JOINTLOOM: COMPACT-STATE GPU RECONSTRUCTION FOR EFFICIENT LEARNING-BASED MULTI-AGENT PATH FINDING

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent path finding (MAPF) requires multiple agents to reach their goals without conflicts. While search-based methods have long dominated MAPF, learned models are increasingly used to predict actions or guide search. Their execution and training require evolving joint states to continuously supply structured model inputs, creating a new computational bottleneck between MAPF state evolution and neural computation. To address this challenge, we propose JointLoom, a compact-state GPU reconstruction framework for efficient learning-based MAPF. JointLoom 1) introduces a reconstructible-state contract that preserves complete joint configurations and consistent input semantics, 2) performs on-demand GPU reconstruction with dependency-aware reuse to reduce redundant input construction and transfer, and 3) asynchronously streams compact states to reduce learner waiting during expert-supervised training. Extensive experiments with graph-based MAGAT and token-based MAPF-GPT validate faithful reconstruction across distinct learned representations, with up to $3.79\times$ faster policy execution and $2.08\times$ faster online expert-supervised training. Source code is available at https://anonymous.4open.science/r/JointooM.

1 INTRODUCTION

Multi-agent path finding (MAPF) is a fundamental problem in multi-agent systems, with applications in warehouse robotics and autonomous transportation, where agents must reach individual goals without conflicts (Stern et al., 2020). MAPF has currently been dominated by search-based methods (Okumura, 2023), while recent learning-based methods directly predict agent actions (Li et al., 2021; Andreychuk et al., 2025) or improve classical search (Jain et al., 2026b). Yet the computational pipeline has not evolved accordingly: learned policies repeatedly transform compact MAPF states into structured, policy-specific inputs for inference and training.

Learning-based MAPF follows two paradigms. Policy-based methods replace online search with neural action prediction, operating on structured representations such as communication graphs (Li et al., 2021) or token sequences (Andreychuk et al., 2025), while learning-guided methods maintain classical search and use learned models to improve decision making (Jain et al., 2026b). We focus on policy-based methods, where these policy-specific views must be repeatedly constructed from the joint MAPF state as agents move, even though the underlying joint configuration remains compact. In GPU-resident execution, view construction lies directly on the decision path; in expert-supervised learning, host-side expansion and transfer, compounded by uneven CPU expert supply, can delay learner updates. Our profiling shows that input acquisition accounts for over half of the expert-supervised training loop for MAGAT. This raises a central question: *what representation should be exchanged between MAPF state generation and learned policies?*

This representation boundary poses three challenges. *1) Reconstructibility*: heterogeneous policies consume different views of the same joint state, from communication graphs (Li et al., 2021) to token sequences (Andreychuk et al., 2025), requiring compact states to retain sufficient information for policy-specific reconstruction. *2) Efficient reuse*: although GPU environments accelerate state transitions (Lan et al., 2022), policy views still require repeated construction, and their derived quantities have different dependencies and lifetimes. *3) Producer–consumer decoupling*: actor–learner systems

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

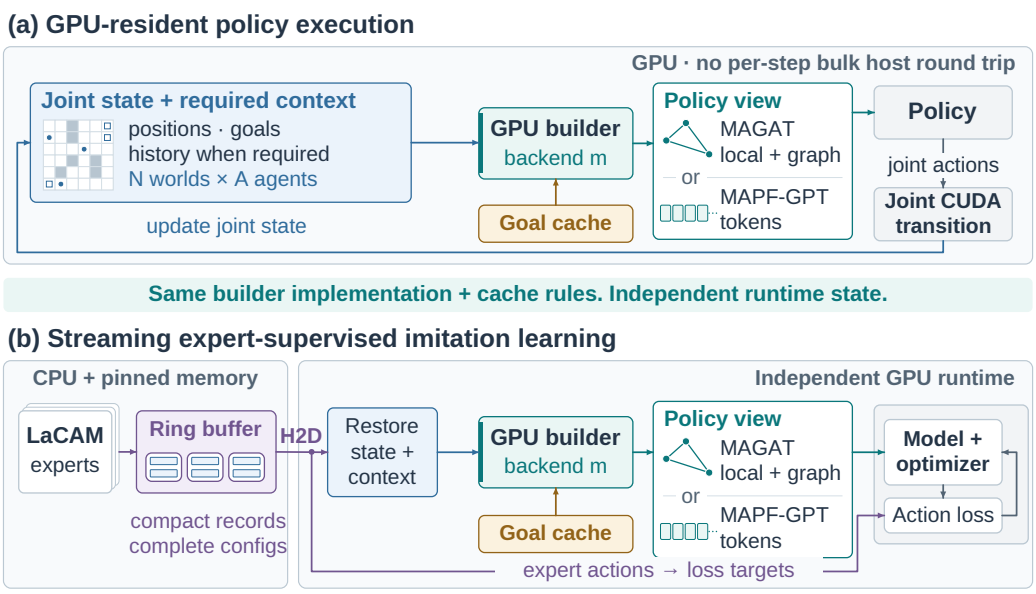

Figure 1: JointLoom applies one reconstructible-state contract to execution and expert-supervised learning.(a) Resident execution reconstructs policy inputs from GPU-resident joint state. (b) Expert training streams compact configurations and action labels from CPU experts and reconstructs policy inputs on GPU. Both workflows reuse the same builders and cache rules.

use buffering to overlap data production and learning (Petrenko et al., 2020; Espeholt et al., 2020), but expert-generated policy inputs still incur construction and delivery costs under uneven CPU supply. These challenges motivate shifting the representation boundary from producer-constructed policy inputs to reconstructible compact states, enabling policy-specific views to be constructed and reused at the GPU consumer across different state-production workflows.

To address these challenges, we propose JointLoom, a GPU reconstruction method that makes compact joint configurations the interface between MAPF state generation and learned policies. As illustrated in Figure 1, JointLoom addresses *reconstructibility* by preserving complete joint configurations and required policy context for on-demand GPU reconstruction of policy-specific views; enables *efficient reuse* by tracking the dependencies of derived quantities; and achieves *producer–consumer decoupling* through buffered compact-state streaming. Together, these mechanisms support both GPU-resident policy execution and expert-supervised learning while preserving joint-state semantics and avoiding redundant input construction and delivery. This design bridges search-based and learning-based MAPF through a common representation boundary: solvers and simulators produce compact joint states, while learned policies construct their required representations on demand at the GPU consumer.

The main contributions are summarized as:

1) A reconstructible-state contract that makes compact joint configurations the exchanged representation between MAPF state generation and learned policies, with explicit completeness, cache dependency, and storage-reuse semantics.

2) We popose JointLoom, a method for GPU-efficient policy-input construction in learned MAPF, combining on-demand reconstruction, dependency-aware caching, and buffered compact-state streaming to unify input construction across policy execution and expert-supervised learning.

3) We extensively evaluate JointLoom across graph- and token-based MAPF policies in both policy execution and expert-supervised learning. At the largest steady GPU-resident MAGAT workload, JointLoom achieves $3.79\times$ faster complete decisions than a GPU-resident JAX/XLA–PyTorch control using the same policy. In expert-supervised learning, it reduces learner waiting and improves training efficiency, translating system-level savings into improved matched-budget learning progress.

2 Background and Compact-State Interfaces

MAPF and the learning task. Standard MAPF uses fixed goals and simultaneous moves subject to vertex and edge-swap constraints (Stern et al., 2020; Skrynnik et al., 2025). A learned policy consumes local features or structured communication inputs derived from the joint configuration (Sartoretti et al., 2019; Li et al., 2021; Wang et al., 2023; Andreychuk et al., 2025).

We study policy execution in simulation and imitation learning from LaCAM expert actions. Both workflows need the same kind of source information: a complete joint configuration and any context required by the policy.

State, context, and model views. We express that shared source contract for policy backend m as

$$b_t^{(m)} = (s_t, h_t^{(m)}), \qquad z_t^{(m)} = \phi_m\left(b_t^{(m)}\right), \tag{1}$$

Here s_t is joint MAPF state, $h_t^{(m)}$ is the policy context beyond that state, and $z_t^{(m)}$ is the model input. The builder ϕ_m produces the *model view*: local features, communication edges, or tokens in the backend's required format. Correct reconstruction includes ordering, masks, and numerical encoding, together with rules for updating and resetting policy context. A simulator state and an expert record that satisfy this specification can use the same builder.

Exposed input delivery. Input preparation can overlap computation. In online training, W is the exposed time from the consumer's request for a complete next stage until its training call begins. For n completed samples in loop time T, throughput $q = n/T$ and wait share $f = W/T$ give

$$t = \frac{1}{q} = w + c, \qquad w = \frac{f}{q}, \quad c = \frac{1-f}{q}. \tag{2}$$

Here w is acquisition wait; c includes compact-path GPU view construction, model/optimizer work, and other costs. Resident decisions start with GPU state and include view construction in their complete-decision timer.

Caches and view lifetimes. Goal-conditioned distance maps illustrate why derived data need a separate lifetime from model views. Distances depend on topology and goals, so they can be reused as agents move; local features and graph edges also depend on positions and must be rebuilt. A model view remains valid until its last consumer finishes, including backward use during training. Only then can the builder overwrite its backing storage for a later call.

Joint inputs. Reconstruction requires a consistent joint snapshot: occupancy, neighbor relations, and graph edges depend on agents together, not isolated rows. The stream therefore delivers all A agents of a selected environment together. This complete configuration is the reconstruction unit; publication stages and optimizer batches are scheduling units assembled from such units (Section 3.2).

Systems context. GPU environments colocate simulation and policies (Shacklett et al., 2021; Makoviychuk et al., 2021; Freeman et al., 2021; Lan et al., 2022); actor–learner systems decouple production from optimization (Petrenko et al., 2020; Espeholt et al., 2020; Mei et al., 2024; Cassirer et al., 2021). In MAPF, both workflows must preserve the joint snapshot used to construct policy inputs. Section 3 turns that requirement into delivery and lifetime rules; Appendices B.2 and H detail backend requirements and related systems.

3 JointLoom: Reconstructible Input Delivery

JointLoom implements Equation 1 through three rules. Producers retain complete joint configurations and required policy context; GPU consumers reconstruct backend-specific views and refresh caches according to their dependencies; publication and storage lifetimes protect each view until its final use. Resident execution applies these rules to simulator state, while expert streaming applies them to restored configurations and buffered stages. Figure 1 shows both paths, and Figure 2 details their reconstruction and handoff.

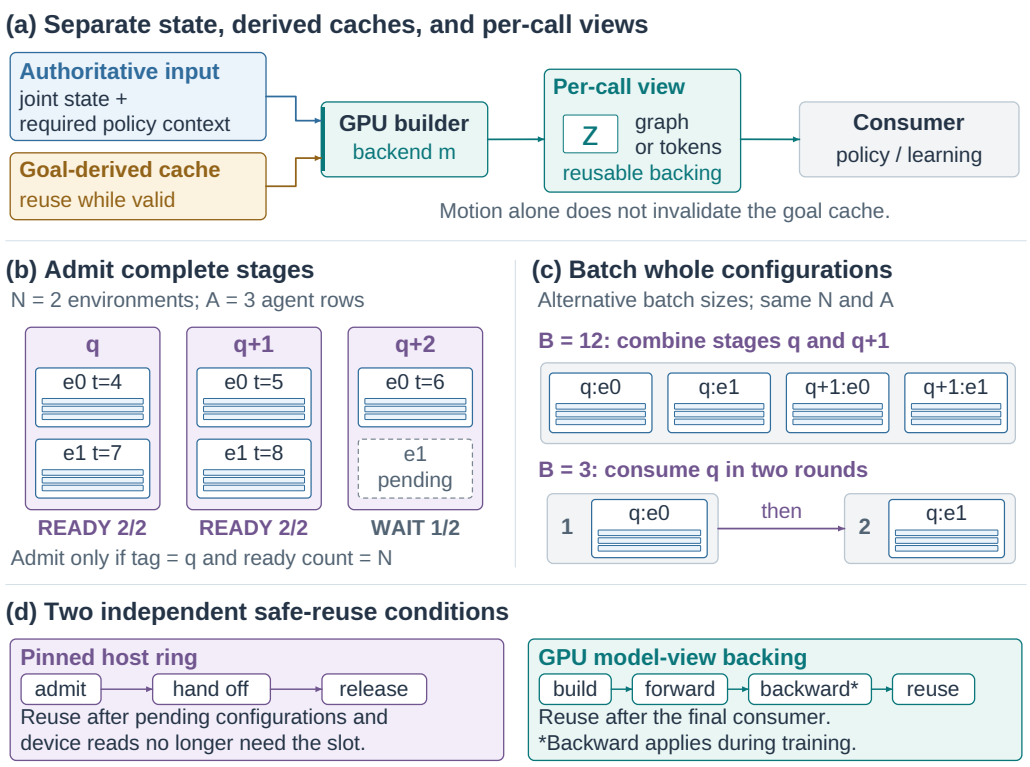

Figure 2: **Reconstruction and handoff contracts.** (a) Authoritative inputs, goal-derived caches, and per-call views have distinct roles; motion alone does not invalidate a goal cache. (b) For $N = 2$ and $A = 3$, each blue tile contains three agent rows and each stage contains two complete configurations. READY/WAIT reports ready configurations out of N; $q + 2$ remains inadmissible. Publication order q is not episode time t. (c) Alternative batch sizes merge two stages ($B = 12$) or consume one stage in two rounds ($B = 3$), without splitting tiles. (d) Ring storage remains protected while configurations or device reads need it; view backing remains protected through the final consumer, including backward during training. The ownership sequences are not synchronized timelines.

3.1 GPU-RESIDENT MAPF SIMULATION

For N environments of A agents each, the resident simulator holds maps, joint positions and goals, occupancy, and episode state on GPU. A joint CUDA transition enforces vertex and edge-swap rules. Assigning one block to each environment permits independent progress and termination; indexing and launch details appear in Appendix B.

The simulator accepts a joint action directly or obtains it from a policy. With a GPU input builder and policy π_θ, one complete decision is

$$b_t^{(m)} \xrightarrow{\text{refresh and } \phi_m} z_t^{(m)} \xrightarrow{\pi_\theta} \hat{a}_t \xrightarrow{\text{joint transition}} s_{t+1}. \tag{3}$$

Before constructing the input, the builder refreshes any goal cache whose dependencies changed. In the MAGAT integration test, the policy selects deterministic argmax actions. It may execute in microbatches, but the transition advances the complete joint batch. The host initializes the run and reads one control byte per measured step; bulk state, input construction, and transitions remain on GPU. Section 5.2 times this complete decision with a fixed policy.

3.2 COMPACT EXPERT STREAMING

CPU LaCAM workers produce complete joint configurations and expert actions along solved trajectories. Instead of host-expanded model inputs, the stream carries compact records to a bounded

pinned-memory ring. MAGAT records use 16 bytes for identifiers, current and goal coordinates, an action label, and a refresh flag; MAPF-GPT additionally carries its required history (Appendix B.6). GPU *state replay* restores recorded state and context rather than re-executing expert actions. The labels supervise cross-entropy training.

Publication and batching. Figure 2(b–c) distinguishes the A-row reconstruction unit, NA-row stage-admission unit, and B-row optimizer batch. Each stage contains one complete A-row configuration from every environment. It is admissible only when its sequence tag matches the expected publication sequence and all N environment sub-blocks are ready. Optimizer batches contain whole configurations: a larger batch may span successive admitted stages, while a smaller batch consumes a stage in several rounds. Sequence q indexes publication order, not a common episode clock.

