LAYERED GAME ENGINE: A COMPOSITIONAL ARCHITECTURE FOR EXECUTABLE GAME WORLDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Game engines distribute world representation, simulation, mechanics, execution order, and presentation across interacting subsystems. We study whether the computational structure of an interactive world can itself be represented explicitly as executable data. We present Layered Game Engine (LGE), a proof-of-concept architecture in which behavior is represented by first-class rules with explicit stage, scope, dependencies, and executable bodies. World state, spatial organization, simulation, projection, and visual processing are separated into computational layers, while a compiled dependency graph determines execution order independently of rule registration order. LGE additionally supports selective execution: an observation identifies a focal entity and a bounded spatial horizon, from which the engine derives relevant regions, entities, surfaces, and dependency-closed rules. A C++20 prototype implements generated 2D and 3D worlds, physics, collision, surface dynamics, material-driven visual behavior, parallel execution, deterministic state and visual hashes, and a headless harness. In the supplied benchmark, a fixed active set of 64 entities, one region, and seven rules is evaluated inside worlds of up to 100,000 entities. At 100,000 entities, cached selective execution reduces median run time from 33.50 ms/tick for full LGE to 7.61 ms/tick. A simple C++ object-update control requires 0.114 ms/tick. We therefore treat the result as an architectural proof of concept rather than a performance-superiority claim.

1 INTRODUCTION

Interactive environments increasingly serve as programmable computational worlds. Reinforcement-learning environments standardize agent interaction, while large multi-agent and procedural environments expose increasingly complex dynamics. Yet the internal computational organization of an environment is usually treated as implementation detail: physics, collision, update systems, rendering, and application logic are arranged procedurally, with dependencies often encoded implicitly in code structure.

This paper asks a narrower architectural question: *can the executable structure of an interactive world be represented explicitly enough to be inspected, reordered, selectively executed, and reused without changing the underlying rule implementations?* The objective is not to replace mature engines or claim that a generic rule system is faster than specialized code. Instead, we make execution structure a first-class representation and measure the consequences.

LGE represents behavior using executable rules. Each rule records a stage, scope, enabled state, dependencies, and an executable body. Stages answer when computation occurs; scopes answer what class of world object it operates over; dependencies define prerequisites. Spatial regions form a separate graph for locality. Projection converts semantic state into a generic VisualField before visual rules operate.

Selective execution separates *what is relevant* from *what is executed*. An observation names a focal entity and finite spatial horizon. The planner derives an active region neighborhood, collects its entities and surfaces, selects rules by scope, and closes the result over rule dependencies. Full and selective simulation use the same state representation and rule bodies.

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.

1.1 Contributions

We make five contributions:

- An executable layered representation in which stage, scope, and dependency are explicit data.
- A selective execution mechanism that derives a bounded spatial and computational set from an observation.
- A VisualField abstraction separating semantic world state from visual processing.
- A C++20 prototype with deterministic state/visual hashing and rule-registration-order invariance tests.
- A preliminary scaling study exposing both the benefit and the substantial overhead of explicit selective execution.

2 Positioning and Problem Definition

2.1 Explicit execution

Let S_t be world state, I_t input, and R an ordered collection of computations. Full execution is

$$S_{t+1} = F(S_t, R, I_t). \tag{1}$$

The research object is the representation of R. A rule contains metadata plus an executable body, allowing the engine to compile a dependency graph once at load time. Execution order therefore does not depend on insertion order.

2.2 Stage, scope, and spatial structure

The prototype defines five scopes—Global, Region, Entity, Surface, and Visual—and seven stages—World, Physics, Surface, Collision, Projection, Visual, and Feedback. Stage is temporal structure; scope is semantic reach; the region graph is spatial structure. Keeping these orthogonal prevents the spatial partition from becoming the rule hierarchy.

2.3 Selective execution

For observation $O = (e, h)$, with focal entity e and region-hop horizon h, the planner constructs an active spatial set A_{spatial}. The selected rules are the dependency closure of rules whose scopes intersect that set, together with global rules:

$$R_A = \text{closure}_{G_R}(R_{\text{scope}}(A_{\text{spatial}}))\,. \tag{2}$$

The selective transition is

$$S'_{t+1} = F(S_t, R_A, I_t). \tag{3}$$

This is an execution contract, not a claim that omitted work is inherently independent. The declared dependency graph determines prerequisites; global rules remain eligible.

