel can is placed behind the vehicle because its spawner fails to find the vehicle and falls back to a zero position, although the spawning mechanism itself works.

How reliable these verdicts are. Each check is calibrated before use. All 60 reference implementations pass every check, and all 275 stored faulty implementations are rejected with at least one definite measured failure. The conditional failure proportion also does not rest on a few checks, a few tasks, or one model. Removing the ten most frequently failed integration checks leaves 97 failing answers (53.9%), leaving out one model at a time gives 62.4–69.1%, and leaving out one task at a time gives 62.0–65.7%. Every one of the 510 designated faulty-implementation assignments to critical scenarios fails a check mapped to its scenario, so each critical requirement has a recorded rejection of an implementation that violates it.

6 Conclusion

GRACE-Bench evaluates whether coding agents can meet game-development requirements in which several systems must work together, using 60 existing-project tasks and 1,270 executable checks. Across 12 models, 115 of 180 answers that validly pass every gate and behavior check still fail integration checks targeting requirements of the same task, so an evaluation without these checks would count 180 answers as complete where 65 pass every check. Runtime evidence connects the examined failures to concrete gameplay discrepancies, including a deferred fuel can placed behind the vehicle because its spawner fails to find the vehicle and uses a zero position. Because every answer is graded on all checks of its task, the same tasks and checks can be rerun to test whether later agents improve at meeting complete requirements, rather than individual mechanisms alone.

AI Use Statement

We used generative AI tools to polish the writing of this paper and to help find related work. LLMs also generated the benchmark tasks through the pipeline described in Appendix G, which names the model used in each role. They produced the task designs, reference solutions, task statements, measurements, probes, rubrics, faulty implementations, and audits, and they screened task difficulty. The pipeline requires human approval of each task design, a human final reading of each task statement, and human acceptance of each finished task; humans also arbitrate escalated disagreements. The authors take responsibility for the final content of this paper, including text and artifacts produced with the aid of generative AI.

Ethics Statement

The tasks are built from open-source Godot projects. In accordance with their licenses, the task package identifies each task's source project and includes its license, and we comply with the corresponding license obligations.

Reproducibility Statement

The 60 tasks are available at https://figshare.com/s/4202f0d2bae53844122f. Appendix G describes in detail how the tasks were constructed.

References

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785. URL https://doi.org/10.1109/TSE.2014.2372785.

Kun Chen, Haorong Hong, Peizhong Gao, Jianfeng Lin, Tongxu Luo, Yuxuan Xie, Chenxu Liu, Jieling He, Zhongyuan Liu, and Zeno Zeng. GameXpert-Bench: How Far Are Coding Agents from Expert Game Development? *arXiv preprint arXiv:2608.21833*, 2026. URL https://arxiv.org/abs/2608.21833v1.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374v2.

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/2602.11103v2.

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

Yixu Huang, Bo Li, Na Li, Zhe Wang, Kaijie Chen, Haonan Ge, Qingyi Si, Yuanzhe Shen, Ruihan Yang, Guangjing Wang, and Hongcheng Guo. GUI agents for continual game generation. *arXiv preprint arXiv:2605.28258*, 2026. URL https://arxiv.org/abs/2605.28258v1. Version 1.

Chaobo Jia, Ruipeng Wan, Ting Sun, Weihao Tan, Borui Wan, Yuxuan Tong, Guangming Sheng, and Hong Xu. GameGen-Verifier: Parallel Keypoint-Based Verification for LLM-Generated Games via Runtime State Injection. *arXiv preprint arXiv:2605.07442*, 2026. URL https://arxiv.org/abs/2605.07442v1.

Yilei Jiang, Jinyuan Hu, Qianyin Xiao, Yaozhi Zheng, Ruize Ma, Kaituo Feng, Jiaming Han, Tianshuo Peng, Kaixuan Fan, Manyuan Zhang, and Xiangyu Yue. OpenGame: Open agentic coding for games. *arXiv preprint arXiv:2604.18394*, 2026. URL https://arxiv.org/abs/2604.18394v2. Version 2.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023. doi: 10.52202/075280-0943. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

Shuo Lu, Yinuo Xu, Kecheng Yu, Siru Jiang, Yongcan Yu, Yubin Wang, Haitao Yang, Yuxiang Zhang, Bin Wang, Ran He, and Jian Liang. WorldCoder-Bench: Benchmarking Physically Grounded 3D World Synthesis. *arXiv preprint arXiv:2606.01869*, 2026. URL https://arxiv.org/abs/2606.01869v2.

Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, Fan Bu, Wenyu Du, Xiaotong Zhang, Kyle Li, Shaobo Wang, Linfeng Zhang, Yuxuan Liu, Xin Lai, Chenxin Li, Yiduo Guo, Zhexin Zhang, Xinyuan Wang, Tianyi Bai, Ziniu Li, and Benyou Wang. GameCraft-Bench: Can Agents Build Playable Games End-to-End in a Real Game Engine? *arXiv preprint arXiv:2606.17861*, 2026. URL https://arxiv.org/abs/2606.17861v1.

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

Jianwen Sun, Chuanhao Li, Zizhen Li, Yukang Feng, Fanrui Zhang, Yifei Huang, Yu Dai, and Kaipeng Zhang. JAMER: Project-level code framework dataset and benchmark on professional game engines. *arXiv preprint arXiv:2606.19830*, 2026. URL https://arxiv.org/abs/2606.19830v2. Version 2.

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. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793v3.

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/2603.07106v2. Version 2.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.189. URL https://aclanthology.org/2025.acl-long.189/.

Wenyu Zhang, Guoliang You, Tianlun, Haotian Zhao, Tianshu Zhu, Haoran Wang, Xiaoxuan Tang, Mingyang Dai, Jingnan Gu, Daxiang Dong, and Jianmin Wu. WebGameBench: Requirement-to-Application Evaluation for Coding Agents via Browser-Native Games. *arXiv preprint arXiv:2605.17637*, 2026. URL https://arxiv.org/abs/2605.17637v2.

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*, 2026. URL https://arxiv.org/abs/2602.10975v1.