Buffered supply and safe reuse. Ring depth D permits bounded write-ahead; ordered stage consumption preserves every environment's contribution while smoothing temporary supply variation. Write-ahead gives the learner ready complete stages at batch boundaries, reducing the exposed wait w under uneven expert runtimes. Incomplete stages remain invisible. Figure 2(d) separates host-ring storage from GPU model-view backing: the former cannot be reused while pending configurations or device reads need it, and the latter remains valid through its final consumer, including backward use. Detected worker failures trigger fail-stop cleanup. Appendix B.7 specifies the lifecycle; Section 5.3 tests ring depth separately from the transfer trigger.

3.3 Shared builders and storage lifetimes

Figure 2(a) distinguishes authoritative state and required policy context from derived goal caches and per-call model views. Once state is resident or restored, both workflows invoke the same backend-specific builder. MAGAT constructs local features, communication edges, and relative positions using joint state and goal-derived energy maps (Li et al., 2021; Fey & Lenssen, 2019). Motion alone does not invalidate a goal cache; goal changes or replay refresh flags trigger updates, and topology replacement requires reinitialization. MAPF-GPT instead constructs tokens and maintains its required action history (Appendices B.5 and C.4).

The workflows allocate independent mutable state. As Figure 2(d) shows, model-view contents remain valid through their final consumer; backing storage is reused only afterward. Policy microbatching reduces transient model storage. Appendix G quantifies the resulting allocation behavior, and Appendix B develops layout options for heterogeneous teams.

Using the shared interface. The resulting interface has two entry points. Policy execution accepts maps, initial configurations, and a policy compatible with an implemented builder. Expert training accepts complete configurations, action labels, and model/optimizer settings. New observation, communication, or history specifications connect through backend builders and reconstruction checks. Table 4 relates each entry point to its inputs, components, outputs, and evaluation evidence. The experimental protocol next tests semantic agreement, complete decisions, and delivery to the learner.

4 Experimental Protocol

The experiments test three consequences of the shared interface: semantic fidelity, faster complete resident decisions, and less input wait during online training. Mechanism controls isolate reconstruction and buffering; matched-budget curves show their effect on learning progress. In online controls, W covers exposed source, queue, prefetch, and handoff delay until the consumer enters its training call; compact GPU view construction belongs to c in Equation 2. For JointLoom (J) against cached CPU prefetch (C), the reduction in acquisition wait per sample is $1 - w_{\mathrm{J}}/w_{\mathrm{C}} = 1 - (f_{\mathrm{J}}/q_{\mathrm{J}})/(f_{\mathrm{C}}/q_{\mathrm{C}})$. By Amdahl's law (Amdahl, 1967), eliminating measured acquisition wait while holding the baseline non-wait time per sample fixed gives a speedup bound of $S_{\mathrm{A}} = 1/(1 - f_{\mathrm{C}})$. This bound relates the baseline's measured wait share to the training-loop speedup attainable by removing that wait alone. Appendix A.1 lists the distinct endpoints and timing boundaries.

Workloads and hardware. All experiments run on one Intel Core Ultra 9 285K workstation with a 16-GiB RTX 5080. Resident performance uses frozen 128×128 Bernoulli maps at obstacle

density 0.2 and a 256-step horizon. Transition scaling varies agents $A \in \{64, 128, 256, 512\}$ and environments $N \in \{16, 64, 256, 1024\}$. Expert training uses 65×65 Maze layouts: the matched-budget MAGAT+ study has four distinct maps, four LaCAM workers, and 2,048 rows per update. Semantic checks also cover random, Maze, and Warehouse maps; Appendix D varies Maze layouts.

Simulation and MAGAT controls. POGEMA with one or eight CPU workers (POGEMA-1/8) supplies CPU transition references; our JAX/XLA backend supplies a semantically matched GPU reference. The complete MAGAT-decision comparison keeps state, inputs, and the same eager policy on GPU in both paths. It fixes checkpoint, precision, microbatching, and argmax action selection while varying builders, transitions, and framework composition. After compilation and loading, the timer runs from input construction to the completed successor state; cache-warm decisions follow a matched full step. A separate deployment comparison uses CPU POGEMA (Appendix C).

Training controls and budget. The suite has three CPU input baselines, each adding a stronger form of host preparation. *Host-expanded synchronous* constructs PyTorch Geometric inputs on CPU and transfers them on demand. *Cached synchronous* reuses map and goal information but still constructs and transfers each batch on demand. *Cached prefetch* adds background builders and pinned-memory H2D lookahead. *JointLoom streaming* instead buffers compact configurations and reconstructs inputs on GPU.

A three-seed online control pairs cached prefetch (H8/Q16) with JointLoom for 300 seconds at eight or sixteen experts. It uses batches of 4,096 or 8,192, respectively, and source-ring depth $D = 512$ in both paths. H8/Q16 denotes eight CPU builders and 16 prefetch slots, separate from D. The eight- and sixteen-expert workloads repeat four Maze layouts twice and four times, respectively. Initialization, source, and expert protocol match within pairs; batch size scales with expert count (Appendix F.4).

To test a fixed CPU resource budget, the three-seed control with eight experts and a 4,096-row batch assigns each paired pipeline the same eight- or sixteen-core affinity. Cached prefetch uses H8/Q16, selected by a predeclared 45-second CPU-pool calibration. The model, expert count, batch, ring depth, and 180-second timer match within each affinity comparison (Appendix F.4).

The MAPF-GPT control pairs H1/Q2 cached prefetch with JointLoom for 180 seconds at four experts, a 1,024-row batch, and ring depth four. Seed 42 is repeated under eight- and sixteen-core affinity (Appendix F.4).

The matched-budget MAGAT+ study holds experts, model, optimizer, batch, and checkpoint rule fixed over a 3,035.922-second maximum. JointLoom uses depth 128 and reaches its 100,000-update cap; host-expanded CPU has one-stage backpressure. Separate fixed-record controls replay identical records.

Fixed-work timers cover optimizer updates but exclude validation and writes; the matched-budget timer includes periodic validation and checkpoints after initialization. Appendix E specifies boundaries.

Learning-curve construction. Figure 5 plots three-seed medians and min–max bands against wall time for JointLoom and two CPU paths, using four frozen validation trajectories. Validation accuracy is the fraction of agent actions matching LaCAM labels on those trajectories. Cached CPU prefetch contributes a matched-setting three-seed cohort; cached synchronous appears in fixed-record controls. Loss is the mean over the preceding 200 updates at each recorded checkpoint and is shown alongside validation accuracy. Appendix F provides individual curves and update alignment.

Verification. Semantic and learning-update checks precede timing. Transition cells use five repetitions; the resident MAGAT decision and MAGAT training comparisons use three paired runs or seeds. MAPF-GPT online training repeats one seed under two CPU-affinity budgets. Appendices A–G provide the full records.

5 EVALUATION

The results trace input delivery from semantically correct reconstruction to complete decisions and training. We verify semantics and simulation throughput, time complete decisions, isolate reconstruction and buffering, and measure exposed wait against cached CPU prefetch. A matched-budget study then shows how faster delivery affects training progress.

5.1 REFERENCE FIDELITY AND SIMULATION SCALING

The POGEMA-referenced suite passes 11,344 backend–case comparisons, checking positions, occupancy, collisions, arrivals, and episode completion. Shared-input checks also pass: reconstruction 4/4, one-step learning 6/6, lifecycle conformance 2/2, and all 12 registered fault cases. These gates establish the semantics used in the timing studies (Appendix A).

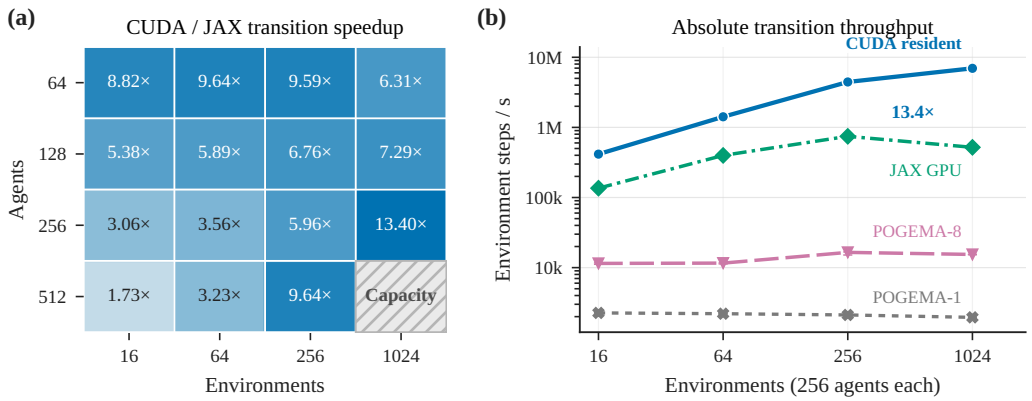

Figure 3: **GPU-resident simulation across agents and environments.** (a) CUDA/JAX transition-throughput ratios. (b) Joint environment steps/s at $A = 256$ (five-run min–max). The $A = 512$ launch uses 512 threads; its $N = 1024$ endpoint requires more memory for the current cache and graph storage than the evaluated GPU provides (Appendix C).

Figure 3 counts complete joint environment steps as agents A and environments N vary. At $A = 256, N = 1024$, CUDA reaches 6.960 million steps/s versus 0.520 million for JAX, a $13.40\times$ ratio. The $A = 512, N = 1024$ cell exceeds the evaluated GPU's memory capacity (Appendix G). Appendix C reports all rates. Transition throughput establishes simulation capacity; complete decisions also include the input builder and policy.

5.2 COMPLETE POLICY DECISIONS ACROSS INPUT FAMILIES

We time Equation 3 against a GPU-resident JAX/XLA–PyTorch path with the same MAGAT policy, after input, logit, action, and successor-state checks (Appendix C.2).

At $A = 256, N = 256$, complete-decision latency falls from 596 to 157 ms, a paired $3.79\times$ improvement (3.75–$3.88\times$). Profiles locate savings in input-building, transition, and framework stages; model execution takes the largest share of the resident path (Appendix C.2).

MAPF-GPT tests ordered tokens and action history. Across three first decisions including goal-cache construction, it gains 1.41–$1.55\times$ over its GPU-resident JAX/PyTorch control; warm decisions spend a larger share in shared Transformer work (Appendix C.4).

5.3 COMPACT RECONSTRUCTION AND BUFFERED EXPERT SUPPLY

Policy execution starts from resident state; imitation learning receives configurations from CPU LaCAM experts. Figure 4 separates compact GPU reconstruction, which changes view construction and transfer, from buffering, which reduces waiting for a complete next stage.

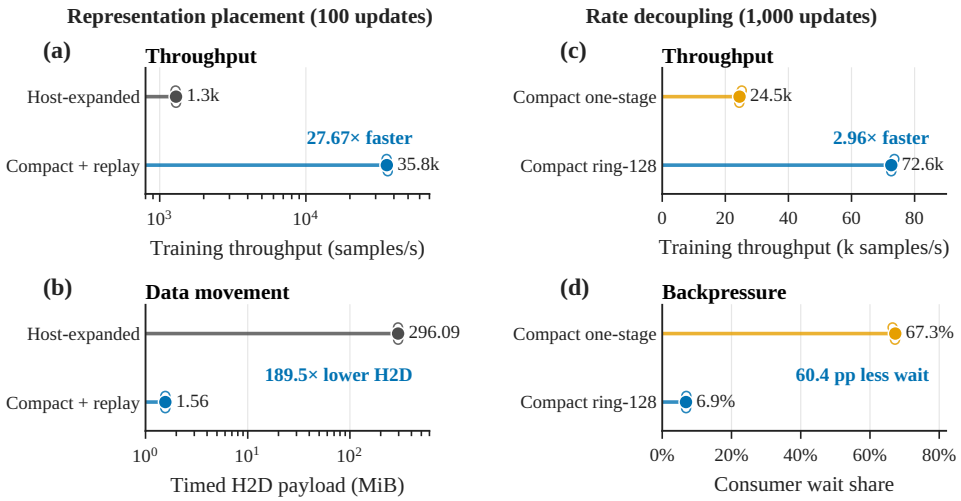

Figure 4: **Separate controls for input reconstruction and buffering.** Left: host-expanded inputs versus compact GPU reconstruction over 100 updates, reporting throughput and H2D bytes over the timed workload. Right: depth-one versus depth-128 buffering with reconstruction fixed over 1,000 updates, reporting throughput and consumer wait. Each point uses three paired runs. Appendix D gives sensitivity and transfer-mode controls.

Compact reconstruction. In the 100-update MAGAT control, compact records and GPU replay replace host-expanded inputs. Throughput rises from 1,293 to 35,779 samples/s (27.7×), while host-to-device (H2D) payload over the timed workload falls from 296.1 to 1.56 MiB (189.5×).

Buffering at fixed representation. With compact reconstruction fixed, increasing ring depth from one to 128 over 1,000 updates raises throughput from 24,500 to 72,607 samples/s (2.96×) and cuts consumer wait from 67.3% to 6.9%. Ordered write-ahead preserves each environment's contribution; transfer-trigger, worker/depth, and within-Maze layout controls characterize the gain (Appendix D).

Compact stages also reduce retained storage: expanded views use about 198× as much space in the measured shapes (Appendix D.6).

Fixed-record MAGAT and MAPF-GPT controls appear in Appendix F. The online controls next test whether these mechanisms keep the learner supplied during expert generation, against cached CPU prefetch.

5.4 Online training: input wait and throughput

Paired online controls compare JointLoom with cached CPU prefetch for MAGAT at eight and sixteen experts and MAPF-GPT at four. Table 1 aligns baseline wait share, the reduction in wait per sample, training-loop speedup, and the Amdahl bound S_A.

Table 1: **Exposed input wait and Amdahl's-law speedup bounds.** C/J denotes cached CPU prefetch/JointLoom; rates are thousands of agent samples/s, and wait reduction uses acquisition time per sample. MAGAT uses three paired 300-s seeds at depth 512; MAPF-GPT uses seed 42 for 180 s at eight-core affinity and depth four. Derived statistics are computed within pairs before aggregation; Appendix F.4 gives the ranges.

Policy; experts/batch	Rate C/J (k/s)	Wait C/J (%)	Wait reduction	Speedup (×)	S_A (×)
MAGAT; 8/4,096	41.05/81.43	50.4/2.5	97.5%	1.98	2.02
MAGAT; 16/8,192	37.41/77.79	52.8/2.4	97.9%	2.08	2.12
MAPF-GPT; 4/1,024	3.657/3.815	5.09/0.71	86.7%	1.043	1.054

At eight and sixteen MAGAT experts, cached-prefetch wait occupies 50.4% and 52.8% of the loop; JointLoom lowers it to 2.5% and 2.4%. Wait per sample falls by 97.5% and 97.9%. Training-loop speedups of $1.98\times$ and $2.08\times$ are close to Amdahl bounds of $2.02\times$ and $2.12\times$ (Appendix F.4).