2.4 Relationship to ECS

LGE shares data-oriented motivations with entity-component systems: entities are records rather than renderer-object hierarchies. The distinction is the unit of explicitness. LGE makes stage, scope, and dependencies part of the executable rule record and keeps the compiled dependency graph as engine state. This makes execution structure directly available to selective planning.

3 System Architecture

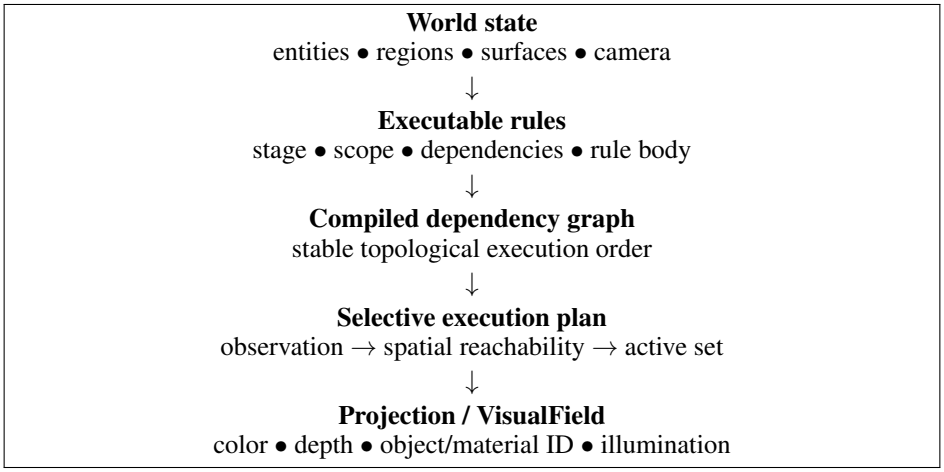

Figure 1: LGE separates world state, executable rules, dependency compilation, spatial execution selection, and visual processing.

3.1 World and rule representation

Entities contain identity, position and velocity, material information, animation state, gravity multiplier, and optional surface dynamics. Regions are axis-aligned boxes with a neighbor graph. GameDefinition stores entities, lights, regions, physics/camera/visual configuration, and the RuleSet. Rules contain identifier, stage, scope, enabled state, dependencies, and executable function.

3.2 Dependency compilation

Enabled rules are compiled into a stage-aware dependency graph using stable topological ordering. Dependency cycles are invalid definitions rather than silently resolved by registration order. This makes the ordering contract inspectable and testable.

3.3 Compositional behavior

SurfaceDynamics is a generic height-field mechanism parameterized by stiffness, damping, and interaction characteristics. The prototype uses it for water and jelly-like behavior without a dedicated simulation type for each phenomenon. Emissive and reflective bodies are expressed through material-driven visual rules; floating behavior uses a gravity multiplier.

3.4 VisualField

After projection, VisualField stores generic observable channels: color, depth, object identity, material identity, and illumination. Visual rules consume this representation rather than coupling directly to semantic world objects. The same abstraction is used for generated 2D and 3D worlds.

4 Execution Model

4.1 Full and selective paths

Full simulation rebuilds region membership, constructs RuleContext, and invokes the compiled rule graph for World, Physics, Surface, Collision, and Feedback. Selective simulation attaches an ExecutionSet and an active-rule list to the same rule context. The rule bodies are shared; only selected work changes.

4.2 Observation to ExecutionPlan

The planner maps the focal entity to its containing region, performs deterministic breadth-first expansion over neighboring regions to the requested horizon, collects active entities and surfaces, selects rules by scope, and closes the selection over dependencies. Active identifiers are sorted to preserve deterministic behavior.

4.3 Parallelism

The engine uses a persistent worker pool with disjoint work ranges. This avoids thread creation inside the frame path while preserving deterministic state/result semantics. The worker pool is part of the prototype substrate, not presented as a novel parallel scheduling algorithm.