A Measurement Definitions

Check outcomes. Each expected check has one of three outcomes: a valid pass, a valid failure, or unresolved. A check is unresolved when it has no usable result, for example because the result is missing, duplicated, malformed, or marked unscored, or because the check could not run. Unresolved checks count neither as passes nor as failures. For a set of checks C, the diagnostic rate is

$$R(C) = \frac{\#\{\text{valid passes in } C\}}{\#\{\text{valid passes in } C\} + \#\{\text{valid failures in } C\}},$$

which is undefined when the denominator is zero. A layer passes only if every expected check in it is a valid pass. An answer has an integration failure if at least one of its integration checks is a valid failure.

Unresolved checks in this evaluation. Among the 707 submitted answers, the only unresolved checks are 53 skipped checks in three submissions: 32 in two Task 10 submissions and 21 in one Task 26 submission. In each, a required script fails to load or run (a parse error in one Task 10 submission and a call to a nonexistent method in the other two), so the checks that depend on it could not run. All three submissions fail a gate and receive zero task credit. The other 704 submitted answers have a valid outcome for every expected check. Table 4 gives the counts for each layer.

Table 4: Check outcomes by layer for the 707 submitted answers. Skipped checks come from the three submissions described above. Valid is Pass plus Fail, and Rate is Pass divided by Valid.

Layer	Pass	Fail	Skipped	Valid	Rate (%)
Gate	1,073	102	0	1,175	91.3
Behavior	5,263	2,240	44	7,503	70.1
Integration	3,584	2,618	9	6,202	57.8

Resampling. Each 95% interval draws the 60 tasks with replacement 10,000 times, recomputes the quantity from the drawn tasks (a task drawn twice counts twice), and takes the 2.5th and 97.5th percentiles; the template-path analysis in Appendix E resamples its 34 path means instead. For the DW difference between two models, both models use the same draws, and the two models are separated when this interval excludes zero. The check-removal analysis in Section 4.3 ranks integration checks by how many of the 180 answers fail them, breaking ties by task identifier and then check name, removes the k highest-ranked checks, and reclassifies every answer.

B Historical Leaderboard Protocol

Configuration metadata are readable in all 720 trajectories. All 720 record mini-swe-agent version 2.4.5[1]. They specify Docker with two CPU cores, 2 GB memory, no container network, a 300-second command timeout, and image tag `grace-bench:latest`. All readable records share the same system/task-template hashes for a single-bash-action protocol that permits agent-initiated Godot self-tests; the historical driver installs the private probe after solving. This is not interactive access to the hidden grader. None of the 720 trajectories reports a `TimeExceeded` or `LimitsExceeded` terminal state.

Trajectory text tokens. We estimate text volume from the complete trajectories of all 720 leaderboard answers using `tiktoken` 0.13.0 with `o200k_base` for every model. Input counts system and user messages, including tool observations and corrective feedback; output counts all assistant replies, retained reasoning, and recorded format-error replies within that answer. Each message is counted once, without accumulating replayed history or combining separate answers. Duplicated response metadata and reasoning copied into content are counted once. Unrecorded reasoning and API framing are excluded. Only DeepSeek-V4-Flash and GLM-5.3 have separately retained reasoning in some records, so these estimates describe recorded text volume rather than comparable billed

[1]Software source: `https://github.com/SWE-agent/mini-swe-agent`.

usage. Token medians use all 60 answers for every model, including answers ending in a format error.

Table 5: Additional leaderboard statistics, in the order of Table 3. Almost resolved is the percentage of 60 tasks where at least 95% of a task's valid non-gate checks pass; it includes resolved tasks. Rounds and Minutes are medians over all 60 answers; elapsed time includes adapter overhead. Input and Output are estimated complete-trajectory text tokens (k = 1,000), with medians over all 60 answers for each model. Each message is counted once, without accumulating replayed history. Recorded reasoning is included where available. These text volumes are not API usage totals.

#	Model	Almost ↑	Rounds	Min.	Input (k)	Output (k)
1	GPT-6 Astra	33.3	19.0	13.5	25.8	9.6
2	Claude Opus 5	20.0	59.5	131.5	35.1	30.8
3	Claude Fable 5	18.3	33.0	96.1	21.8	28.6
4	Qwen3.8-Max	20.0	113.5	213.0	52.1	51.4
5	GPT-5.6 Sol	18.3	18.0	7.1	26.1	6.3
6	GPT-5.5	6.7	28.0	16.8	30.9	9.1
7	GLM-5.3	15.0	142.0	242.3	57.7	316.7
8	Kimi K3	5.0	71.5	193.5	39.2	37.8
9	Claude Opus 4.8	5.0	52.5	34.2	22.3	57.5
10	DeepSeek-V4-Flash	6.7	86.0	71.4	51.7	320.3
11	GLM-5.2	3.3	249.0	143.4	83.5	82.0
12	Qwen3.7-Max	0.0	51.0	51.8	33.8	22.7

C Detailed Recorded Cases

This appendix gives the scope of the targeted Task 48 replay and the recorded evidence for the Task 3 case. Task numbers follow the 60-task roster in Appendix H.

C.1 Task 48: replay scope and provenance

The case uses the GLM-5.3 answer for `74_hill_climb_pickup_lifecycle`. Its full evaluation reports gate 1/1, behavior 11/11, and integration 6/8. Section 4.4 of its public `TASK.md` requires the deferred placement formula stated in Section 5.1, terrain-relative height, and entry within two physics frames of the release signal.

The separate replay assembles the stored project and changed files and invokes the existing probe's scene-readiness and deferred-release checks. It reproduces both the planned-position and car-ahead conditions. The car-ahead condition records planned $x = 22{,}500$, car $x = 23{,}000$, and required $x = 23{,}500$; the answer releases at $22{,}500$ and the reference at $23{,}500$. The submitted `collectible_spawner.gd` contains the correct placement formula but supplies zero when the ancestor-only car lookup fails.

The replay uses the controlled evaluation scenario, including its vehicle positioning, rather than a continuous natural-driving sequence. Only two checks are replayed; the full layer counts in Section 5 come from the archived evaluation. Screenshots are taken after the scenario returns, with the game paused and camera smoothing reset. Identical cropping and annotations are applied without repositioning game objects.

C.2 Task 3: burnout/contact boundary

This case connects a recorded evaluation outcome with the task contract, rubric, and submitted project files; causal interpretation is limited to the available evidence.