For MAPF-GPT, the 5.09% baseline wait share falls to 0.71%; wait per sample drops by 86.7%. The $1.043\times$ training-loop speedup approaches its $1.054\times$ Amdahl bound, with the same pattern under sixteen-core affinity. Both policy families remove most exposed wait; their initial wait shares yield different Amdahl bounds on speedup from eliminating input wait.

A three-seed fixed-affinity MAGAT control reaches $2.44\times$ and $2.05\times$ at eight and sixteen cores. At eight cores, lower non-wait cost adds to the input-wait savings (Appendix F.4). The matched-budget study then tests how sustained delivery changes optimization and validation progress.

5.5 Training progress under a common maximum budget

The matched-budget MAGAT+ study uses four LaCAM experts and a 2,048-row batch. JointLoom, host-expanded synchronous CPU, and a matched-setting cached-prefetch CPU cohort share expert protocol, initialization, model, optimizer, and checkpoint settings over a 3,036-s maximum (Appendix E).

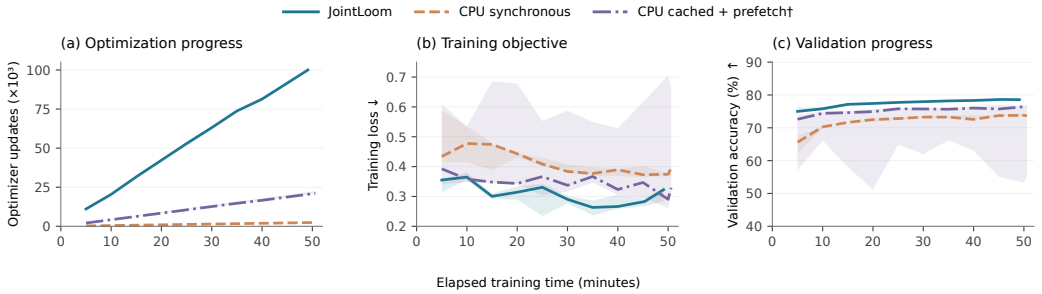

Figure 5: **Training progress against two CPU baselines.** Three-seed medians and min–max bands for updates, preceding-200-update mean loss, and accuracy on the fixed validation set. †Cached CPU prefetch contributes a matched-setting cohort with per-update loss histories. Individual curves and cohort details appear in Appendix F.

Optimization and validation progress. JointLoom reaches its 100,000-update cap in 2,941–2,983 s. At the time limit, host-expanded CPU completes 2,430–2,450 updates and cached prefetch 20,893–22,597: 40.8–$41.2\times$ and 4.43–$4.79\times$ as many updates at the same batch size.

Timers include periodic validation and checkpoint writes. Best recorded validation accuracy reaches 78.55–78.90% for JointLoom versus 73.88–74.24% for host-expanded CPU. Figure 5 shows loss and fixed-validation trajectories alongside update counts. Under this common budget, faster input delivery supports more optimizer updates and higher validation accuracy.

6 Conclusion

JointLoom makes the complete joint configuration and required policy context the interface between MAPF state producers and learned-policy consumers. Graph and token builders use that interface in both resident decisions and buffered expert training, with cache, publication, and storage-lifetime rules preserving input semantics. Across both policy families, online controls reduce exposed acquisition wait; MAGAT comparisons show training-loop gains close to the speedup available from removing that wait. Under a matched budget, compact streaming also completes more optimizer updates and reaches higher validation accuracy than synchronous CPU input construction. These results make input delivery an explicit, reusable target for accelerating learned MAPF workflows.

Reproducibility Statement

Appendices A–G specify the correctness checks, record layouts, stage consumption, paired timing protocols, checkpoint selection, recorded learning curves, and capacity measurements. Table 4 identifies the inputs and reusable components for the evaluated workflows. Appendix I maps their results to frozen evidence and software versions.

AI Use Statement

Generative AI assisted with manuscript review, restructuring, and presentation. It also provided feedback on the qualitative cost analysis, experimental design, and interpretation of results. The semantic, builder, learning-update, and lifecycle checks reported in this paper assess the implementation. The authors are responsible for the final text, claims, and code.

References

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 483–485, 1967. doi: 10.1145/1465482.1465560.

Anton Andreychuk, Konstantin Yakovlev, Aleksandr Panov, and Alexey Skrynnik. MAPF-GPT: Imitation learning for multi-agent pathfinding at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 23126–23134, 2025. doi: 10.1609/aaai.v39i22.34477.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Vincent Abramowitz, Paul Coyette, Daniel Duckworth, Laurence Illing Midgley, Thomas Kalloniatis, Julien Perolat, et al. Jumanji: A diverse suite of scalable reinforcement learning environments in jax. In *International Conference on Learning Representations*, 2024.

Michał Bortkiewicz, Władysław Pałucki, Vivek Myers, Tadeusz Dziarmaga, Tomasz Arczewski, Łukasz Kuciński, and Benjamin Eysenbach. Accelerating goal-conditioned reinforcement learning algorithms and research. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4gaySj8kvX.

Albin Cassirer, Gabriel Barth-Maron, Eugene Brevdo, Sabela Ramos, Toby Boyd, Thibault Sottiaux, and Manuel Kroiss. Reverb: A framework for experience replay. *arXiv preprint arXiv:2102.04736*, 2021. URL https://arxiv.org/abs/2102.04736.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: Scalable and efficient deep-rl with accelerated central inference. In *International Conference on Learning Representations*, 2020.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax: A differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

Chengyang He, Tanishq Duhan, Gadiel Sznaier Camps, Fangyuan Wang, Yuhong Cao, Jiankai Sun, Ge Sun, Mac Schwager, and Guillaume Sartoretti. PRIMAL3: Pathfinding via reinforcement and imitation multi-agent learning—leveraging LaCAM3. *arXiv preprint arXiv:2608.04905*, 2026.

Rishabh Jain, Keisuke Okumura, Michael Amir, Pietro Liò, and Amanda Prorok. Pairwise is not enough: Hypergraph neural networks for multi-agent pathfinding. In *International Conference on Learning Representations*, 2026a. URL https://arxiv.org/abs/2602.06733.

Rishabh Jain, Keisuke Okumura, Michael Amir, and Amanda Prorok. Graph attention-guided search for dense multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29504–29512, 2026b. doi: 10.1609/aaai.v40i35.40192.

Shalin Jain, Jiazhen Liu, Siva Kailas, and Harish Ravichandar. JaxRobotarium: Training and deploying multi-robot policies in 10 minutes. In *Proceedings of the Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 1975–1996, 2025. URL https://proceedings.mlr.press/v305/jain25a.html.

He Jiang, Yutong Wang, Rishi Veerapaneni, Tanishq Duhan, Guillaume Sartoretti, and Jiaoyang Li. Deploying ten thousand robots: Scalable imitation learning for lifelong multi-agent path finding. In *2025 IEEE International Conference on Robotics and Automation*, pp. 1–7, 2025.

Saman Kazemkhani, Aarav Pandya, Daphne Cornelisse, Brennan Shacklett, and Eugene Vinitsky. GPUDrive: Data-driven, multi-agent driving simulation at 1 million FPS. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=ERv8ptegFi.

Tian Lan, Sunil Srinivasa, Huan Wang, and Stephan Zheng. WarpDrive: Fast end-to-end deep multi-agent reinforcement learning on a gpu. *Journal of Machine Learning Research*, 23(316):1–6, 2022.

Qingbiao Li, Weizhe Lin, Zhe Liu, and Amanda Prorok. Message-aware graph attention networks for large-scale multi-robot path planning. *IEEE Robotics and Automation Letters*, 6(3):5533–5540, 2021. doi: 10.1109/LRA.2021.3077863.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac Gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Zhiyu Mei, Wei Fu, Jiaxuan Gao, Guangju Wang, Huanchen Zhang, and Yi Wu. SRL: Scaling distributed reinforcement learning to over ten thousand cores. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=lajn1iROCu.

Keisuke Okumura. LaCAM: Search-based algorithm for quick multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 11655–11662, 2023. doi: 10.1609/aaai.v37i10.26377.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7652–7662, 2020.

Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, T. K. Satish Kumar, Sven Koenig, and Howie Choset. PRIMAL: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters*, 4(3):2378–2385, 2019. doi: 10.1109/LRA.2019.2903261.

Brennan Shacklett, Erik Wijmans, Aleksei Petrenko, Manolis Savva, Dhruv Batra, Vladlen Koltun, and Kayvon Fatahalian. Large batch simulation for deep reinforcement learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=cnhSKyPDb0.

Alexey Skrynnik, Anton Andreychuk, Anatolii Borzilov, Alexander Chernyavskiy, Konstantin S. Yakovlev, and Aleksandr Panov. POGEMA: A benchmark platform for cooperative multi-agent pathfinding. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=DG4k8PJ8qv.

Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Barták. Multi-agent pathfinding: Definitions, variants, and benchmarks. *Artificial Intelligence*, 287:103382, 2020. doi: 10.1016/j.artint.2020.103382.

Yimin Tang, Xiao Xiong, Jingyi Xi, Jiaoyang Li, Erdem Bıyık, and Sven Koenig. RAILGUN: A unified convolutional policy for multi-agent path finding across different environments and tasks. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 8645–8651, 2025. doi: 10.1109/IROS60139.2025.11247527.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

Rishi Veerapaneni, Qian Wang, Kevin Ren, Arthur Jakobsson, Jiaoyang Li, and Maxim Likhachev. Improving learnt local MAPF policies with heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pp. 597–606, 2024. doi: 10.1609/icaps.v34i1.31522.

Yuanzhe Wang, Tian Zhi, Zihang Wei, Hongguang Wang, Jiaming Guo, Yang Zhao, Zisheng Liu, Shiyu Quan, Xing Hu, Zidong Du, and Yunji Chen. Discrete diffusion for complex and congested multi-agent path finding with sparse social attention. *arXiv preprint arXiv:2605.13296*, 2026.

Yutong Wang, Bairan Xiang, Shinan Huang, and Guillaume Sartoretti. SCRIMP: Scalable communication for reinforcement- and imitation-learning-based multi-agent pathfinding. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 9301–9308, 2023.

Kun Wu, Mert Hidayetoğlu, Xiang Song, Sitao Huang, Da Zheng, Israt Nisa, and Wen-mei Hwu. Hector: An efficient programming and compilation framework for implementing relational graph neural networks in gpu architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 460–476, 2024. doi: 10.1145/3620666.3651322.

Jingtian Yan, Zhifei Li, William Kang, Kevin Zheng, Yulun Zhang, Zhe Chen, Yue Zhang, Daniel Harabor, Stephen F. Smith, and Jiaoyang Li. Advancing MAPF toward the real world: A scalable multi-agent realistic testbed (SMART). *IEEE Robotics and Automation Letters*, 11(6):7428–7435, 2026. doi: 10.1109/LRA.2026.3688062.

Chuanlong Zang, Anna Mannucci, Isabelle Barz, Philipp Schillinger, Florian Lier, and Wolfgang Hönig. GRACE: A unified 2d multi-robot path planning simulator & benchmark for grid, roadmap, and continuous environments. *arXiv preprint arXiv:2603.10858*, 2026.

A SEMANTIC FIDELITY AND CORRECTNESS CHECKS

The performance comparisons rest on four semantic contracts: transition semantics, reconstructed inputs, one-step learning, and stage publication. This section records the paired paths and frozen checks for each contract.

A.1 COMPARISON CONTRACTS

Table 2: **What each comparison measures.** Simulation and complete-decision integration evaluate the resident workflow. Fixed-work, online-supply, and matched-budget controls evaluate expert streaming. Each E4 integration protocol retains its paired workload and timer.

Question	Compared paths	Shared factors	Measurement
Simulation	CUDA; our JAX GPU backend; POGEMA-1/8	Initial states and joint actions	Transition rate and exact state fields
MAGAT integration	CUDA resident; our GPU-resident JAX/XLA–PyTorch control	Checkpoint, inputs, precision, microbatch, argmax	View through completed transition (E4b)
MAPF-GPT integration	CUDA resident; GPU-resident JAX control (E4c); hybrid CPU POGEMA control (E4g)	Checkpoint, tokens, history, microbatch, argmax within each pair	First and warm complete decisions
CPU deployment	POGEMA-8 + MAGAT; resident MAGAT	Original paired E4 workload	Complete deployment under the E4 timing scope
Streaming mechanisms	Host-expanded; compact replay; shallow/deep ring	Model, optimizer, sample budget and expert workload within each control	Fixed-work training throughput and wait
Online input supply	Cached CPU prefetch; compact streaming	Model, batch, experts, and seed within each pair; CPU affinity in fixed-resource controls	Consumer input wait and training-loop throughput
Learning utility	Compact streaming; synchronous and cached-prefetch CPU paths	Expert protocol, model/optimizer settings, batch, maximum time, selection	Completed updates, training loss, and validation accuracy

The contracts in Table 2 keep each performance result within its own timing scope. The checks below establish the computations and data lifetimes used in those scopes; Section 5.1 summarizes their outcomes.

A.2 STANDARD-MAPF TRANSITION SUITE

POGEMA is the reference implementation. Identical initial states and joint action sequences are replayed through the CUDA and JAX backends. The suite contains 38 frozen requests and 5,672 unique cases. Comparing each backend with POGEMA gives 11,344 backend–case rows in total.

Directed cases cover unconstrained motion, wall blocking, vertex conflicts, edge swaps, movement cycles, initially arrived agents, early all-arrived termination, and horizon truncation. Other requests cover random, Maze, and Warehouse map families; 64, 128, 256, and 512 agents; and the density and environment pools used in the scaling study. These are semantic checks; the performance protocols specify their own map workloads below. Each compared step checks positions, goals, occupancy, collision outcomes, arrival flags, termination, truncation, and episode length. Both backends match all registered fields (Table 3).

Table 3: **Reference agreement in the frozen transition suite.** CUDA and JAX are each compared with POGEMA on the same 5,672 cases. All registered state fields match; the totals count backend–case comparisons.

Candidate	Frozen cases	Comparison rows	Audited state fields
CUDA	5,672	5,672	Position, goal, occupancy, collision outcome, arrival, termination, truncation, episode length
JAX	5,672	5,672	Position, goal, occupancy, collision outcome, arrival, termination, truncation, episode length
Total	5,672	11,344	Zero mismatches in the frozen suite

Table 4: **Inputs, reusable components, and evidence for the evaluated workflows.** Policy execution denotes the complete simulator–policy loop; expert-supervised learning measures delivery efficiency and finite-budget training progress.

Research task	Experiment supplies	Implemented components	Outputs and evidence
Batched policy execution	Maps, initial configurations, checkpoint, and a supported input specification	Resident state and transition; GPU input builder; policy-loop integration	Actions and successor states; transition and MAGAT timing (Sections 5.1–5.2)
Expert-supervised learning	Complete configurations and action labels from LaCAM; model and optimizer settings	Compact transport; ordered buffering; GPU state replay and input construction	Updates and selected checkpoints; fixed-work and matched-budget studies (Sections 5.3–5.5)

A.3 Builder, learning, and lifecycle checks

The suite also verifies policy inputs and training updates. Four builder trials compare MAGAT and MAPF-GPT inputs at the first and final stages of a frozen 32-stage stream. Six one-step learning trials cover both backends and three seeds, comparing inputs, logits, loss, gradients, and post-optimizer parameters. Structural and discrete fields must match exactly; floating-point fields must satisfy the registered absolute and relative tolerances. All four builder trials and all six learning trials pass.