4.4 Determinism

The headless harness executes identical generated games twice with a pure scripted input function and compares state hashes at 60, 240, and 600 ticks. It separately hashes the projected VisualField. A registration-order test reverses the RuleSet while preserving dependencies and checks that state and visual hashes remain unchanged.

5 Prototype Implementation

The implementation is C++20 with a platform-independent engine core. The Windows GUI build uses Win32/GDI only for presentation; a headless build drives the same engine without a window. Loading a GameDefinition builds the region graph, material identifiers, surface state, rule set, and compiled dependency graph once. GameState stores definition, entities, surfaces, time, camera, compiled rules, region graph, entity index, and entity-to-region mapping.

The prototype is deliberately a proof of concept. It contains sufficient simulation and rendering structure to exercise the layered representation, but it is not a production engine. This distinction becomes important in the performance results.

6 Evaluation Methodology

6.1 Research questions

1. RQ1: Can distinct environment behaviors be expressed through shared computational mechanisms?
2. RQ2: Can an observation produce a bounded active execution set while preserving the same rule bodies and state representation?
3. RQ3: How do full, selective, and cached-selective costs change as world population increases while the active set remains fixed?
4. RQ4: What execution overhead is introduced by the explicit layered representation relative to simple low-level C++ workloads?

6.2 Protocol

The supplied benchmark executes 120 ticks per repetition and reports the median of five repetitions. Populations are 1,000, 10,000, 100,000, and 500,000 entities. At 1K, 10K, and 100K, the selective plan contains 64 active entities, one region, and seven rules. Full LGE, selective LGE, and cached selective LGE are measured.

6.3 Measurement scope

The terminal output does not record CPU model, compiler version, optimization flags, operating-system build, or thermal state. We therefore treat the measurements as an internal prototype scaling study, not a cross-machine ranking. The naive-collision values are near zero in the supplied output

and are not used as evidence because the workload is insufficiently documented. LGE values for 500K were not reported and are not extrapolated.

6.4 Cross-engine comparison protocol

The accompanying benchmark artifacts also contain measurements for a plain C++ object-update control, Bevy ECS 0.19, and an existing Godot Node2D/SceneTree benchmark. These workloads use the same reported entity populations, 120 ticks, and five repetitions, but they are not feature-matched implementations of a complete game engine. We therefore use them as common-workload context rather than as a universal engine ranking. Godot was not rerun for this comparison; its existing benchmark artifact is retained as supplied.

The principal cross-system quantity is median wall time per tick. For LGE, we report the latest supplied terminal run; for Bevy and Godot, we report the supplied benchmark artifacts. Because the measurements are not a single synchronized rerun, the comparison is interpreted descriptively rather than as a controlled head-to-head study.

6.5 Main LGE results

Table 1: Supplied LGE benchmark. Active is entities / regions / rules; times are median milliseconds for the complete 120-tick run.

Entities	Full	Selective	Cached selective	Active
1,000	33.7891	18.9063	12.1359	64/1/7
10,000	256.6832	147.6982	79.5608	64/1/7
100,000	4019.5842	1796.8349	912.7326	64/1/7
500,000	–	–	–	–

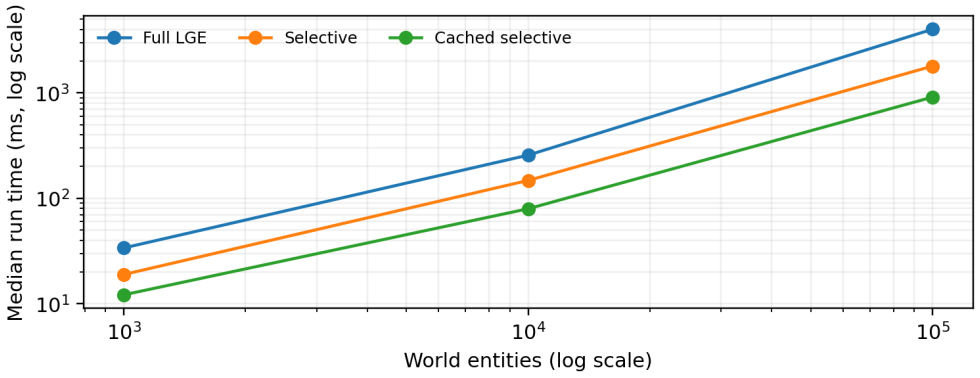

Figure 2: Full, selective, and cached-selective LGE scaling. Both axes are logarithmic.