Observed: the sole failed check concerns natural torch expiry and same-frame contact. Section 5 of `TASK.md` requires that burnout-frame contact "refills nothing at all." The saved probe marks contact aligned with the torch's departure frame. At the final sample, the untouched run has lantern tier 1, alpha 0.60; the contact run has tier 2, alpha 0.78. Neither reports a flare. This records a

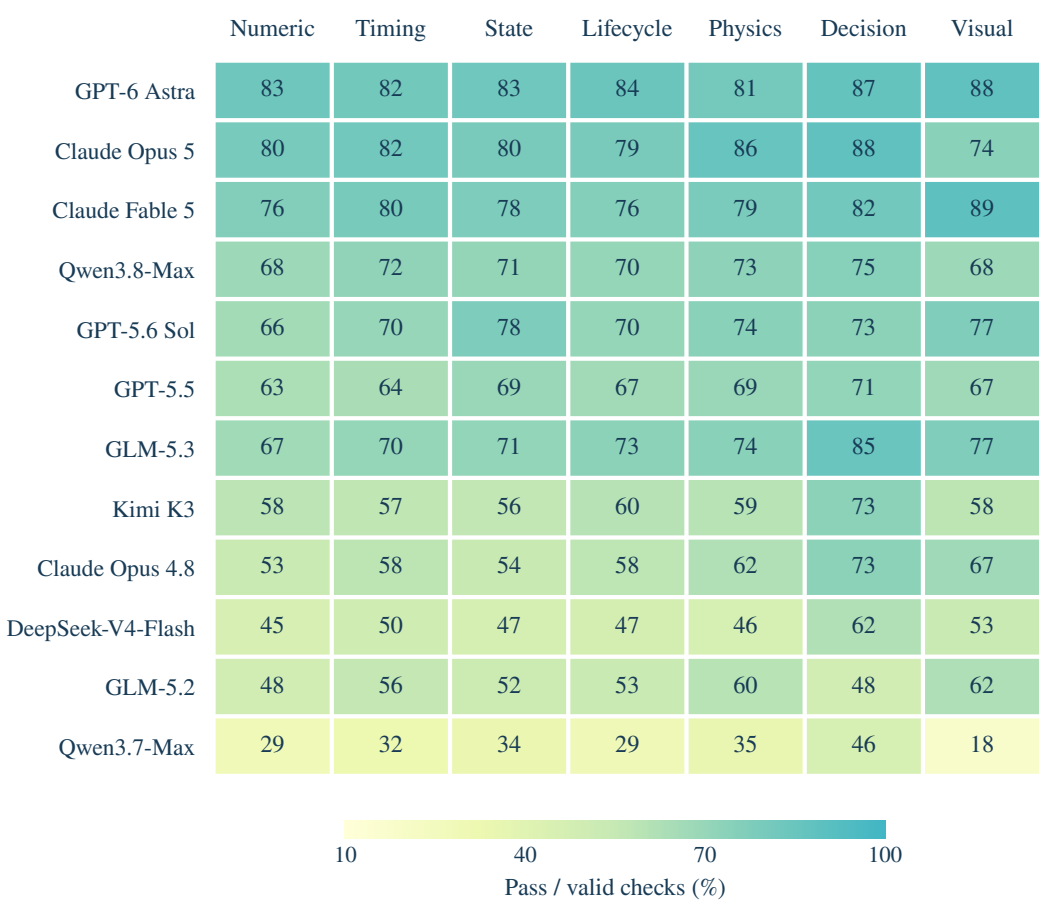


Figure 3: Model-by-dimension comparison across behavior and integration checks together. Values are rounded percentages of valid checks passed; color runs linearly from 10 to 100, the range spanned by both matrices in steps of ten. Decision denotes the game's own decision rules, not general model intelligence. Figure 4 gives the integration-only matrix.

boundary mismatch, not the internal event order: the submitted `torch.gd` explicitly checks expiry before returning fuel.

D Additional Dimension Comparisons

These profiles use all submitted answers. Figure 3 pools behavior and integration checks, while Figure 4 restricts the matrix to integration. Figure 5 compares the two layers within each diagnostic dimension, and Table 6 reports exact valid and unresolved counts.

E Check-Label Sensitivity Analysis

E.1 Task and template-path weighting

To reduce domination by tasks with more checks, we first compute behavior-minus-integration rates within an answer, average over calculable submitted answers within each task, then weight the 60 tasks equally. The mean difference is 12.0 percentage points; 51 tasks have a positive difference. Resampling these 60 task means 10,000 times gives a 95% empirical percentile interval of 8.7–15.3 points. A separate equal-template-path analysis yields 11.4 points (7.7–14.7). The latter changes both grouping and weighting, rather than merely adjusting the same estimator's standard error. Neither interval is an interval for the 115/180 conditional failure proportion.