Two lifecycle trials then exercise reservation, publication, consumption, and release. Twelve fault cases cover partial inputs, incomplete stages, ordering and validation errors, pre-publication exceptions, consumer exceptions, and process teardown. All lifecycle trials and registered faults pass their specified conformance, rejection, or propagation checks. Failure handling uses the fail-stop behavior described in Appendix B.7.

The evidence archive retains the frozen inputs and tolerances for these transition, builder-stream, and one-step learning gates.

B Runtime Layout and Complete-Block Lifecycle

The checks above depend on which state is authoritative and when a producer may release it. We describe the evaluated workflows and backend requirements before specifying layout, compact records, and the publication lifecycle.

B.1 Workflow inputs, outputs, and reuse

Table 4 describes the research tasks exercised by the reported experiments. It separates the inputs supplied by an experiment from the execution and data-delivery components implemented in JointLoom. The resident loop and expert stream use independent state but share the selected backend's input builder and goal-cache rules. Training utility is assessed through optimization and action accuracy on frozen validation trajectories.

A policy using an implemented input specification can reuse its builder. Changes to observations, communication edges, token encoding, or required history call for a matching builder and reconstruction checks. The taxonomy below distinguishes the two evaluated interfaces from possible extensions; the following layout and capacity analysis gives the dimensions supported by the present implementation.

B.2 Learned-MAPF interface taxonomy

Each policy backend determines its own model view, required context, and refresh rules. Table 5 distinguishes the evaluated MAGAT and MAPF-GPT interfaces from extensions that require new builders or additional state.

Table 5: **State, model views, and refresh rules across learned-MAPF interfaces.** Context denotes information needed beyond standard-MAPF state s_t. The final column identifies evaluated uses and the implementation needed to extend them. Table 4 maps the supported input specifications to execution and training tasks. Derived caches and model views remain reconstructible.

Family	Consumer view	Persistent context beyond s_t	Refresh / invalidation	Status here
MAGAT / LaGAT / HMAGAT	Local features with pairwise graph; directed hypergraph for HMAGAT	None in the reactive policy; guided search owns separate planner state	Rebuild graph or hypergraph as the configuration changes	MAGAT execution and MAGAT+ training evaluated; HMAGAT requires a dedicated hypergraph builder
MAPF-GPT	Current local observation, ordered neighbor tokens, action-history tokens	Bounded action history h_t; goal-distance cache is derived	Roll history and local tokens each decision; refresh cache on goal change	Input/update checks; GPU and CPU decision pairs; fixed-record and online training
PRIMAL / SCRIMP / PRIMAL3	Local grid; global/local communication; topology features and dual graphs	Recurrent/communication state; PRIMAL3 also uses persistent execution priorities	Update per decision; reset recurrent state per episode; rebuild topology-dependent views	Design extension: recurrent state and PRIMAL3 graph/refinement builders
CS-PIBT / LaCAM guidance	Per-agent action distributions or ordered preferences consumed by search	Search frontier, priorities, and configuration history belong to the planner	Query preferences at each decision or generated search configuration	LaCAM supplies expert data; downstream guidance archived; CS-PIBT integration is a design extension
RAILGUN	Centralized map-wide convolutional tensor	Model-specific global workspace	Rebuild when the global configuration changes	Requires a new centralized-view builder
DiffLNS	Joint spatiotemporal categorical-action drafts with sparse social attention	Diffusion sample and downstream LNS2 repair state	Update through denoising and neighborhood repair	Requires a trajectory sampler and search integration
Lifelong methods (e.g., SILLM)	Local heuristic features plus learned communication	Recurrent context and the current goal assignment	Goal reassignment invalidates goal-derived state in our cache design	Extended task semantics with lifelong goal reassignment

Across these interfaces, semantic requirements and storage lifetimes are distinct. Table 6 separates authoritative state, derived caches, compact transport, and model-view storage.

Table 6: **Runtime objects, owners, and lifetimes.** Caches are derived data even when they remain on GPU. Model-view contents and the allocation backing them can have different lifetimes.

Class	Owner and lifetime	Recurring boundary
State and required context	Authoritative state and policy history; run, episode, or context lifetime	Initialization or compact training records
Derived goal cache	Reconstructible distances or energy maps; valid until dependency change or refresh	GPU-local reuse and explicit invalidation
Compact transport	LaCAM blocks plus ring metadata; publish once, consume over one or more rounds, then release safely	Compact H2D payload and host control
Per-call model view	GPU observations, graphs, tokens, and masks; valid through final consumer use, including backward	Constructed at the consumer; backing reused afterward

B.3 Runtime indexing and allocation

For N environments with A agents each, the runtime indexes each agent row by

$$r(e, i) = eA + i, \qquad 0 \le e < N, \quad 0 \le i < A. \tag{4}$$

Changing A or N changes tensor extents and strides. Static topology, dynamic state, goal-derived maps, and backend workspace have separate allocations and lifetimes. The evaluated binary embeds maps in a 128×128 build-time envelope and fixes the backend fields of view (FOVs).

A larger map increases per-agent BFS workspace. The current block-local frontier and dynamic-shared-memory guards have been evaluated within the 128×128 envelope. Larger maps can use a global-memory or multi-kernel BFS after configuration-specific validation. Changing the FOV changes model-input semantics and calls for a compatible checkpoint and parity checks.

Table 7: **Design options for heterogeneous teams.** These options preserve complete environment configurations and identify the allocation or indexing changes for each design.

Layout	Compact transport	GPU builder / allocation	Expected trade-off
Pad to $A_{\max}$	Fixed rectangular stage plus active mask	Reuse current strides and kernels	Simplest; work and persistent state scale with padding
Bucket by A	One homogeneous ring and runtime per bucket	Reuse current kernels per bucket	Less padding; more launches, queues, and scheduling state
True ragged offsets	Concatenated rows plus environment offsets	Offset-aware transition, cache, graph, and token kernels	Least padding; broadest kernel and parity revalidation

B.4 Layout options for heterogeneous teams

The rectangular indexing above uses one team size A per runtime. Table 7 compares padding, bucketing, and ragged offsets for heterogeneous teams. Each keeps publication at the level of a complete environment configuration but changes allocation or scheduling costs.

B.5 Goal-conditioned derived state

Within the evaluated rectangular layout, goal caches retain backend-specific encodings. For free-cell distance $D_{e,i}(c) = d_{G_e}(c, g_{e,i})$, the MAGAT energy encoding is

$$E_{e,i}(c) = \begin{cases} 255 - D_{e,i}(c), & D_{e,i}(c) \le 254, \\ 0, & \text{otherwise}, \end{cases} \tag{5}$$

The MAGAT cache is a contiguous `uint8` tensor with shape $[N, A, H, W]$. A dirty agent–environment pair launches a goal-rooted, four-neighbor BFS block over static free cells. The goal has value 255; a representable cell at distance d has value $255 - d$. Unreachable cells and distances above 254 both map to zero, as required by the MAGAT input protocol.

MAPF-GPT stores exact goal distances in a $[N, A, H, W]$ `uint16` tensor, with 65,535 marking unreachable cells. Its token builder encodes local differences $D(c) - D(x_{e,i})$, clips finite differences to $[-20, 20]$, and reserves a separate unreachable token. Neighboring costs also determine the greedy-direction mask. The backends share source positions, goals, and cache lifetimes, while retaining their own dtypes and numerical conventions.

Invalidation is tracked per agent. External state loading marks every MAGAT goal dirty, ensuring that the first decision builds the required maps. Subsequent resident updates prefix-compact dirty flags and recompute changed goals. Standard-MAPF transitions keep goals fixed, so these flags remain clean after motion alone. Streaming records can request a refresh explicitly. The MAPF-GPT unpacker compares each incoming goal with a two-coordinate cache and combines the result with the reset flag; its BFS kernel exits immediately for clean agents.

Motion still changes occupancy, local views, graphs, and tokens. A topology replacement requires environment or builder reinitialization; the reported experiments retain no derived state across different topologies. Table 8 summarizes the two representations and their invalidation rules.

The comparisons hold each backend's numerical input protocol fixed. A different storage format would need reconstruction-equivalence and performance checks. Changing reconstructed values or token conversion would also require policy validation, because it changes the model input itself.

B.6 Compact training records

The stream carries the state needed to reconstruct inputs and refresh their caches. A MAGAT row contains eight 16-bit fields. Stage sequence and ring metadata identify the ordered configuration; independent environments retain their own episode clocks. Equality of current and goal coordinates

Table 8: **Persistent goal caches for the two backends.** Both caches are refreshed per affected agent. Their storage types and numerical encodings follow their respective model-input protocols.

Backend	Persistent storage	Numerical representation	Invalidation
MAGAT	`uint8` $[N, A, H, W]$	$255 - D$ for $D \leq 254$; zero otherwise	Explicit reset/refresh or changed goal
MAPF-GPT	`uint16` $[N, A, H, W]$	D; 65,535 for unreachable, followed by relative-cost tokenization	Explicit reset or detected goal change

Table 9: **Compact transport fields.** MAGAT uses the eight-field core; MAPF-GPT appends five previous commands. Observations, graphs, and tokens are reconstructed at the consumer.

Columns	Field	Storage	Role
0	Environment id	16-bit integer	Selects the environment owner
1	Agent id	16-bit integer	Contiguous index in $[0, A)$
2–3	Current x, y	16-bit integer	Authoritative position
4–5	Goal x, y	16-bit integer	Fixed standard-MAPF target
6	Expert action	16-bit integer	Supervision class in $[0, 4]$
7	Refresh flag	16-bit integer	Reset/derived-state refresh marker
8–12	Action history	16-bit integer	MAPF-GPT only: five prior commands

gives arrival status. MAPF-GPT adds five previous action commands. Table 9 lists the fields; observations, graphs, and tokens are constructed at the GPU consumer rather than placed in transport records.

MAGAT computes the arrival mask as

$$m_{e,i} = [x_{e,i} = x^g_{e,i}] \wedge [y_{e,i} = y^g_{e,i}]. \tag{6}$$

MAPF-GPT produces the corresponding active mask in its CUDA token kernel. A training protocol can use this mask to exclude arrived-agent labels from loss while retaining those agents in the forward pass and joint context. The primary matched-budget MAGAT+ run includes arrived-agent labels.

B.7 Stage layout, publication, and multi-round consumption

These records must reach the learner as complete configurations. Let D be ring depth in complete stages. Each stage contains NA rows. For publication sequence q, environment e, and agent i, the offset is

$$o(q, e, i) = (q \bmod D)NA + eA + i. \tag{7}$$

All A agents of a configuration are contiguous. One stage contains a configuration from every environment at the same publication sequence q; successive configurations from an environment occupy successive stages until the ring wraps. Environments may finish and reset at different times. Their within-episode clocks can therefore differ even when they contribute to the same publication stage.

Publication and consumption. The NA-row stage is the admission unit, the A-row configuration is the reconstruction unit, and the B-row batch is the optimizer unit. An admitted stage can supply several optimizer rounds. Conversely, a large batch can combine whole configurations from successive admitted stages. The readiness predicate depends on publication, independently of the optimizer batch size.

For $N = 8, A = 256$, each stage holds 2,048 rows. At $B = 1024$, its eight complete configurations supply two rounds of four configurations. Each selected builder still receives all 256 agents. Consumption progress records the sub-blocks that remain to be used, separately from stage readiness.

Pending sub-blocks and outstanding device reads retain valid storage; model-view contents remain valid through their final forward/backward use.

The lifecycle is:

1. A producer forms exactly A rows for one configuration and expert joint action, checking one environment id and contiguous agent ids $0, \ldots, A-1$.
2. Under the reservation lock, it obtains that environment's write sequence. It waits when it is D stages ahead of the consumed sequence.
3. The producer copies the complete sub-block into pinned shared memory and then increments the stage-ready count.
4. Stage q becomes admissible when its sequence tag equals q and its ready count reaches N.
5. The consumer forms batches from whole configurations in admitted stages, restores their state, and constructs the policy inputs. A stage can supply several rounds, and a batch can span several stages.
6. Consumption progress tracks the configurations still needed from each stage. Their data and any in-flight reads remain valid; model-view backing is separately retained through its final consumer, including backward.
7. After all scheduled sub-blocks have been handed off and reuse is safe, release clears the ready count, sequence tag, and row-ready flags, then advances the consumed sequence.

These rules specify the ownership conditions for reuse. They leave the choice of event, counter, or copy-stream implementation open. Ring depth D counts publication stages regardless of how many optimizer rounds consume them. A ready stage can still contain pending data, so readiness and release are separate conditions.

Detected failures. A producer failure before publication leaves the stage incomplete and invisible to the consumer. On a detected producer exit or deadline violation, the supervisor aborts the run, terminates the remaining workers, and discards the run-local ring. Recovery starts a fresh run rather than reassigning the incomplete slot in place. Table 10 gives the visibility and response at each failure point.

Table 10: **Visibility and response at each publication failure point.** An invisible stage fails the readiness predicate and is withheld from DMA admission. Normal release also waits for safe reuse after consumption.

Failure point	Consumer-visible state	Implemented response
Before or during row copy	Stage remains incomplete	Detect exit/deadline, abort run, discard ring
After copy, before publish count	Written rows remain invisible	Detect exit/deadline, abort run, discard ring
After publish count	Complete environment block may contribute to stage	Consume only if all N blocks publish; otherwise abort
Normal completed stage	Full NA rows become visible once	Consume all assigned sub-blocks across the required rounds; clear metadata when reuse is safe

This fail-stop protocol assigns one writer to each logical environment. A deterministic regression hard-exits a child after reservation but before publication, checks that DMA admission stops, discards the ring, and verifies normal publication in a fresh run. The regression checks fail-stop visibility and restart after owner exit.

C Simulation and Policy-Integration Measurements

Simulation throughput captures one part of a learned-policy decision. This section first measures transitions, then includes input construction and policy execution in paired MAGAT and MAPF-GPT decisions. Diagnostic profiles locate the stages that account for the complete-decision gains.

C.1 TRANSITION THROUGHPUT, TIMING, AND SCALING

All transition rates count joint environment steps per second. One step advances every agent in one environment, without multiplying the rate by A. Timers cover repeated transitions over frozen trajectories. Input construction, warm-up, correctness replay, monitoring, memory sampling, and setup synchronization take place outside that region.

The matrix combines versioned transition archives. Retained A2 rows use five repetitions after five warm-up steps and a measurement interval of at least ten seconds. Newly filled E2 cells use their frozen lightweight protocol; the minimal $A = 256, N = 1024$ runner uses 120 transitions per repetition. Each exported cell records its runner and source. Within-cell backend ratios share a workload and timing boundary. Measurement horizons are recorded per cell, so the analysis centers on paired ratios and the reported scaling range.

All cells use frozen 128×128 Bernoulli maps with obstacle density 0.2 and episode horizon 256. The $A = 256$ row is the largest team size with accepted measurements at every displayed environment count for CUDA, JAX, POGEMA-8, and POGEMA-1. The joint matrix also retains $A = 512$, including its memory-limited endpoint. CUDA uses 256 threads per environment through $A = 256$ and 512 threads at $A = 512$.

Table 11: **CUDA/JAX transition-throughput ratios across the joint matrix.** The $A = 512$ row uses 512 threads per environment. The $A = 512, N = 1024$ request exceeds the current runtime's mandatory cache and MAGAT graph storage and cannot be admitted within GPU memory.