6.6 Low-level baseline context

Table 2: Low-level C++ controls from the supplied terminal output. They are not complete alternative game-engine implementations.

Entities	Array update	C++ objects	Spatial grid
1,000	0.1130	0.1081	0.0070
10,000	1.7979	1.2352	0.0075
100,000	56.5677	13.6981	0.0074
500,000	80.7673	89.0684	0.0080

7 RESULTS

7.1 SCALING AND SELECTIVE EXECUTION

At 100,000 entities, only 64 entities are active, or 0.064% of the world population. Cached selective execution is 2.78×, 3.23×, and 4.40× faster than full LGE at 1K, 10K, and 100K respectively. The growing reduction indicates that selective execution eliminates some work as the world grows.

The stronger property—population-independent cost for a fixed local observation—does not hold. Cached selective execution rises from 12.1359 ms per 120-tick run at 1K to 912.7326 ms at 100K. Semantic locality is therefore not yet computational locality.

At 100K, cached selective LGE requires 7.606105 ms/tick, while C++ object updates require 0.114151 ms/tick. The prototype therefore incurs substantial overhead. We do not interpret the benchmark as evidence that LGE is faster than conventional engines.

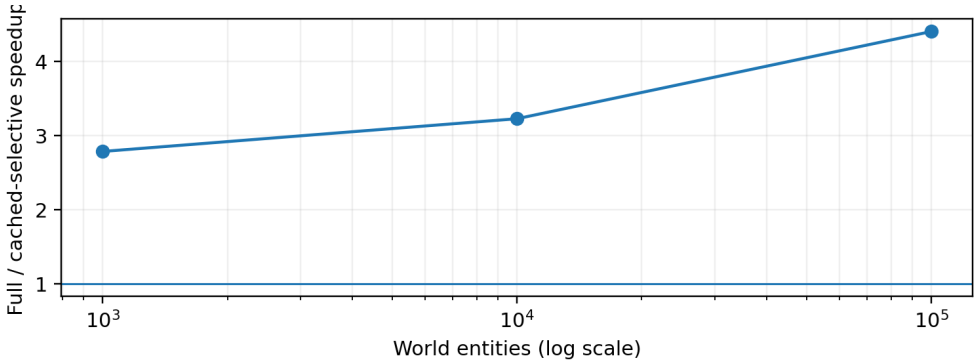

Figure 3: Full-to-cached-selective speedup computed from the supplied medians.

7.2 COMMON-WORKLOAD COMPARISON

Table 3 places the LGE measurements beside the supplied Bevy ECS, Godot Node2D, and plain C++ object-update workloads. The comparison is intentionally narrow: each entry is the reported median wall time per tick for an entity-update benchmark, not end-to-end game-engine performance.

Table 3: Common-workload comparison from the supplied benchmark artifacts. Values are median milliseconds per tick. The LGE values use the latest supplied terminal run; Bevy, Godot, and C++ values use the existing benchmark artifacts.

Entities	LGE full	LGE cached	Bevy ECS	Godot Node2D
1,000	0.2816	0.1011	0.0015	0.3114
10,000	2.1390	0.6630	0.0127	4.7407
100,000	33.4965	7.6061	0.1242	58.0917

The plain C++ object-update control requires 0.000693, 0.007795, and 0.066613 ms/tick at 1K, 10K, and 100K entities in its supplied artifact. Bevy ECS requires 0.001518, 0.012708, and 0.124180 ms/tick, while the Godot Node2D benchmark requires 0.311392, 4.740683, and 58.091692 ms/tick. At 100K entities, full LGE is 33.4965 ms/tick and cached selective LGE is 7.6061 ms/tick.

These measurements illustrate the trade-off exposed by the prototype. LGE carries substantially more work than the minimal C++ and Bevy entity-update workloads, while the supplied Godot Node2D benchmark is itself more expensive on the same nominal entity counts. None of these observations establishes comparative engine quality because the workloads expose different abstraction layers, feature sets, and runtime costs. The useful result for this paper is that explicit layered execution has a measurable cost, and selective execution reduces that cost within LGE.