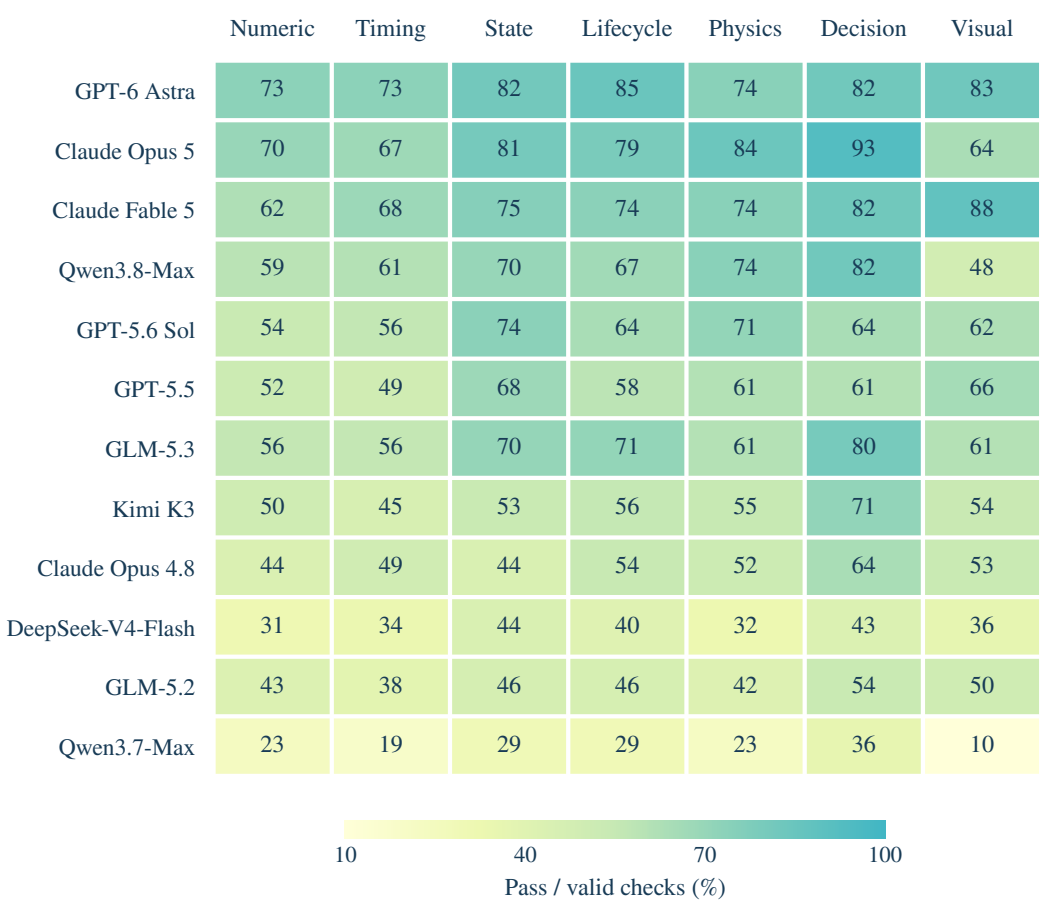


Figure 4: Integration-only model-by-dimension matrix. The color scale (10 to 100) and row order match Figure 3. A cell pools valid checks in that model–dimension combination.

Table 6: Exact dimension counts over submitted answers, for the dimensions defined in Table 2. P/F/S denote valid pass, valid failure, and skipped dependent checks.

	Behavior			Integration		
Dimension	P	F	S	P	F	S
Numeric	783	390	3	334	318	2
Timing	1355	536	11	532	508	0
State	691	331	6	745	470	3
Lifecycle	885	390	8	1155	766	4
Physics	614	266	5	216	153	0
Decision	218	66	0	225	108	0
Visual	717	261	11	377	295	0

E.2 Grouping by secondary diagnostic labels

Each rubric check declares a primary dimension and a secondary field. The secondary field mixes diagnostic dimensions with test-design tags such as boundary and ordering. We count only dimension entries to describe the check's declared diagnostic breadth, then group valid outcomes by this count (Table 7). This is a comparison of existing labels, with different checks and tasks represented in each group.

Behavior rates fall across the groups, from 929/1,218 (76.3%) at zero secondary dimensions to 41/80 (51.2%) at four. The three narrowest groups hold 77.6% of all valid behavior and integration

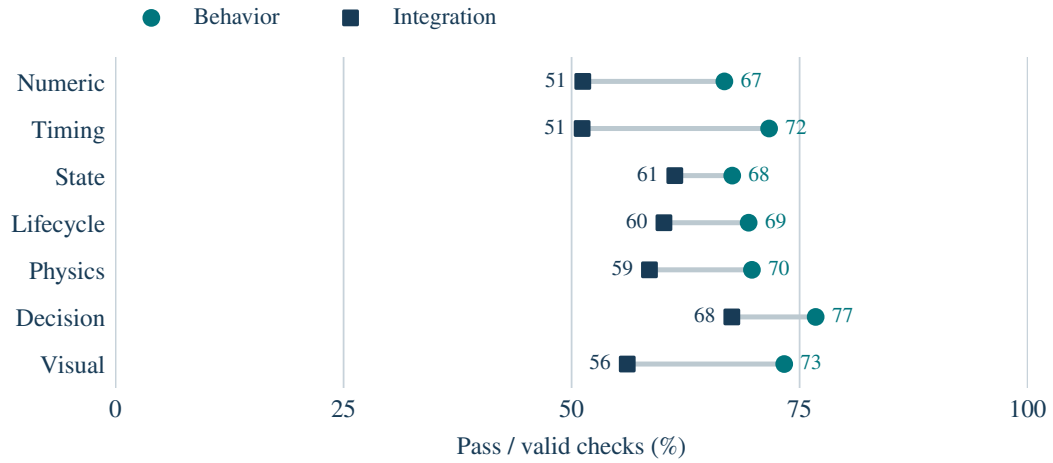


Figure 5: Behavior and integration within each diagnostic dimension, pooled over submitted answers. These compare different designated checks rather than matched variants of the same check. Timing and numeric have the lowest integration rates.

checks. In each, the integration rate is lower—by 17.1, 12.4, and 9.8 points—with a task-clustered interval excluding zero. At 3 secondary dimensions the behavior-minus-integration difference is 7.5 points with an interval spanning zero. At 4 secondary dimensions the behavior-minus-integration difference is 2.7 points with an interval spanning zero. At four, the rates are 48.6% integration against 51.2% behavior on only 80 valid behavior checks.

Reweighting behavior checks to match the integration layer's distribution of secondary-dimension counts changes the pooled 12.4-point difference but still leaves 9.9 points. This shows that the observed rate difference persists under this reweighting.

Table 7: Check-pass rates over submitted answers, stratified by the number of secondary *dimensions* a check declares. The rubric's secondary field also carries test-design tags; those are excluded, so the count summarizes declared diagnostic breadth. Rates are passes over valid outcomes; unresolved checks are excluded. Behavior checks in this roster declare at most four secondary dimensions and integration checks at most five, so the two layers share strata 0–4. Differences are integration minus behavior, with 95% intervals from 10,000 resamples of the 60 tasks (checks within a task are not independent draws).