Agents \ Environments	16	64	256	1,024
64	8.82×	9.64×	9.59×	6.31×
128	5.38×	5.89×	6.76×	7.29×
256	3.06×	3.56×	5.96×	13.40×
512	1.73×	3.23×	9.64×	Exceeds memory

Table 11 gives CUDA/JAX ratios for every registered cell. Table 12 supplies absolute throughput at $A = 256$, including both CPU reference configurations. At $N = 1024$, the CUDA and JAX rates are 6,959.78k and 519.55k joint steps/s.

Table 12: **Absolute transition rates at 256 agents per environment.** Values are thousands of joint environment steps/s. POGEMA-8 uses the disclosed eight-worker diagnostic protocol for $N = 16$–256 and the minimal runner at $N = 1024$.

Method \ Environments	16	64	256	1,024
CUDA resident transition	415.50	1,416.54	4,444.11	6,959.78
JAX GPU transition	135.83	398.32	745.63	519.55
POGEMA-8	11.48	11.60	16.50	15.41
POGEMA-1	2.26	2.20	2.11	1.95

Each displayed median and min–max range uses five retained repetitions. POGEMA-8 values at $N = 16$–256 use the disclosed eight-worker diagnostic protocol; $N = 1024$ uses the minimal runner. These rates provide absolute CPU context. The source registry records the two POGEMA-8 runner variants. The plotted ranges show the minimum and maximum over repetitions.

C.2 MAGAT INTEGRATION AGAINST OUR GPU-RESIDENT CONTROL

Transition-only throughput does not include the policy workload. The MAGAT integration adds input construction, policy execution, and action selection around the transition. We implement both the CUDA path and the matched JAX/XLA–PyTorch control. In the control, JAX/XLA owns GPU state, constructs observations, and applies transitions; eager PyTorch constructs the graph and runs the frozen MAGAT checkpoint. Both paths preserve GPU residency, the frozen instances, and deterministic argmax actions.

The frameworks request copy-disabled DLPack handoffs. Exact-shape lowering and compilation, checkpoint loading, input upload, parity replay, and result readback are outside the timer; completion includes the post-transition state. E4b steady rows run a matched exact-shape full step before measurement. Its first-decision path uses the framework compilation cache. The steady rows supply the main MAGAT result. The token-backend E4c measurements in Appendix C.4 invoke retained compiled executables directly.

Table 13 records the allocator, compilation, layout, policy, and completion settings. The PyG policy implementation is fixed, while input builders, transitions, and framework composition vary. Compiler settings use the default framework heuristics. The comparison measures these implemented GPU-resident paths.

Table 13: **Settings for the GPU-resident integration controls.** The table separates enforced controls from their implementation scope. Included costs remain inside the complete-decision timer; setup exclusions are applied symmetrically within each pair.

Item	Enforced contract	Implementation detail
Process and allocator	Fresh process per row; fixed workload shape; JAX bulk preallocation disabled	One workstation with default framework allocators
Compilation	Exact shapes lowered/compiled before timing; E4c calls retained executables; steady E4b performs a matched full-step warm-up	E4b first decisions use the framework compilation cache
Layout and bridge	DLPack handoffs request `copy=False`; Torch inputs are contiguous where required; bulk state stays on GPU during timing	Default XLA layout and framework allocation
Policy	Same frozen checkpoint, eager PyTorch implementation, model microbatch, and deterministic argmax	Builder and framework paths select their own kernels
Bounded dynamic work	MAGAT energy construction uses 16-environment chunks; MAPF-GPT exact costs use at most 256 agent-goal maps per chunk	Chunk work and waits are included in first-decision timing
Dynamic shape	Eager MAGAT `nonzero` and its shape synchronization remain in the materialized path	Synchronization is included in measured path time
Completion	Both rows finish at the completed post-transition state; correctness readback follows timing	Diagnostic profiles and formal timers use separate runs

Before timing, gates compare inputs, logits, argmax actions, relevant history, and post-transition state. MAGAT model inputs and logits have zero maximum absolute error in this gate. Across MAGAT and supplementary MAPF-GPT, all 12 phase/workload cells and 36 paired timing repetitions match their applicable action-trajectory, history, and final-state hashes. Table 14 reports MAGAT E4b medians from three fresh processes per condition; the E4 CPU pair is reported separately in Appendix C.3.

Table 14: **MAGAT integration under the E4b protocol.** The control is the GPU-resident JAX/XLA–PyTorch path. Times are marginal medians and speedups are medians of within-process ratios. First decisions include cache construction; steady decisions use the warm cache.

Contract / phase	Envs	Control (ms)	Ours (ms)	Ratio
Same GPU, steady	16	437.766	9.541	45.65×
Same GPU, steady	64	482.906	37.511	12.87×
Same GPU, steady	256	595.631	157.234	3.79×
Same GPU, first decision	16	526.405	14.611	36.00×
Same GPU, first decision	64	1,518.333	53.483	28.40×
Same GPU, first decision	256	5,511.454	220.431	25.00×

At $A = 256, N = 256$, the CUDA path takes 157.234 ms per complete decision, compared with 595.631 ms for JAX/PyTorch, a paired $3.79\times$ improvement. Across accepted steady workloads, the CUDA path is 3.79–$45.65\times$ faster. The range reflects how the control amortizes overhead as

workload grows and shared policy execution occupies more of each complete decision. The main text reports the largest accepted steady workload.

First decisions also construct the goal-conditioned energy maps. At $N = 256$, this preparation takes 5,146.858 ms, or 93.4% of the control's first-decision latency. The corresponding $25.00\times$ ratio measures initialization through the first action and completed transition. Subsequent decisions use the warm cache and are reported separately in the table.

C.3 CPU-DEPLOYMENT MAGAT COMPARISON (E4)

E4 pairs eight-process POGEMA driving MAGAT with the corresponding JointLoom resident path. At 256 environments, the measured latencies are 55,225.307 and 229.926 ms, with a reported $240.19\times$ speedup (Table 15). This is the CPU-to-resident deployment comparison. E4b instead compares GPU-resident implementations. The runners use different phase boundaries and diagnostic overheads, so each result keeps its original pair of latencies.

Table 15: **CPU/POGEMA–MAGAT and resident MAGAT under E4.** The latency pair and its ratio use the original E4 timing scope, separately from the E4b GPU-resident comparison.

Environments	POGEMA-8 (ms)	JointLoom (ms)	Ratio
256	55,225.307	229.926	$240.19\times$

C.4 MAPF-GPT TOKEN-BACKEND INTEGRATION

To test reuse beyond the MAGAT graph input, MAPF-GPT exercises the same reconstruction interface with tokens and action history. E4c extends the GPU-resident comparison to this second input specification. The JAX control constructs exact `uint16` cost-to-go maps and neighbor tokens and retains state and the goal cache on GPU. Both paths execute the released MAPF-GPT-2M checkpoint in PyTorch with 256-agent microbatches. The timed loop includes input construction, policy execution, action selection, and transition, with no bulk host round trip.

One E4c gate per formal workload compares exact cost and neighbor tokens, logits, argmax actions, history, and full state for five consecutive steps. All three gates pass. The 36 timing rows form 18 path pairs with matching action-trajectory, history, and final-state hashes. Table 16 separates first decisions from the decision after one matched warm step.

First-decision ratios include goal-cache construction and reach 1.405–$1.547\times$. After warm-up, shared Transformer execution dominates both accelerator-native paths. The CPU-deployment comparison below measures the additional effect of moving environment transitions to GPU.

A separate E4g campaign compares a stronger hybrid CPU deployment path with the same resident CUDA path. The control advances POGEMA environments on CPU, constructs tokens on GPU, and runs the same MAPF-GPT-2M policy there. All three five-step semantic gates pass with zero token, logit, action, history, state, or termination mismatches. Each cell in Table 17 has three fresh paired processes; all 18 pairs match initial inputs, checkpoint, actions, history, and final state. The cold and steady CPU-deployment ratios are reported separately from the E4c GPU-resident comparison above.

The complete-decision comparison includes the same Transformer policy on both paths. Online MAPF-GPT training exposes the input-side benefit directly: compact streaming reduces consumer wait per sample by 86.7% against cached CPU prefetch (Table 29). Together, the decision and training measurements locate input delivery within two full policy workflows. Table 17 uses the verified v2 paired campaign.

C.5 LATENCY INTERPRETATION AND DIAGNOSTIC PROFILES

The MAGAT and MAPF-GPT profiles explain their different gains through the costs shared by both paths and the costs changed by input delivery. For a serialized fixed-work comparison, let C denote shared policy or optimizer work and R denote input construction, transfer, replay, transition, and synchronization:

Table 16: **MAPF-GPT integration with resident state and goal caches on both paths.** Both paths share the PyTorch policy. First decisions include cache construction; warm decisions follow one matched full step. Times are marginal medians and speedups are medians of within-process paired ratios.

Phase	Agents	Envs	Materialized (ms)	Resident (ms)	Ratio
First decision	128	16	320.400	206.498	1.547×
First decision	128	64	946.170	672.037	1.405×
First decision	256	16	524.668	360.536	1.456×
Steady after one	128	16	152.288	150.120	1.015×
Steady after one	128	64	607.265	604.577	1.004×
Steady after one	256	16	304.282	302.080	1.007×

Table 17: **MAPF-GPT complete decisions against CPU POGEMA with GPU tokens and the same GPU policy.** Marginal median latencies and median paired speedups over three fresh-process repetitions per cell.

Phase	Agents	Envs	CPU / resident (ms)	Paired ratio
First	128	16	222.2 / 214.9	1.027×
First	128	64	713.3 / 680.7	1.048×
First	256	16	384.9 / 369.9	1.041×
Steady	128	16	158.6 / 151.4	1.048×
Steady	128	64	639.2 / 608.8	1.050×
Steady	256	16	318.1 / 304.1	1.046×

$$T_{\text{early}} = C + R_{\text{early}}, \qquad T_{\text{late}} = C + R_{\text{late}}, \qquad S = \frac{C + R_{\text{early}}}{C + R_{\text{late}}}. \tag{8}$$

Here early and late label the compared input-construction paths, not first and warm decisions. In an already GPU-resident path, bulk host transfer is absent and the remaining difference lies in builders, transitions, and framework execution. The warm MAPF-GPT decisions are dominated by shared Transformer execution. Streaming additionally depends on producer and learner service rates and queueing. Equation 8 locates speedup in costs that change between input paths; reported ratios use paired workloads and timers.

To locate this work in the implemented paths, the profiles in Figure 6 break down host-side time in separate instrumented runs. Host NVTX wall ranges and GPU-projected durations are different timing domains. Tables 18 and 19 report them separately; values across those domains must not be summed.

Table 19: **Host and GPU timing by stage in the paired diagnostic traces.** M is the materialized JAX/Torch control; R is the resident path. Host ranges and GPU-projected durations are reported separately and must not be added across columns.

Path	Stage	Host range (ms)	GPU proj. (ms)	Launches
M	View/graph	342.823	2.217	24
M	Model	148.050	152.001	238
M	Action/transition	275.873	0.299	6
M	Framework sync	2.168	0.000	0
M	Unassigned residual	0.092	271.699	0
R	View/graph	0.036	4.979	6
R	Model	153.492	152.712	250
R	Action/transition	0.063	0.070	3
R	Framework sync	4.174	0.007	5
R	Unassigned residual	0.025	0.001	0

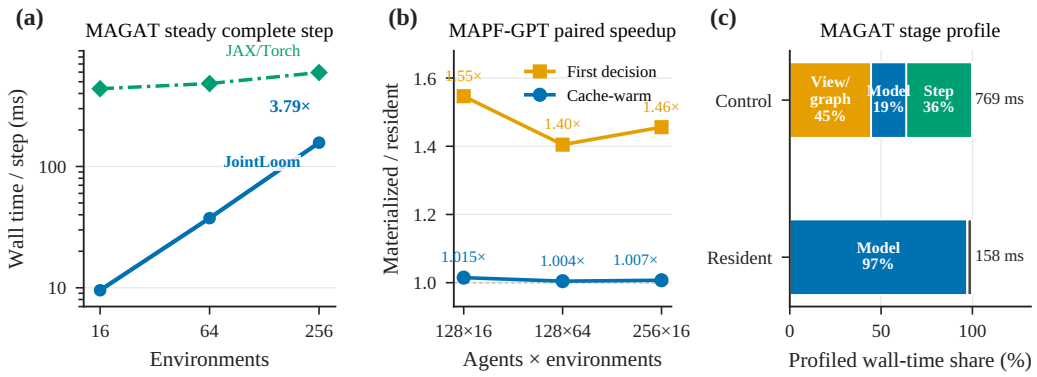

Figure 6: **Supplementary complete-decision measurements and profiles.** (a) MAGAT warm-step latency, with values in Table 14. (b) MAPF-GPT first/warm-step speedups against its GPU-resident control. (c) Independent MAGAT host-side timing profiles. Formal latency measurements use separate uninstrumented runs.

Table 18: **MAGAT diagnostic profiles at $A = 256, N = 256$.** Durations are milliseconds. Host wall time and GPU-projected work are separate timing domains. Unassigned projected work lies outside the projections of the annotated host leaves.

Path	Outer host wall	GPU projected	Launches	Unassigned projected
JAX/Torch materialized	769.005	426.216	268	271.699
CUDA resident	157.790	157.768	264	0.001

In the materialized path's host ranges, view/graph construction takes 342.823 ms, model execution 148.050 ms, action/transition 275.873 ms, and synchronization 2.168 ms. The resident path spends 153.492 ms around the model and 4.174 ms around synchronization; its other host ranges are below 0.1 ms. These ranges show where the host-visible delay occurs.

JAX device work can outlast the short host leaf that enqueues it, leaving 271.699 ms outside the leaf projections. That GPU-projected residual remains unassigned. Launch totals are similar (268 versus 264), while operator work and synchronization differ. Instrumentation also raises the materialized host wall relative to the formal 595.631-ms median. The profiles support stage-level interpretation; the uninstrumented paired runs determine the reported speedups.

E4d extends the first-decision and warm-step protocols across the declared MAGAT and MAPF-GPT workload matrices. It records formal uninstrumented latencies separately from normalized host-side timing profiles. Figure 7 combines these complementary views and keeps rejected workloads visible.