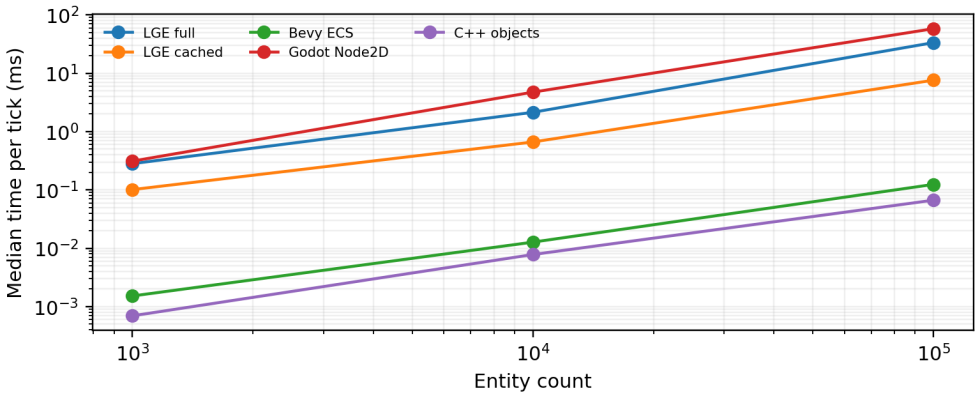

Figure 4: Cross-system common-workload context at 1K, 10K, and 100K entities. Values are median milliseconds per tick; the y-axis is logarithmic.

7.3 Compositionality

The prototype demonstrates shared mechanisms: water and jelly-like behavior use the same SurfaceDynamics primitive; emissive and reflective responses are material-driven; floating uses a gravity multiplier. These examples establish expressiveness, not lower developer effort. Modification cost has not been measured in a controlled study.

8 Determinism and Architectural Invariants

8.1 State and visual hashes

Two independent runs with identical definitions and scripted inputs produce state hashes at multiple horizons. The render chain is independently checked by hashing VisualField color, finite depth, object identifiers, and material identifiers. These are regression signals for semantic and visual execution.

8.2 Registration-order invariance

The rule set is reversed while declared dependencies remain unchanged. State and VisualField hashes must remain identical to the reference run. This test targets the architectural claim that dependency metadata, rather than insertion order, defines execution order.

8.3 Scope of evidence

Hash equality is an implementation invariant, not a formal proof for all compiler or hardware configurations. A stronger study should include randomized rule-registration permutations, multiple worker counts, longer horizons, and multiple toolchains.

9 Discussion

9.1 Semantic versus computational locality

The central result is the gap between semantic active-set size and executed work. At 100K entities, only 64 are selected, yet cached selective execution remains 7.61 ms/tick. The architecture can represent locality before the implementation fully realizes it.

9.2 Likely overhead sources

The selective path rebuilds region membership and performs plan construction, entity/surface collection, scope filtering, dependency closure, rule dispatch, and simulation. Some rule bodies may still traverse globally represented structures. Persistent region membership, incremental spatial indices, compact active arrays, scope-specialized dispatch, precompiled dependency-closed plans, and selective projection are direct next experiments.

9.3 What is not yet measured

The current paper does not measure modification time, developer effort, downstream learning performance, or automated reasoning quality. Those are motivations and future evaluation targets. Similarly, selective rendering requires an independent scaling study.

10 Related Work

10.1 Interactive environments

OpenAI Gym and Gymnasium standardize agent-environment interfaces, while DeepMind Lab and Neural MMO expose richer visual or multi-agent environments. Procgen studies generalization over procedurally generated game-like tasks. LGE addresses a different boundary: the internal executable structure of the environment.

10.2 Entity-component architectures

ECS separates entity identity, component data, and systems. LGE shares this data-oriented motivation but makes rule stage, scope, and dependency explicit and uses the resulting graph for selective execution.

10.3 Game engines and structured computation

Production engines provide optimized physics, rendering, resource management, scheduling, and tooling. The benchmark artifacts used here include Bevy ECS and Godot Node2D workloads as concrete ecosystem reference points, but neither is treated as a feature-complete implementation of the LGE workload. LGE is not presented as a replacement for these systems. Its contribution is an experimental representation in which execution structure is inspectable and selectable. More generally, LGE relates to graph-based computation, but combines dependency ordering with orthogonal spatial scope and a common world-to-visual representation.