Secondary	Behavior		Integration		Difference	
dimensions	rate	valid	rate	valid	pp	95% interval
0	76.3%	1,218	59.2%	612	−17.1	[−27.8, −6.7]
1	73.7%	1,931	61.3%	1,032	−12.4	[−20.9, −3.6]
2	68.7%	3,118	58.9%	2,730	−9.8	[−16.6, −3.1]
3	63.0%	1,156	55.5%	1,359	−7.5	[−15.5, 0.8]
4	51.2%	80	48.6%	457	−2.7	[−39.8, 28.4]
5	—	0	33.3%	12	—	—

F Optional Visual Feedback: Supplementary Experiment

Related feedback workflows. GameDevBench studies editor screenshots and runtime video as feedback. OpenGame (Jiang et al., 2026) updates debugging guidance from build, test, and runtime outcomes; Play2Code (Huang et al., 2026) couples coding with GUI-agent playtesting. PlayCoder (Peng et al., 2026) uses behavioral feedback for repair and stages execution, test, and interactive-play evaluation. Our workbench comparison examines a narrower configuration change:

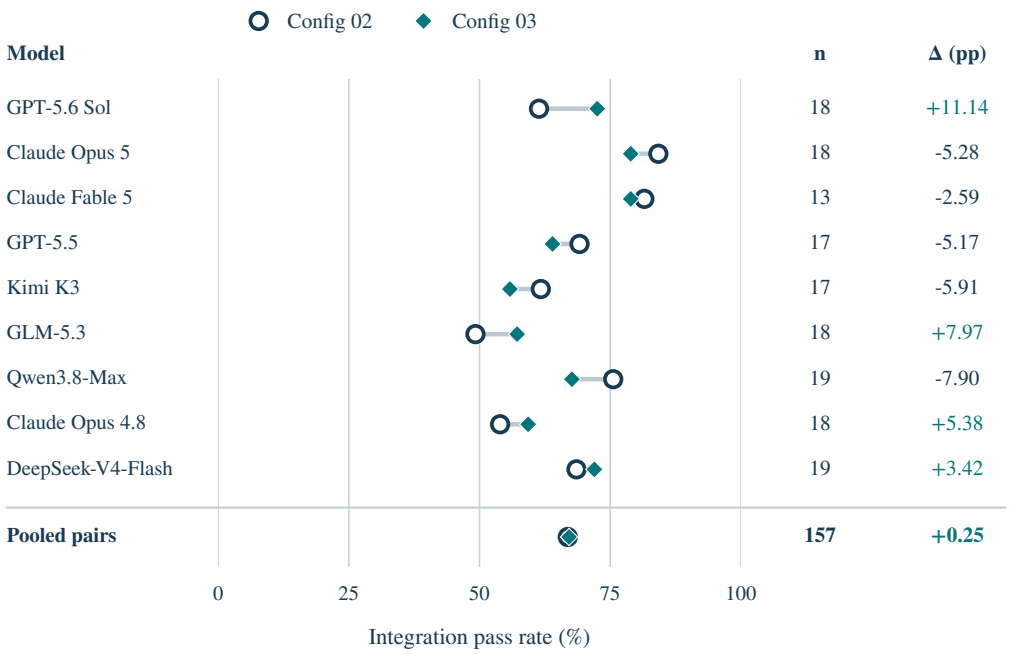


Figure 6: Integration outcomes with structured tools (02) and optional visual feedback (03). Within each model, paired tasks are weighted equally; the pooled row weights task–model pairs equally. n is the number of complete pairs and Δ is 03 minus 02 in percentage points. Because completion subsets differ across models, the rows are not comparable to each other as a ranking, and no confidence intervals are shown.

optional images on top of structured state and action tools, with executable checks retained for final grading. The comparison concerns optional image access within this interface.

F.1 Experiment and descriptive results

A separate experiment compares two workbench configurations on a coverage-selected 20-task subset and nine model identifiers. Configuration 02 exposes structured game-state and action tools. Configuration 03 adds optional screenshots, image messages, and the corresponding instructions; images are not forced into every turn. Both configurations already have a workbench, so this is not a tools-versus-no-tools ablation.

The planned matrix contains 360 runs, with one answer per task, model, and configuration. The frozen workbench snapshot contains 351 scored-file records, but only 157 task–model pairs have complete valid measurements on both sides; this distinction prevents interrupted or empty records from being scored as behavioral failures. We compute each layer's check fraction within an answer and then average equally over complete pairs. This is a paired task–model macro average, unlike the check-pooled profiles above.

Across the 157 pairs, behavior rates change from 73.26% to 75.96%, integration from 66.88% to 67.13%, and the dimension-complete score from 39.87% to 38.21%. Model-level integration differences range from -7.90 to $+11.14$ points (Figure 6); the direction is not uniform across models. Only 16 of the 174 scored-file configuration-03 runs attach an image, totaling 55 image attachments. The low, self-selected uptake limits conclusions about the effect of visual evidence. The following protocol details and sensitivity analyses delimit this comparison.

F.2 Protocol and sensitivity analyses

Controlled resources and differing feedback. Each configuration uses two CPU cores, 2 GB memory, and a 300-second per-command limit, without a total wall-clock or step limit. Configura-

tion 03 permits at most 40 attached images per answer, with longest edge 1,024 pixels and at most 2 MB per image. Audited paired task assets, starter projects, and tool bindings match across all 180 planned pairs, apart from the intentional feedback configuration. The experiment does not establish comparability with historical configuration 01.

Completion accounting. Of the 360 planned runs, 177 configuration-02 and 174 configuration-03 records have scoring files. Nine runs remain interrupted. Among the 351 scoring-file records, 350 have nonempty probes and 328 have complete valid checks. There are 173 pairs with scoring files on both sides, 172 with nonempty probes on both sides, and 157 with complete measurements on both sides. Completed scoring-file records include 343 `Submitted` and eight `RepeatedFormatError` terminal states. The main complete-pair analysis includes format-error terminal states when their final probe is valid; the submitted-only sensitivity restricts both sides to `Submitted`. Interruptions are not replaced with zero. Table 8 recomputes the comparison on each eligibility subset.

Table 8: Sensitivity of workbench integration rates to eligibility. Each row recomputes the mean on the indicated paired subset; subsets change the target population. Δ is configuration 03 minus 02 in percentage points.