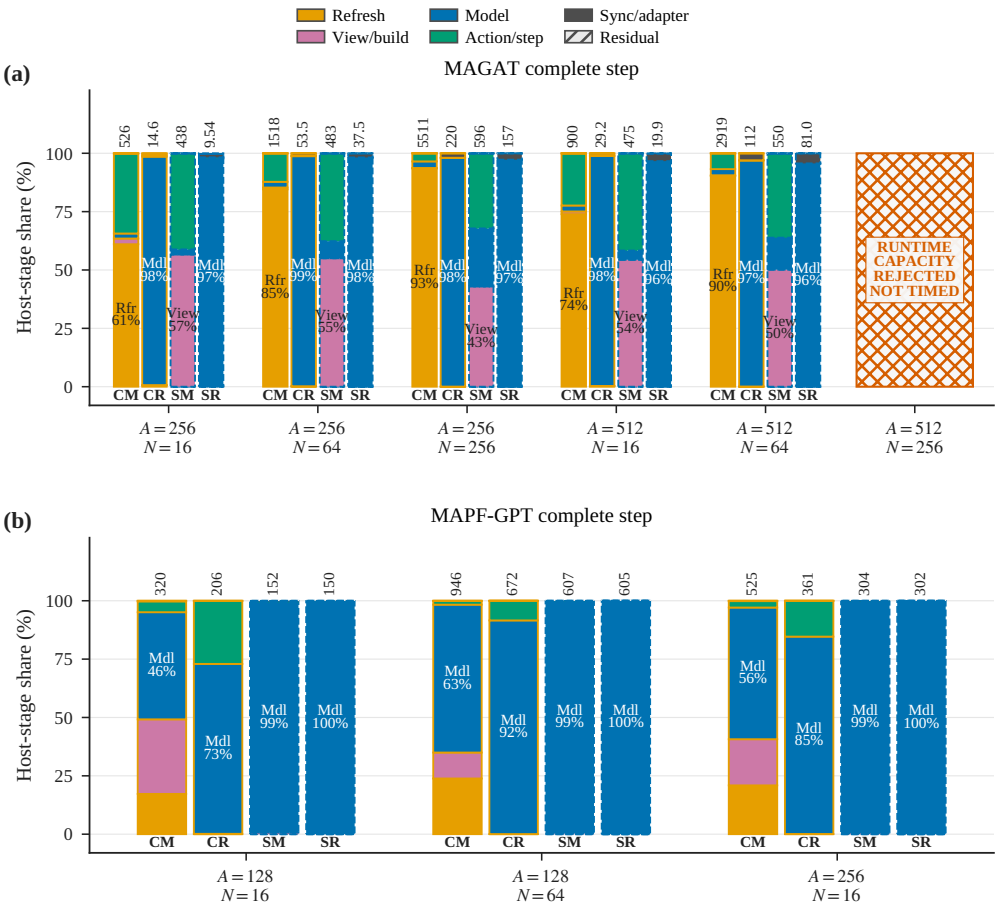

Figure 7: **Host-side time breakdown across the integration workloads.** Panels (a)/(b) show MAGAT/MAPF-GPT. C/S denote first/steady-after-one steps; M/R denote materialized/resident paths. Each stack averages three independently normalized host profiles and displays the residual. Rotated labels give separately measured uninstrumented median latencies from three fresh processes; min–max ranges are retained in the figure sidecar. The crosshatched $A = 512, N = 256$ MAGAT cell is an allocation rejection for both paths and phases, not a timing measurement. Action/step combines action selection and transition. The shares describe host ranges rather than exclusive GPU-kernel time.

As MAGAT workload grows, cache construction occupies more of the materialized first decision, while the model occupies more of its warm host-side time. The resident path is model-dominated at accepted points. Warm MAPF-GPT profiles are dominated by shared Transformer execution on both paths. Annotated host ranges account for 99.717–99.998% of host wall time, and each profile retains its residual. These shares describe host-visible execution rather than exclusive GPU-kernel time or hardware arithmetic utilization.

The rejected workload in Figure 7 also shows why allocation must be checked separately from timing. At $A = 512, N = 256$, the known-byte subtotal is below the nominal budget, yet both full-path processes fail during allocation. The recorded failure is a `CUDACachingAllocator`/NVML assertion. The workload is therefore reported as a runtime-capacity rejection with no timing pair. The preflight rule in Equation 10 can reject excessive known storage, but model, library, and allocator workspace still affect admission.

D Pipeline Sensitivity and Mechanism Controls

The main ablation separates reconstruction from buffering. The following controls explain when buffering helps by varying expert supply, ring depth, batch size, topology mix, and transfer triggering (Figure 8). Here P counts producer workers, and B counts agent rows per optimizer update; B_{env} in Appendix G counts environments per policy microbatch. At $N = 8, A = 256, B = 1024$, one admitted 2,048-row stage supplies two optimizer rounds, as described in Appendix B.7.

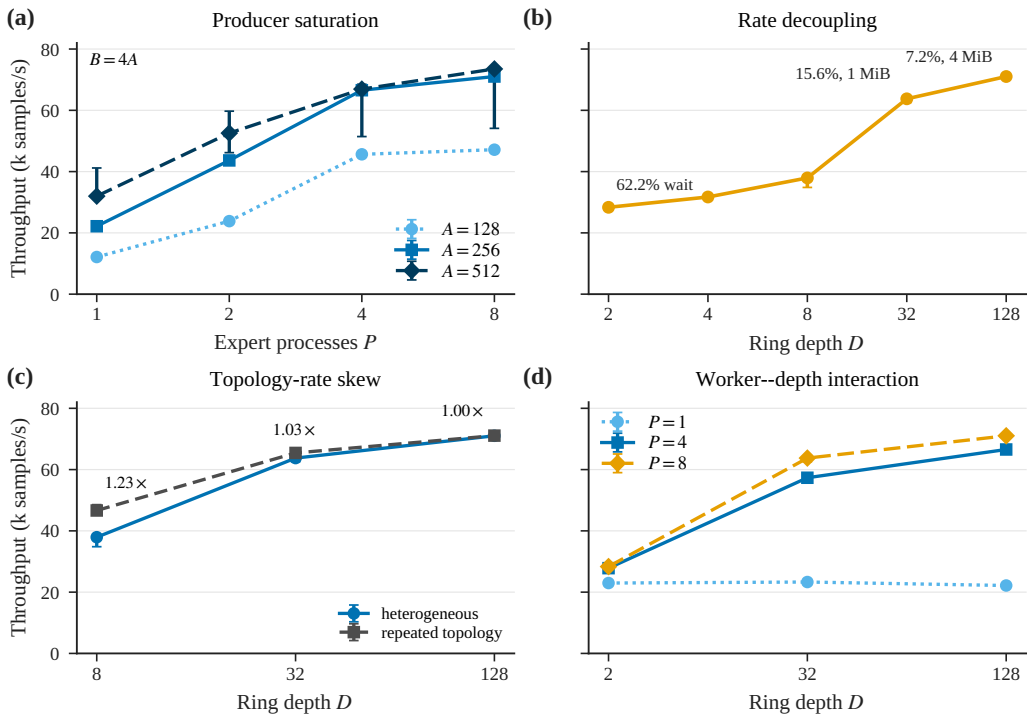

Figure 8: **Expert supply, buffer capacity, and their interaction.** (a) Worker scaling at $B = 4A$. (b) Ring depth at $A = 256, N = 8, P = 8, B = 1024$; depth 32 reaches 89.8% of the depth-128 throughput with 1 MiB of ring storage. (c) One repeated versus eight distinct Maze layouts: the throughput ratio contracts from $1.23\times$ at depth 8 to $1.00\times$ at depth 128. (d) The $P \in \{1, 4, 8\}$, $D \in \{2, 32, 128\}$ factorial measures how buffering changes the benefit of extra workers. Points are medians over seeds 42–44 and whiskers are min–max. Wait share divides marginal median wait by marginal median wall time. Validation, checkpoint I/O, initialization, teardown, and NVML sampling are outside the timer.

D.1 Producer Parallelism

The producer scan uses eight logical environments, 1,024,000 samples per row, and three paired seeds. Batch size scales with agent count. Increasing LaCAM workers from one to eight raises throughput and reduces learner wait (Table 20). Four workers are near the measured knee; eight provide a smaller further gain at the larger batches.

Table 20: **Training throughput as expert-worker count increases.** P1–P8 denote worker counts; throughput entries are samples/s. Four workers are near the observed knee, with smaller further gains at eight workers.

Agents	Batch	P1	P2	P4	P8	P8/P1
128	512	12.1k	23.8k	45.6k	47.1k	3.89×
256	1,024	22.2k	43.6k	66.6k	71.0k	3.20×
512	2,048	32.0k	52.6k	66.9k	73.5k	2.30×

Median consumer wait falls from 75.4% to 4.8% at 128 agents, from 70.1% to 7.2% at 256 agents, and from 59.6% to 8.0% at 512 agents. Some 512-agent seeds enter the configured LaCAM timeout/retry path. The reported medians retain this variation across runs.

D.2 Ring depth and optimizer granularity

Holding producer count fixed isolates the effect of buffering. At $A = 256$, eight environments, eight producers, and $B = 1024$, each stage supplies two consumption rounds. Across three paired seeds, increasing depth from 2 to 128 raises throughput from 28.3k to 71.0k samples/s and reduces median consumer wait from 62.2% to 7.2%. Depth 32 reaches 63.8k samples/s, or 89.8% of the depth-128 endpoint, with 1 MiB of raw ring storage; depth 128 uses 4 MiB.

With depth fixed at 128 and the sample budget at 1,024,000, increasing B from 256 to 2,048 raises throughput from 27.2k to 80.7k samples/s. Each batch contains whole configurations; B determines how many rounds consume a stage. Peak allocation increases from 0.20 to 0.91 GiB. This experiment measures a throughput–memory trade-off at fixed sample count. Larger batches perform fewer optimizer updates. The matched-budget quality study instead holds batch size fixed (Appendix E).

D.3 Maze-layout-induced supply variation

To test whether deeper buffering also absorbs differences between maps, the heterogeneous condition uses eight distinct Maze layouts. The homogeneous control repeats one Maze layout with independently seeded starts and goals. Both conditions remain within the Maze family. At depth 8, the homogeneous case is $1.23\times$ faster (46.6k versus 37.9k samples/s). The ratio falls to $1.03\times$ at depth 32 and $1.00\times$ at depth 128. The deeper ring absorbs layout-dependent variation in producer runtimes. The control isolates within-family layout variation in expert supply and ring buffering.

D.4 Joint worker–depth control

To test how buffering changes the value of extra producers, we vary $P \in \{1, 4, 8\}$ and $D \in \{2, 32, 128\}$ jointly at $A = 256, N = 8, B = 1024$. Throughputs for $P = 1, 4, 8$ are 23.0k, 27.8k, and 28.3k samples/s at depth 2; 23.3k, 57.3k, and 63.8k at depth 32; and 22.2k, 66.6k, and 71.0k at depth 128. The separate scans above intersect only at $P = 8, D = 128$; this factorial directly measures their interaction.

The P4/P1 ratio increases from $1.21\times$ to $2.46\times$ and $3.00\times$ as depth grows; P8/P4 stays between $1.02\times$ and $1.11\times$. A shallow ring masks most of the benefit of additional expert supply. Deeper buffering makes four workers useful, with smaller gains beyond that point. The factorial shows how ring capacity converts additional expert supply into learner throughput.

D.5 Buffer depth versus transfer trigger

The preceding tests vary buffer capacity; a separate control checks whether the transfer trigger explains its benefit. The 2×2 control in Table 21 varies ring depth and the transfer trigger while retaining the same shared producer ring. The consumer either initiates H2D synchronously or uses background-copy prefetch. This isolates buffer capacity from the choice of copy trigger.

Table 21: **Buffer depth and transfer trigger.** A deeper ring improves throughput with either trigger. Matched-depth throughputs are comparable for this compact payload.

Transfer trigger	Depth	Samples/s	Consumer wait
Consumer synchronous H2D	2	28.4k	62.3%
Consumer synchronous H2D	128	72.3k	7.4%
Background copy stream	2	28.3k	62.2%
Background copy stream	128	71.0k	7.2%

Deep-ring gains are $2.55\times$ with synchronous H2D and $2.51\times$ with background copy. At fixed depth, async/sync throughput is $0.999\times$ for depth 2 and $0.982\times$ for depth 128. Buffer capacity improves supply under either trigger; matched-depth throughput remains comparable across copy triggers. This identifies ring capacity as the active supply mechanism for the measured compact payload.

D.6 Same-stack GPU view lifetime diagnostic

A further fixed-record MAGAT diagnostic isolates the timing of GPU input materialization within one CUDA builder, PyTorch model, and optimizer. At $E = 4$, batch 2,048 and 512 updates, three fresh processes per path replay the same frozen stages and initial model. The paths construct views on demand, construct synchronously before the update, or prepare one batch ahead on a separate CUDA stream. Table 22 isolates the fixed-work effect of view timing and storage lifetime.

Table 22: **E4 same-stack GPU materialization control.** Ratios and min–max ranges are paired by interleaved repetition against on-demand construction. Peak memory differences measure PyTorch allocations.

Input timing	Median samples/s	Paired ratio	Added peak MiB
On demand	82,429	$1.000\times$	0
Synchronous prebuild	82,527	$0.995\times$ (0.994–1.004)	5.34
One-batch async prebuild	84,042	$1.021\times$ (1.016–1.021)	5.34

The ratio of marginal median throughputs need not equal the median paired ratio. A separate four-stage ownership probe copies only the tensor values needed by MAGAT forward/backward, excluding unused builder capacity buffers. All saved views remain unchanged after later builds and match fresh reconstruction. Each E4 compact stage occupies 32,768 bytes, versus 6,505,368 bytes for an independently owned view; E8 uses 65,536 versus 13,007,136 bytes. Thus early retention costs about $198\times$ as many bytes per stage in these shapes, while one-batch overlap yields about 2% extra E4 update throughput. The storage and timing observations explain why the system constructs views near consumption.

E Training Budget and Checkpoint Selection

The fixed-work controls measure delivery efficiency; the matched-budget study asks how that efficiency affects training progress. It trains MAGAT+ through compact streaming and the host-expanded synchronous baseline. Both use cross-entropy loss, the same optimizer configuration and cosine schedule, four LaCAM expert workers, four 65×65 Maze maps, 512 agents per environment, and a 2,048-row optimizer batch. The study has three training seeds and four frozen validation trajectories. Here $N = 4$ and $NA = B = 2048$, so each stage supplies one update. The separate sensitivity study exercises multi-round consumption.

The baseline expands observations and PyG inputs on CPU and transfers them synchronously, with one-stage backpressure. Compact streaming sends state records to GPU builders and buffers expert supply. Figure 5 labels these paths "CPU synchronous" and "JointLoom," respectively. Both use the shared online-expert protocol, separately from the released MAGAT+ offline pipeline.

Both timers begin after worker, CUDA, model, and validation initialization. Periodic validation and checkpoint writes count toward the budget. The final forced checkpoint and teardown follow the training timer. Validation and training-loss observations form the learning-progress analysis.

Each paired seed has a 3,035.921933-second maximum budget. Streaming stops at its 100,000-update cap before that limit; the host-expanded path stops after the first completed update that crosses it, producing the small overruns in Table 24. Periodic validation runs approximately every 300 seconds, and each path retains the top three checkpoints by the same overall-accuracy criterion. Validation frequency is matched in wall time, so update intervals differ with throughput.

A preflight runs 20 complete stages per seed through both paths with hashing enabled. Each hash covers the full 4×512-row compact stage, including identifiers, positions, goals, expert actions, and refresh state. All 60 stage hashes and all three initial models match (Table 23).

Table 23: **Matched initial models and compact-stream prefixes.** Each stage hash covers the complete compact record. The gate checks 20 stages per training seed before the formal timed runs.

Seed	Initial model	Compared stages	Exact stage hashes
0	Match	20	20/20
1	Match	20	20/20
2	Match	20	20/20
Total	3/3	60	60/60

Stage q is publishable only after the q-th frontier from every environment is ready. Stages are consumed in sequence, so faster producers can write ahead but cannot replace a slower environment's contribution. Formal runs disable hash capture to avoid adding device-to-host synchronization to the timed streaming path. The tested prefix is byte-exact; later time-bounded LaCAM calls may differ across processes. Ordered consumption still preserves the scheduled environment mixture across the 2,430–2,450 shared formal stages.