11 Limitations and Threats to Validity

- **Prototype maturity:** globally coupled operations remain in the selective path.
- **Benchmark metadata:** CPU, compiler, flags, OS, and thermal metadata are absent from the supplied run.
- **Workload representativeness:** the benchmark is synthetic and fixes the active set at 64 entities.
- **Baseline scope:** low-level C++ controls are not equivalent game-engine implementations.
- **Compositionality:** shared mechanisms are demonstrated, but modification effort is unmeasured.
- **Determinism:** hash-based tests do not cover every hardware/compiler configuration.
- **Scale:** 500K LGE measurements are absent and are not inferred.

12 Reproducibility Statement

The prototype is C++20 and includes a platform-independent headless harness, deterministic scripted input, world-state hashing, VisualField hashing, rule-order invariance checks, and execution counters. The current benchmark uses 120 ticks and five repetitions with median reporting. The experimental package should record the exact source revision, compiler and flags, CPU model, OS build, worker configuration, benchmark command line, raw output, and per-stage timing breakdown.

13 Conclusion

LGE investigates whether an interactive world can represent its own executable structure explicitly. The prototype separates stage, scope, dependency, spatial organization, simulation, projection, and visual processing. Rules are first-class executable records; the dependency graph determines ordering independently of registration order; and selective planning converts a localized observation into an active spatial and computational set.

The evaluation gives a deliberately mixed result. The semantic active set remains fixed at 64 entities while the world grows to 100K, and cached selective execution improves from a 2.78× reduction relative to full LGE at 1K to 4.40× at 100K. Yet cached selective execution remains 7.61 ms/tick at 100K, far above the 0.114 ms/tick C++ object-update control. The prototype therefore does not establish performance superiority.

Instead, the measurements identify a concrete research problem: LGE has semantic locality but incomplete computational locality. The next stage is to eliminate globally coupled bookkeeping, profile the remaining costs, and determine whether explicit execution structure can approach the computational behavior implied by its semantic active set.

References

1. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym. arXiv:1606.01540, 2016.
2. Towers, M. et al. Gymnasium: A Standard Interface for Reinforcement Learning Environments. arXiv:2407.17032, 2024.
3. Beattie, C. et al. DeepMind Lab. arXiv:1612.03801, 2016.
4. Suarez, J., Du, Y., Zhu, C., Mordatch, I., and Isola, P. The Neural MMO Platform for Massively Multiagent Research. arXiv:2110.07594, 2021.
5. Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging Procedural Generation to Benchmark Reinforcement Learning. ICML, PMLR 119:2048–2056, 2020.
6. Tasnim, A. and Zhao, T. The Essence of Entity Component System. ACM/SIGAPP Symposium on Applied Computing, 2026.
7. Tasnim, A. and Zhao, T. The Semantics of Entity Component System. ACM SIGAPP Applied Computing Review, 26(2), 2026.

AI USE STATEMENT

In this work, we used generative AI tools for the following purposes, corresponding to the disclosure categories selected in the ICLR 2027 submission form.

- **Aid or polish writing:** AI tools were used to improve wording, organization, clarity, and presentation of manuscript text.
- **Retrieval and discovery:** AI tools were used to assist with finding and organizing related work and submission-policy information.
- **Research ideation or execution:** AI tools were used to discuss the architecture, experimental design, interpretation of benchmark results, and implementation-related reasoning. The prototype implementation and experiments were performed and controlled by the authors.
- **Draft sections:** AI tools were used to draft and restructure portions of the manuscript from author-provided project information, implementation details, and benchmark results.
- **Proving mathematical claims:** AI tools were used to assist with formulation and checking of mathematical expressions. Final claims and correctness remain the authors' responsibility.

We did not use generative AI to generate the synthetic benchmark dataset. AI-assisted text was reviewed against the implementation, benchmark output, and supplied project materials. AI tools were not treated as experimental evidence. We take responsibility for the final content, claims, and artifacts produced with the aid of generative AI.