Paired subset	Pairs	02 (%)	03 (%)	Δ
Complete measurements	157	66.88	67.13	$+0.25$
Both submitted	150	69.70	69.12	-0.58
No recorded engine errors	151	68.01	68.09	$+0.08$
No recovery in either run	144	65.79	65.89	$+0.10$
10 tasks complete for all nine models	90	75.32	73.89	-1.43

Interpretation. The task subset was selected for coverage, not randomly sampled. There is one run per configuration, no replicated-seed variance estimate, and completion differs by model. Taking images is an agent decision; users of images are not a randomized treatment group. The 55 image attachments in scored-file runs come from only 16 configuration-03 answers; one additional interrupted answer attaches six images and is not included in that denominator. The aggregate paired differences do not establish equivalence, universal improvement, or ineffectiveness of visual feedback. Channel failures and recovery history also limit attribution.

G TASK CONSTRUCTION PIPELINE

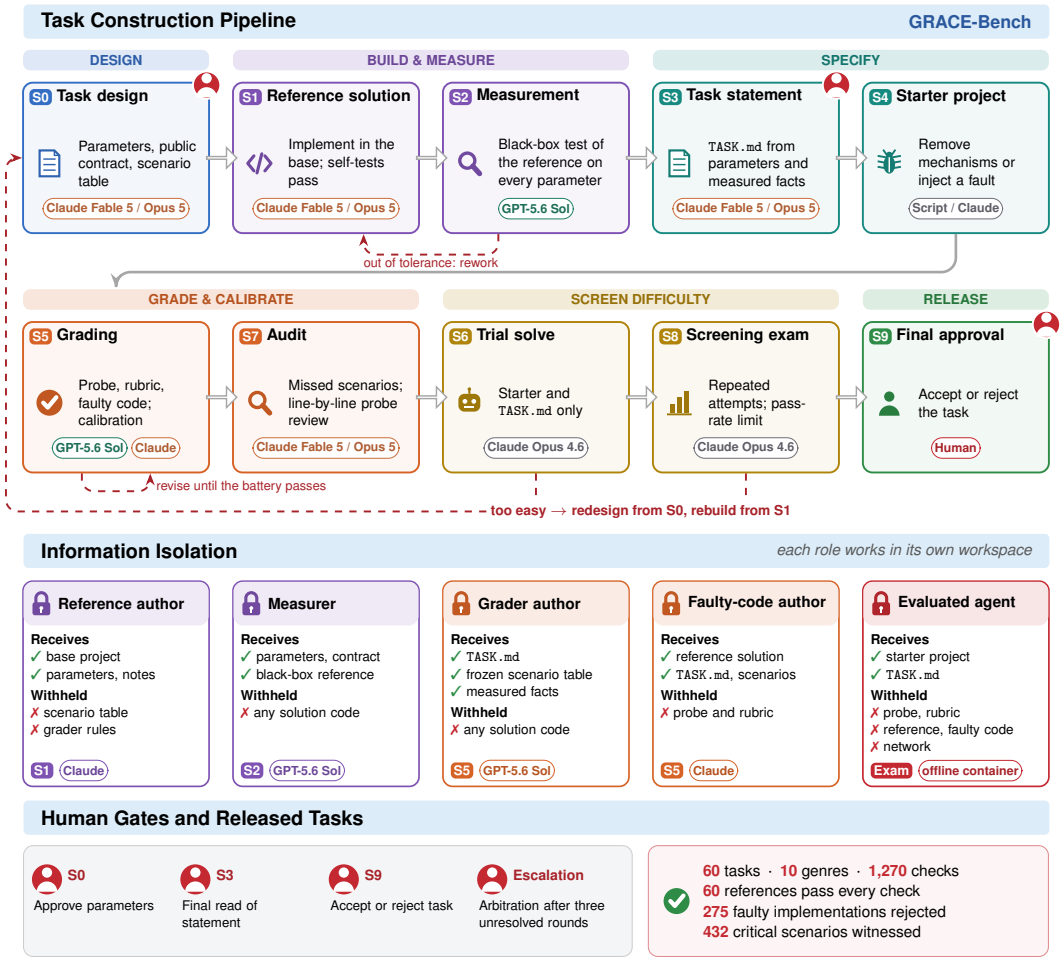


Figure 7: **Task construction pipeline.** Top: stages in execution order. Stage numbers follow the production records, so the audit (S7) runs before the trial solve (S6). Chips name the model used in each role, and red badges mark human gates. Dashed arrows are revision loops: measured mismatches return to S1, calibration failures repeat S5, and tasks that screening finds too easy go back to S0. Middle: what each construction role receives and what it is denied. Bottom: human gates and summary counts for the released tasks.

Each task passes through ten stages, S0–S9, shown in Figure 7. A separate session with a fixed role instruction and its own workspace carries out each stage. Every stage leaves files on disk, and later stages read those files rather than earlier conversations. A scheduler advances a task only when the current stage's machine checks pass. Failures return the task to the responsible stage, and disagreements that stay unresolved after three rounds go to a human.

Base projects. Tasks start from public Godot 4 2D projects written in GDScript. A project qualifies only if its code carries a permissive license (MIT, Apache-2.0, or CC0), its assets are checked separately, it has roughly 10 to 100 script files, and at least three runtime systems in it can interact. We screened 94 repositories and rated 37 as fully suitable. Popularity was not a criterion. A widely copied project is more likely to appear in training data, so its tasks require larger behavioral changes. Every task changes the base behavior, and the design records each parameter's original and modified value.

Design, reference, and measurement (S0–S2). In S0 a designer writes a parameter file. Each measurable parameter records six fields:

- the value;
- the tolerance (zero for exact quantities);
- the start and end anchors of the measurement;
- what is observed;
- how to report a measurement that could not be taken.

The file also contains a public contract stating how the tested objects are reached, driven, and observed. A scenario table lists at least twelve scenarios in eight categories, and at least 40% of them are boundary cases; 58 of the 60 tasks were built under this scenario rule. A schema check sends incomplete drafts back to the designer. A human approves the design before building starts. In S1 a developer implements the reference solution in the base project. The grading workspace is built by the same function as this development workspace. In S2 a measurer first submits a measurement plan that names, for each parameter, how it will be driven, where sampling starts and stops, and how failures will be reported. The plan is reviewed against the parameter file before any measurement runs. The measurer then tests the running reference as a black box. Each parameter is marked as a match, a mismatch, or not measured. Mismatches return to S1 for rework. Failed measurements return to the measurer, because a failure to measure is not evidence that the behavior is wrong.