Table 24: **Training progress and checkpoint selection under a common maximum budget.** Every pair receives 3,035.921933 seconds and the same selection rule. P/C denote compact streaming / the host-expanded synchronous baseline. P reaches its 100k-update cap; C stops at the first completed update beyond the time limit, accounting for its small recorded overrun.

Seed	Path	Actual wall (s)	Updates	Samples (M)	Best step	Best val.
0	P	2,983.106	100,000	204.800	91,184	78.8994%
0	C	3,037.193	2,430	4.977	2,156	73.8907%
1	P	2,940.697	100,000	204.800	100,000	78.5515%
1	C	3,036.291	2,450	5.018	2,450	73.8777%
2	P	2,946.597	100,000	204.800	92,287	78.6331%
2	C	3,036.642	2,439	4.995	2,402	74.2363%

Streaming completes 100,000 updates and 204.8M agent samples per seed; the baseline completes 2,430–2,450 updates and 4.977–5.018M samples. The fixed-batch update ratios are 40.816–$41.152\times$. Selected-checkpoint validation accuracy improves by 4.397–5.009 percentage points across the three pairs. The next appendix presents the learning trajectories and stronger CPU controls in detail.

F Learning Curves and Stronger CPU Controls

The main training results connect input wait to optimization progress. Here we show the individual learning curves, then strengthen the CPU input path from cached construction to background prefetch under fixed-record and online protocols.

F.1 Recorded observations and update alignment

Figure 5 uses all six original matched-budget runs and adds three cached-prefetch CPU logging runs. Hardware, model, batch size, maximum budget, initial model hashes, and frozen validation trajectories agree across the recorded cohorts. The cached-prefetch reruns additionally evaluate early update checkpoints and record terminal evaluation outside the training timer. Main-figure aggregation uses the common scheduled time-checkpoint sequence from about 300 seconds onward, with terminal observations grouped separately; the complete early observations remain in the individual curves. Compact streaming uses a ring depth of 128; the conventional CPU path uses one-stage backpressure. The complete per-update loss histories are retained. Figure 9 complements the time-axis presentation with recorded validation points and non-overlapping 200-update loss means. The update axis is logarithmic because the stopping counts differ by about fortyfold. Curves end at their last observed update. The update-aligned view shows recorded optimization trajectories alongside fixed-validation points.

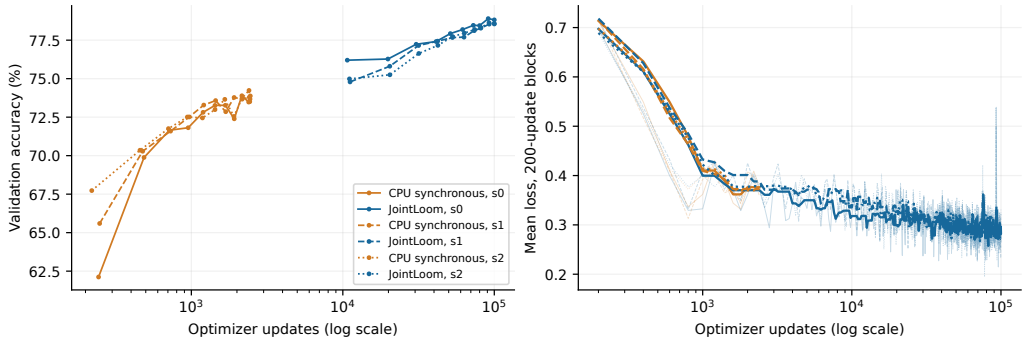

Figure 9: **Learning trajectories versus update count.** Colors distinguish paths and line styles distinguish seeds; every seed is shown separately. Validation points are recorded observations on the same frozen trajectories, without smoothing. In the loss panel, faint lines show disjoint 200-update block means and prominent lines show their trailing five-block median. Raw per-update losses remain in the artifact.

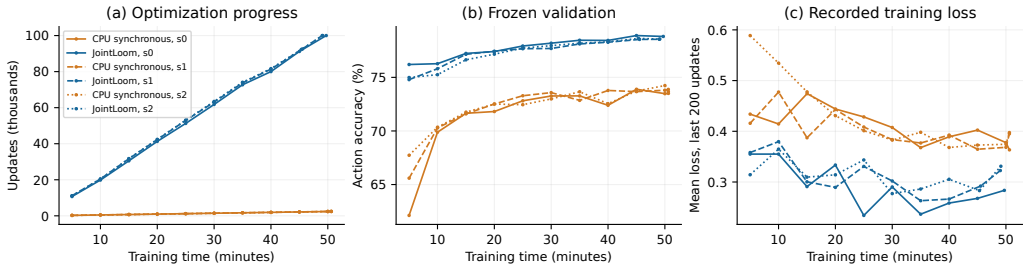

Figure 10: **Individual matched-budget runs behind the main-figure aggregates.** All three seeds per path are shown at their own recorded times. This original view uses the order updates, validation accuracy, and training loss.

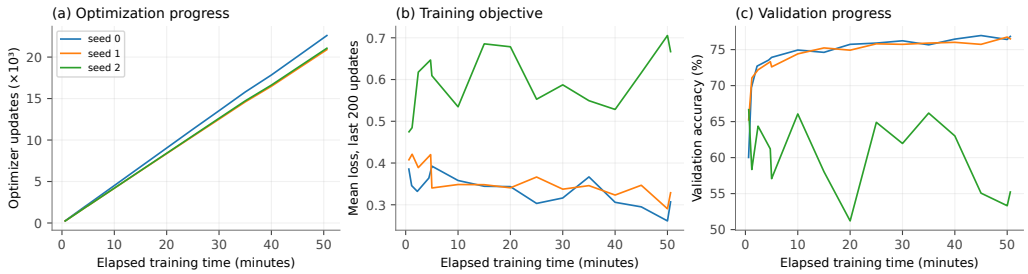

Figure 11: **Individual cached-prefetch CPU logging reruns.** Each seed contributes its own recorded update, preceding-200-update mean loss, and fixed-validation trajectory. Early update-triggered checkpoints are shown here; the main figure uses the common scheduled time checkpoints.

The main figure groups corresponding scheduled validation rounds within each method, then plots the median observed value at the median actual training time; shading spans all three values. At the late original streaming checkpoint, actual observation times differ by up to about 33 seconds; terminal times differ by about 42 seconds. The plot uses recorded checkpoint times throughout. The cached-prefetch series uses complete per-update losses from the logging reruns in all three panels. Its updates and validation points are taken from the same reruns in all three panels. The logging reruns complete 20,893–22,597 updates, matching the update-count range of the paired strong-CPU cohort. Figure 11 shows the individual validation trajectories. Each run records its own online expert stream under the matched generation protocol.

For the time-axis loss panel, the mean of the preceding 200 losses is attached to each checkpoint's recorded training time. The validation panel shows actual accuracy at each observation, including decreases; best-checkpoint accuracy in Table 24 is a separate summary. Four frozen validation trajectories are shared across all nine runs, verified by matching hashes. The training seed is the repetition unit; validation samples and checkpoints remain nested observations within each training run.

F.2 Fixed-record cached CPU controls

Table 25 uses 18 completed runs: two model families, three paths, and three seeds. CPU paths cache reusable inputs; the prefetched path additionally overlaps host preparation through a bounded queue. Compact streaming reconstructs inputs on the GPU. Each paired run uses the same archived records for 512 optimizer updates, with batch sizes and model settings held fixed within a family. MAGAT uses batches of 2,048 rows and ring128; MAPF-GPT uses batches of 1,024 rows and ring4. The cached CPU prefetch uses two workers and queue depth four for MAGAT, and one worker and queue depth two for MAPF-GPT; all paths use seeds 42–44. These measurements isolate fixed-work delivery throughput. The two CPU paths add successive levels of host preparation, and ratios are paired within each fixed-work protocol. The longer online runs address training progress under their own time budgets.

Table 25: **Fixed-record throughput with stronger CPU controls.** Compact-streaming / CPU samples-per-second ratios, min–max over three seeds. Each run completes 512 updates; online training is examined below.

Policy	Cached CPU, synchronous	Cached CPU, prefetched
MAGAT	11.00–11.18×	5.67–5.69×
MAPF-GPT	1.278–1.284×	1.043–1.045×

F.3 Online training against cached CPU prefetch

The three-seed online MAGAT comparison extends the matched-budget study to cached CPU prefetch. It uses the same maximum budget and batch size 2,048, with compact ring depth 512 and

cached-prefetch CPU ring depth 128. Table 26 retains every seed. Compact streaming performs 5.185–5.235× as many updates; Figure 12 shows the corresponding validation trajectories.

Table 26: **Online MAGAT with a stronger CPU baseline.** P: compact ring512; C: cached-prefetch CPU ring128. All runs use approximately 3,036 seconds. Best and final accuracy use the common frozen validation set.

Seed	P/C updates	Update ratio	Best P/C (%)	Final P/C (%)
0	117501/22446	5.235×	78.879/76.552	78.770/76.018
1	118177/22782	5.187×	78.769/76.958	78.630/76.936
2	117435/22647	5.185×	72.697/72.585	70.541/62.418

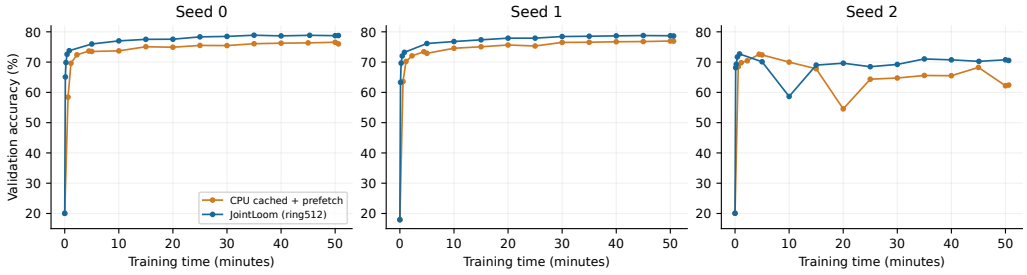

Figure 12: **All three seeds of the stronger-baseline online study.** Accuracy is shown at recorded times for all three paired seeds.

These runs retain the original cosine schedule with a 100k-update period; compact runs continue beyond that point. The complete checkpoint record is shown in Figure 12; per-update loss curves come from the matched-setting logging cohort in Figure 5.

F.4 Online strong-CPU throughput and fixed resource budget

Table 27 gives the individual observations behind Table 1. CPU input work uses a cached H8/Q16 pool with pinned-memory lookahead; both paths use the same source-ring depth 512, same 512 agents per environment, model initialization and source manifest within a seed, and a 300-second online training window. H8 denotes eight input-builder processes; Q16 is its separate prefetch queue. The E8/E16 workloads repeat four Maze layouts twice/four times, respectively. Each run follows the same LaCAM generation protocol and records its own online trajectory. The paired ratio measures complete-system throughput. The E8 and E16 workloads scale batch size with expert count.

Table 27: **Three-seed online MAGAT strong-CPU control.** Throughput is thousands of agent samples/s. Input wait is the consumer's fraction of the timed window spent acquiring input, CPU/JointLoom.

Experts	Batch	Seed	CPU / JointLoom (k/s)	Paired ratio	Input wait (%)
8	4096	0	42.48 / 81.56	1.920×	49.0 / 2.5
8	4096	1	41.05 / 81.43	1.984×	50.4 / 1.3
8	4096	2	37.79 / 80.20	2.122×	53.4 / 2.9
16	8192	0	39.97 / 77.87	1.948×	49.6 / 2.4
16	8192	1	34.91 / 77.17	2.211×	55.4 / 2.4
16	8192	2	37.41 / 77.79	2.080×	52.8 / 2.3

For every paired online row, let q be completed agent samples/s and f the measured fraction of training-loop time spent acquiring the next input. The interval ends before the training call; compact-path GPU view construction is part of the remaining loop work. Input wait per completed sample is f/q. We calculate the reduction in wait per sample as $1 - (f_J/q_J)/(f_C/q_C)$ and the Amdahl

bound $S_{\mathrm{A}} = (1 - f_{\mathrm{C}})^{-1}$. The latter assumes zero acquisition wait and unchanged remaining loop time per sample. From the six MAGAT seed pairs in Table 27, median wait reduction per sample is 97.5% (97.3–98.7%) at eight experts and 97.9% (97.5–98.0%) at sixteen. The median training-loop speedups are $1.98\times$ and $2.08\times$, compared with median Amdahl bounds of $2.02\times$ and $2.12\times$. Each derived quantity uses one seed pair before aggregation; marginal medians in the main table may come from different seeds.

All twelve path runs pass trace, sample-count, loss, and semantic checks. The seed-0 pairs are hash-verified prior results, and seeds 1–2 complete the three-seed comparison.

A three-seed, 180-second fixed-resource comparison holds E8, batch 4096, D512, and the same task under CPU-affinity budgets of eight or sixteen cores. The CPU input pool is H8/Q16 in both cases, selected by a declared 45-second calibration with seed 99. Within each seed and affinity set, both pipelines share the model initialization and physical CPU set. Table 28 gives medians and paired ranges: JointLoom reaches $2.44\times$ throughput with eight cores and $2.05\times$ with sixteen. Consumer input wait falls from about 55% and 53% to about 4% in the respective budgets. Median H2D payload falls from roughly 3,308 to 16 bytes per sample. Ratios are paired within each affinity set on the host's hybrid-core topology.

Table 28: **Three-seed fixed CPU-affinity MAGAT online comparison.** Same GPU, batch, task, and 180-second window within each pair. Throughput and wait are medians; the paired ratio includes min–max.

CPU budget	Throughput CPU/JointLoom (k/s)	Paired ratio	Input wait CPU/JointLoom (%)
8 cores	32.74 / 79.03	$2.438\times$ (2.388–2.483)	54.9 / 4.2%
16 cores	38.63 / 79.06	$2.053\times$ (2.047–2.077)	53.0 / 4.2%

The eight-core median baseline spends 54.9% of its loop acquiring input, giving a $2.22\times$ Amdahl bound for removing wait alone. The measured $2.438\times$ throughput gain exceeds this fixed-non-wait reference, as do all three eight-core seed pairs. Reductions in non-wait cost add to the input-wait savings under this CPU budget.

The analogous MAPF-GPT online control holds four expert environments, batch size 1024, ring depth four, the MAPF-GPT-2M policy, and seed 42 fixed within each CPU-affinity pair. The cached CPU path uses H1/Q2. Table 29 reports consumer input-acquisition wait alongside training-loop throughput; wait includes CPU construction or prefetch delays encountered when obtaining the next batch.

Table 29: **MAPF-GPT online input wait and training-loop throughput.** Seed 42 is repeated under eight- and sixteen-core affinity budgets. Each path runs for approximately 180 s with the same model, batch, experts, and initial state within its pair. Values are cached CPU prefetch / JointLoom; wait share divides each path's input-acquisition wait by its measured training-loop time.