Statement and starter (S3–S4). S3 drafts `TASK.md` from the parameter file and the measured behavior. The statement describes the required behavior, but it does not name scenarios or checks. If the inputs to the statement are unchanged, the approved statement is reused. If they change, a new draft must come with a diff against the previous one. A human reads the final statement before it is frozen. S4 builds the starter project. For extension tasks, it removes the target mechanisms from the reference. For repair tasks, a separate author injects a fault that changes at most ten lines in one to three existing files and adds no new files.

Grading and calibration (S5, S7). In S5 a grader author writes the probe and rubric using only `TASK.md`, the frozen scenario table, and the measured facts. The rubric maps every check to scenarios and covers every scenario. A separate author writes faulty implementations that each change one to three lines of the reference. Each comes with a matrix that pairs it with the boundaries the statement emphasizes. Critical scenarios also need targeted witness implementations. A task passes calibration only when all of the following hold:

- the reference passes every check;
- the starter fails the checks it targets;
- each faulty implementation fails a check mapped to its scenario;
- a rerun of the reference agrees check by check;
- any alternative reference also passes every check.

Section 3.4 reports the resulting counts. S7 then audits the calibrated grader. It first searches for scenarios the table missed, then reviews the probe line by line against the statement. Any change to the probe reruns S5 calibration.

Difficulty screening (S6, S8). After calibration, a task is screened for difficulty with Claude Opus 4.6, which is not among the evaluated models. S6 is a single trial solve under exam conditions: only the starter project and `TASK.md`, with grading applied after the session ends. S8 repeats the attempt several times and compares the task-level pass rate with a preset limit. A task is returned if the screening model solves it too often or comes close to a full pass. It is not patched by adding checks. It goes back to S0, where the design of what makes it hard is changed: which mechanisms are removed, which fault is injected, or which systems are coupled. It is then rebuilt from S1 onward. Screening runs are not part of the reported results. Because tasks were filtered against one model, the task set may be harder for that model and similar models than for others.

Information isolation. Each role sees only the files it needs (Figure 7, middle):

- The reference author cannot read the scenario table or the grader rules.

- The measurer and the grader author receive no solution code.
- The authors of faulty implementations do not see the probe or rubric.

Solutions and grading tools also come from different model families. Claude models write the references and faulty implementations, while GPT-5.6 Sol writes the measurements, probes, and rubrics. As a result, a reference solution and the checks that accept it always come from different model families. During construction, isolation comes from each workspace's contents and from the role instructions; there is no operating-system sandbox. In the evaluation, agents run in network-disabled containers that hold only the starter project and `TASK.md`, and the probe is installed after the session ends.

Human role. The pipeline requires human approval at three gates: the S0 parameter design, the final reading of `TASK.md` in S3, and final acceptance in S9. Humans also arbitrate escalated disagreements. Everything else is automated. The web workbench exposes each gate as an explicit action and records every approval in an audit log.

Construction models and exam performance. Most task designs, reference solutions, statements, faulty implementations, and audits were produced by Claude Fable 5. Claude Opus 5 took over these roles after a model switch partway through construction. GPT-5.6 Sol produced the measurements, probes, and rubrics. For five tasks, the reference was finished in separate Claude Code maintenance sessions. All three construction models also appear in the evaluation. Having built a task does not mean a model can solve it in the exam setting. During construction the models had the full design documents, no time limit, and repeated rounds of feedback from measurement, calibration, and review; the construction logs record more than 8,000 Claude sessions across the 60 tasks. In the evaluation, each answer comes from a single session that sees only the starter project and `TASK.md` and gets no feedback from the checks.

H Task Roster and Check Allocation

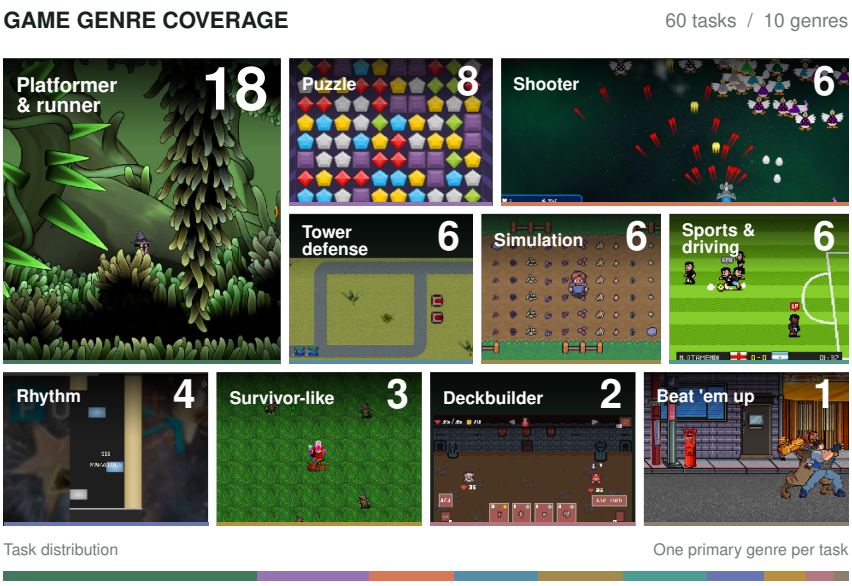


Figure 8: **Task distribution across ten primary game genres.** Numbers give task counts. The bottom strip shows task proportions using the colors beneath the screenshots. Each task has one primary genre, based on its game template. Screenshots come from project authors or local gameplay captures; their sizes do not encode frequency.

Table 9 lists the 60 evaluated tasks. G/B/I denote the gate, behavior, and integration check counts. The # column is the task number used throughout this paper (1–60); the monospace token is the task's archival identifier.

Table 9: The 60 evaluated tasks and 1,270 rubric checks.

#	Task identifier	G	B	I
1	100_cornucobia_run_scaling	3	12	12
2	102_farmsim_herd_rhythm	2	6	15
3	103_survivors_lantern_light	2	8	7
4	104_portalgolf_aim_visuals	2	11	8
5	107_beatemup_gang_ai	1	7	7
6	112_breadgirl_hazard_lifecycle	1	6	14
7	120_exponaut_jump_windows	1	13	5
8	121_superbread_checkpoint_lifecycle	1	6	9
9	122_farming2d_irrigation_flow	1	13	6
10	125_supersoccer_stamina_subs	2	12	4
11	14_ant_colony_systems	1	8	9
12	15_spellweavers_combat	2	13	8
13	18_lightbound_navigation	1	5	13
14	19_tower_defense_systems	1	12	10
15	202_farmingsim_bulletin_visuals	3	11	10
16	205_asteroids_split_momentum	1	8	11
17	210_hwtycoon_staff_lifecycle	3	7	12
18	211_survivors_weapon_evolve_states	2	13	5
19	212_ovomania_judgement_visuals	3	11	7
20	21_deckbuilder_synergies	2	14	7
21	21_wave_survivor_tactics	2	14	8
22	22_metroidvania_progression	2	18	12
23	29_duel_defense_targeting	3	21	4
24	30_sokoban_warehouse_state	1	13	4
25	31_space_massacre_lifecycle	1	9	5
26	32_tetris_visual_feedback	1	20	1
27	33_flick_physics_mastery	2	10	5
28	43_modplatformer_rule_zones	1	19	19
29	45_olesa_inventory_surfaces	3	14	10
30	46_ovo_mania_ui_flow	2	13	11
31	48_bread_girl_powerup_states	1	9	9
32	51_towerdef_economy_upkeep	1	10	7
33	53_spellweavers_wardcraft	1	10	15
34	54_space_massacre_hit_feedback	1	6	8
35	56_tiny_wizard_room_tactics	1	9	13
36	57_super_soccer_match_flow	1	4	7
37	58_duel_defense_unit_lifecycle	2	8	7
38	59_ovo_mania_note_timing	2	7	8
39	60_tetris_gravity_curve	1	10	7
40	61_metroidvania_damage_visuals	3	8	16
41	63_contra_squad_ai	2	8	7
42	64_portal_golf_turn_rules	1	11	3
43	66_crystal_dash_timer_coupling	1	4	10
44	67_exponaut_score_curve	1	8	11
45	68_speedrun_hud_feedback	1	10	5
46	70_numbercluck_flock_ai	2	11	7
47	73_flick_launch_timing	2	13	12
48	74_hill_climb_pickup_lifecycle	1	11	8
49	76_ant_colony_caste_ai	2	15	4
50	80_blokki_clear_visuals	2	10	9
51	82_olesa_entity_lifecycle	2	11	10
52	84_modplatformer_pursuit_ai	3	8	14
53	86_metroidvania_dash_physics	2	12	5
54	87_spellweavers_motion_physics	1	4	8
55	88_deckbuilder_intent_ai	1	8	11
56	93_duel_defense_status_visuals	1	7	12

Continued on next page

#	Task identifier	G	B	I
57	`94_towerdef_missile_lock_visuals`	1	10	16
58	`99_speedrun_alert_network`	1	7	6
59	`L100_75x83_research_hints`	4	34	13
60	`L500_91x92x77_ritual_disclosure_kinematics`	2	12	2
	Total	100	642	528

I Comparison with Related Evaluation Protocols

Table 10 separates the artifact under evaluation from the evidence and unit of judgment. Runtime interaction, regression testing, state access, and diagnostic decomposition occur in prior work; none is an exclusive property of GRACE-Bench. Our analysis combines fixed within-task layer labels with prerequisite-conditioned integration outcomes and raw measurement validity. The comparison describes published protocols, not an experiment running competing evaluators on the same submissions.

Table 10: Protocol comparison with close neighbors. Entries describe reported methods, not exhaustive capabilities or a shared performance ranking.

Work	Evaluated artifact	Grading evidence and diagnostic unit
FeatureBench (Zhou et al., 2026)	Incremental feature development and from-scratch reconstruction.	Fail-to-Pass and Pass-to-Pass tests, with resolved and partial-correctness metrics.
GameDevBench (Chi et al., 2026)	Tutorial-derived Godot project tasks.	Deterministic tests, including runtime interactions; verified references and human task review.
GameCraft-Bench (Luo et al., 2026)	Complete Godot projects.	Build gate, replay of submitted interaction traces, and multimodal rubric scores for mechanics, content, functional visuals, and presentation.
WebGameBench (Zhang et al., 2026)	Deployed browser games from structured specifications.	Browser-agent pass/fail/unverified evidence supports delivery labels; evaluator/infrastructure-invalid records are excluded from delivery-rate denominators.
GameXpert-Bench (Chen et al., 2026)	Generation, web-game repair, and multi-turn optimization tracks.	GameFix: deterministic bug-site F2P/P2P checks. GameGen: implementation/runtime diagnostics. GameOpt: requirement, integration/edge-case, and regression items.
WorldCoder-Bench (Lu et al., 2026)	Generated Three.js worlds.	Hidden, mutation-hardened runtime contracts; affordance/state/transition coverage and the fraction of hidden assertions passing (V-Cov).
GameGen-Verifier (Jia et al., 2026)	Verification of generated web-stack games.	Injected states, bounded actions, and mixed VLM/assertion verdicts; specification-level failure propagation. Failed state setup can also yield a fail verdict.
GRACE-Bench	Extension and repair of existing Godot 2D projects.	Fixed executable checks with gate/behavior/integration labels and seven diagnostic dimensions; conditional integration outcomes retain valid pass, measured failure, and unresolved status.

Engine execution does not determine the oracle. AutoUE (Yin et al., 2026) generates Unreal Engine games and collects executed interactions, screenshots, and logs for model-based evaluation. In contrast, JamBench, introduced in JAMER (Sun et al., 2026), uses offline LLM-generated configuration followed by rule-driven input and structural/runtime-behavior scoring without LLM verdicts. Its behavior score compares activity with reference statistics rather than proving every specified gameplay rule. These protocols illustrate why engine choice, runtime evidence, and final decision rule must be described separately. GRACE-Bench's optional image feedback belongs to the solving interface; it does not replace its executable grading checks with a visual judge.

Scope of the comparison. A delivery label, repaired bug site, rubric score, and passed assertion are different units. Tasks, interfaces, budgets, and aggregation also differ, so external scores cannot establish relative difficulty.