CPU cores	Input wait (s)	Input wait (% of wall)	Throughput (k samples/s)	JointLoom/CPU throughput
8	9.17 / 1.28	5.09 / 0.71	3.657 / 3.815	$1.043\times$
16	9.03 / 1.25	5.01 / 0.70	3.661 / 3.815	$1.042\times$

Across both CPU budgets, input-acquisition wait per sample falls by about 86.7%. The compact record reduces H2D payload from 1,033 to 26 bytes per sample. The cached CPU path spends about 5% of its timed loop acquiring input, so the remaining model/optimizer and other work governs the training-loop rate. At eight-core affinity, the Amdahl bound is $1.054\times$ and the observed speedup is $1.043\times$. By comparison, the MAGAT controls above begin with about half of their timed window in input acquisition and consequently show a larger training-loop gain.

F.5 Additional buffer-depth diagnostic

A short online diagnostic varies ring depth with the model and batch size fixed. Table 30 reports one 180-second run per configuration (seed 99), using a separate instrumented runner. This sensitivity sweep locates the useful ring-depth range; the three-seed comparison above supplies the repeated online throughput estimate. Compact throughput increases between depths 128 and 512 and changes little at 1024; the cached CPU path remains near 15k samples/s. This motivates the separate ring512 study, while the original ring128 results retain their recorded settings.

Table 30: **Short buffer-depth diagnostic.** Throughput is in thousands of agent samples/s. Raw MAGAT ring storage is $2048 \times D \times 16$ bytes; model, caches, and prefetch queues are excluded. Compact wait measures time spent acquiring the next input.

Depth	Raw ring (MiB)	Compact (k/s)	CPU prefetch (k/s)	Compact wait (s)
128	4	73.86	15.08	29.26
512	16	87.88	15.07	1.45
1024	32	87.96	15.21	1.44

F.6 Supplementary checkpoint and search records

The evidence archive retains exploratory guided-search and standalone-policy evaluations with their deployment versions. Its checkpoint inventory distinguishes the July single-run model from the August three-seed cohort. The finite-budget training analysis uses scheduled checkpoints and frozen validation trajectories from the reported training runs. The next section accounts for the memory limit observed in the resident scaling experiment.

G Memory Use and Capacity Limits

The memory limit in Appendix C follows from allocations that persist as workload size grows. Using the object lifetimes in Table 6, we separate authoritative state, derived caches, reusable view backing, parameters, and model transients:

$$M_{\text{req}} = M_{\text{auth}} + M_{\text{cache}} + M_{\text{backing}} + M_{\text{params}} + M_{\text{transient}}(B_{\text{env}}, A). \tag{9}$$

Here $B_{\text{env}} \leq N$ counts environments per policy microbatch; B in the training protocol counts agent rows per update. Microbatching reduces model transients while full-instance state and view backing remain allocated. We measure PyTorch peak allocation and deduplicate tensor storage in a separate census. These counters describe this implementation's storage, rather than device-wide usage across PyTorch, JAX, and POGEMA.

For the current MAGAT layout, let $d = 13$ denote observation diameter and $W \times H$ the allocation envelope. The known-storage subtotal is

$$M_{\text{known}}(N, A, W, H) = NA(WH + 16d^2 + 16) + 28NA^2 \quad \text{bytes.} \tag{10}$$

The terms cover byte-valued energy maps, four-channel float32 node storage, eight uint16 compact fields, and worst-case edge backing. Each reserved edge uses two int64 indices and three float32 attributes. A runtime count and prefix offset compact valid edges to logical length E, but physical backing is reserved for NA^2 edges. Its per-call contents can change while the allocation persists, so inference microbatching leaves this term unchanged.

For the existing allocation scheme, the S3 capacity check rejects a request when $M_{\text{known}} + R > F$, where F is runtime free device memory and $R = 1$ GiB. The subtotal covers known allocations; model, optimizer, CUDA-library, and allocator workspace require additional memory. The rule can reject an excessive request before allocation, while a request below the subtotal threshold still needs runtime admission.

Table 31: **MAGAT policy microbatching and memory use.** Both rows use $A = 256, N = 256$. Microbatching changes peak model-call allocation while preserving the persistent inventory.

Configuration ($A = 256, N = 256$)	Persistent	Peak allocated	Outcome
Full policy batch	1.630 GiB	8.532 GiB	Complete
16-environment microbatch	1.630 GiB	2.073 GiB	Complete

At $A = 256, N = 256$, using 16-environment policy microbatches reduces peak allocation from 8.532 to 2.073 GiB, a 6.459-GiB decrease or $4.116\times$ ratio (Table 31). Persistent storage remains 1.630 GiB. The microbatch size is a fixed experimental setting; state and full-batch graph backing remain allocated on both paths.

In the separate S3 census at $A = 256, N = 4$, the depth-128 pinned host ring uses 2.000 MiB and the 1,024-row compact GPU staging tensor uses 16 KiB. These transport allocations are small relative to the goal maps and model workspace. Table 32 decomposes the known storage for the larger $A = 512, N = 1024$ request.

Table 32: **Known-storage preflight at $A = 512, N = 1024$.** The current runtime rejects this configuration before attempting the allocation. Values are mandatory known storage, separate from additional model and library workspace.

Mandatory component	Scaling	Lifetime / reduced by inference microbatch?	GiB
Compact packed state	$O(NA)$	Persistent / no	0.008
Derived energy maps	$O(NAWH)$	Derived persistent / no	8.000
MAGAT edge attributes	$O(NA^2)$	Reused view backing / no	3.000
MAGAT edge indices	$O(NA^2)$	Reused view backing / no	4.000
MAGAT node storage	$O(NAd^2)$	Reused view backing / no	1.320
Mandatory known total	—	Sufficient rejection subtotal	16.328

The known subtotal is 16.328 GiB. The runtime exposes 15.194 GiB free; a 1.000-GiB reserve leaves a 2.134-GiB shortfall, so S3 rejects the request before attempting allocation. This explains the allocation failure retained by the earlier E2 sweep. Goal-derived state and worst-case MAGAT graph capacity account for most of the boundary, rather than compact ring storage.

H Extended Related-Work Context

Table 5 classifies policy inputs. The related systems below clarify how existing work produces and delivers inputs, and why our evaluation separates resident execution from expert streaming.

Learned-MAPF inputs and consumers. PRIMAL and SCRIMP use local grids and communication features (Sartoretti et al., 2019; Wang et al., 2023); MAGAT and HMAGAT use pairwise graphs or directed hypergraphs (Li et al., 2021; Jain et al., 2026a). MAPF-GPT uses local observations, ordered neighbors, and bounded action history (Andreychuk et al., 2025). Other interfaces consume action rankings, map-wide tensors, or joint spatiotemporal drafts (Veerapaneni et al., 2024; Tang et al., 2025; Wang et al., 2026). Table 5 distinguishes the model views and additional policy or planner state in these families. JointLoom implements execution and expert-data delivery for the MAGAT graph and MAPF-GPT token interfaces. New input specifications connect through backend builders; state producers reuse existing builders by supplying the specified state and context.

Accelerated environments. For the resident workflow, batched GPU environments provide the relevant systems context: they colocate independent simulations and policy computation (Shacklett et al., 2021; Makoviychuk et al., 2021; Freeman et al., 2021; Bonnet et al., 2024). WarpDrive provides a resident multi-agent data store, GPUDrive links GPU simulation to learning outcomes, and JaxGCRL emphasizes time-to-result (Lan et al., 2022; Kazemkhani et al., 2025; Bortkiewicz et al., 2025). JaxRobotarium studies accelerated multi-robot simulation, learning, and deployment with

continuous dynamics and sim-to-real RL (Jain et al., 2025). JointLoom evaluates complete MAPF policy decisions by timing MAGAT input construction, policy execution, and transition together.

Expert supply, buffering, and replay. For the training workflow, Sample Factory, SEED RL, and SRL separate actors, inference, and optimization with buffers or services (Petrenko et al., 2020; Espeholt et al., 2020; Mei et al., 2024). Reverb provides configurable replay sampling, eviction, and insert-to-sample rate control (Cassirer et al., 2021). Our stream applies ordered stage publication to complete MAPF configurations. Publication waits for every environment sub-block, and optimizer batches take whole configurations over one or more rounds. This couples buffering to explicit input-consistency and storage-ownership rules. The evaluated protocol consumes stages in publication order.

MAPF platforms and training systems. Within MAPF, complementary platforms define tasks and evaluation: POGEMA standardizes grid evaluation; GRACE aligns grid, roadmap, and continuous abstractions; SMART evaluates planned paths under dynamics and execution uncertainty (Skrynnik et al., 2025; Zang et al., 2026; Yan et al., 2026). SILLM and PRIMAL3 develop scalable learned-MAPF solvers and training mechanisms (Jiang et al., 2025; He et al., 2026). JointLoom provides an execution and data-delivery layer for supported policies alongside these model and evaluation platforms. It uses POGEMA as the CPU reference and retains policy or expert settings within each comparison. Our JAX/XLA backend is an internally implemented, semantically matched GPU control; E4 supplies a CPU/POGEMA–MAGAT deployment pair. Their published throughput values provide systems context, while our paired evaluations hold task and timing boundaries fixed.

Kernel-level optimization. At the level of individual operators, Triton generates tiled GPU kernels, and Hector reduces relational-GNN memory traffic through graph-aware intermediate representations (Tillet et al., 2019; Wu et al., 2024). These tools address execution within a builder or model. Our compact-state interface specifies the information that a builder receives, the dependencies of its caches, and the lifetime of its outputs.

I Reproducibility and Evidence Inventory

This inventory links each reported result to its software environment and frozen evidence archive. All experiments use one Intel Core Ultra 9 285K workstation with 24 physical cores and one 16-GiB NVIDIA GeForce RTX 5080. The frozen environment records CUDA 12.8, NVIDIA driver 595.71.05, Python 3.10.20, PyTorch 2.11.0+cu128, and JAX 0.6.2. Tables 33 and 34 map results to their recorded evidence archives.

The paper uses JointLoom as the system name. Frozen runner and archive identifiers retain their original names so that result provenance and file hashes remain stable across manuscript revisions.

Artifact inventory and reproducibility. The evidence tables map reported results to frozen archives. Each archive records its configuration, environment, source reference, and raw or summarized measurements. Figure renderers reconstruct the reported ratios from those records and validate protocol fields and source hashes. Two reproducibility entry points follow the evaluated workflows: resident MAGAT policy execution and compact LaCAM expert streaming for MAGAT+ training. The release validation protocol covers a clean build, semantic checks, figure regeneration, manuscript compilation, and identity/path scans. Third-party dependencies retain their licenses and versioned acquisition instructions.

Table 33: **Evidence archives for correctness, execution, and mechanism controls.** Identifiers name the frozen archives used for these results.

Evidence	Frozen archive identifier	Primary use
A1/B1	`a0b-refreeze-4fbcc29-20260721`	Transition, builder, learning, lifecycle, and fault gates
E2	`e2-matrix-20260722-212312`; `minimal-fast-scaling-256a-20260722-140813`	Joint scaling and baseline context
E3/E4	`e3-stage-20260723-140433`; `e4-deployment-20260723-153000`	Resident profile and paired CPU-deployment result
E4b	`e4b-accelerator-native-complete-20260811-184856/report-v2`	MAGAT same-GPU steady and first-decision integration controls
E4b/profile	`e4b-stage-profile-20260812-182800`	Bilateral MAGAT stage-ownership diagnostic
E4c	`e4c-mapf-gpt-complete-20260812-192257`	MAPF-GPT exact-cache first-decision and steady controls
E4g	`e4g-mapf-gpt-cpu-deployment-20260925-v2`	MAPF-GPT hybrid CPU deployment with GPU tokens and policy
E4d	`e4d-stage-scaling-20260813-133134`	Cross-scale complete-step latency and separate host and GPU timing profiles
F1/core	`f1-ablation-20260728`; `core-ablation-*-20260728`	Representation and one-stage/deep-ring mechanisms
F2–F4	`f2-agent-producer-summary-20260728`; `f3-ring-depth-summary-20260728`; `f4-batch-granularity-summary-20260728`	Producer, ring-depth, and batch-granularity sensitivity
F5–F6	`f5-ring-skew-control-summary-20260728`; `f6-infra-factorial-summary-20260728`	Topology-skew and transfer-trigger controls

Table 34: **Evidence archives for training, resource controls, and supplementary studies.** Identifiers name the frozen archives used for these results.

Evidence	Frozen archive identifier	Primary use
S7	`s7-worker-depth-20260813-140802`	Joint worker-count × ring-depth interaction
View lifetime	`jointloom-gpu-input-prefetch-20260923-v1`	Same-stack prebuild and retained-view diagnostic
E6/S2	`e6-retained-evidence-20260726`; `s2-matched-budget-multiseed-lagat-formal-20260812`	100k stability, matched-budget training, and archived search registry
Strong CPU	`jointloom-magat512-main-20260921-v1`; `jointloom-strong-cpu-learning-20260921-v1`	Long-budget paired study and Figure 5 logging reruns
Online E8/E16	`jointloom-strong-cpu-multiseed-20260924-v1`	Three-seed E8/E16 online throughput and input wait
CPU budget MAGAT	`jointloom-fixed-cpu-budget-e8-b4096-20260925-v1`; `jointloom-fixed-cpu-budget-e8-b4096-multiseed-20260925-v1`	Three-seed E8/B4096 fixed-affinity online training
CPU budget MAPF-GPT	`jointloom-mapf-gpt-fixed-cpu-budget-b1024-20260925-v1`	Token-policy fixed-affinity online control
S1	`paper-supplement-20260806/closed-loop-audit`	Supplementary standalone MAGAT audit
E5-Q	`e5q-mapf-gpt-quality-20260812-193722`	Supplementary MAPF-GPT checkpoint diagnostics
S3	`paper-supplement-20260806/memory-decomposition`	Allocator census and safe capacity preflight

Timing and statistical records. For interpreting those archives, the reported performance measurements distinguish these timing scopes:

1. transition-only execution, excluding input construction and policy inference;
2. E4b cache-warm MAGAT decisions after a matched full-step warm-up and goal-cache initialization;
3. E4b first MAGAT decisions, including initial goal-energy construction;
4. E4c cache-warm MAPF-GPT decisions after one matched warm step;
5. E4c first MAPF-GPT decisions, including exact goal-cache construction;
6. E4 CPU deployment, retaining its original POGEMA-8/resident pair;
7. E4g MAPF-GPT hybrid CPU deployment, separating first and warm complete decisions within its paired runner;

8. fixed-work training, timing optimizer updates at a fixed sample or update count;
9. online supply controls, reporting training-loop throughput and the consumer's input-acquisition wait over their stated windows; and
10. matched training wall time, including periodic validation and checkpointing and excluding the terminal forced checkpoint and teardown.

E4d extends the E4b and E4c workloads and adds separate host-side timing profiles; it retains their first-decision and warm-step timing definitions. The fixed-work training ablations have the workload and timing settings specified in Section 5.3 and Appendix D. Ratios use the relevant paired runner and timing scope. Formal latencies and diagnostic profiles remain separate records.

The E4b publication source is the append-only `report-v2` directory. The superseded report remains in provenance records. The E4g hybrid CPU comparison uses its verified v2 campaign after correction of a history serialization check; v1 and the analyzer record remain archived. Long-run MAGAT and MAPF-GPT rows document completion and backend compatibility. Matched-budget MAGAT training uses three seeds. The stronger fixed-record controls cover both model families; the online strong-baseline MAGAT and fixed-affinity studies each retain all three seeds. Archived policy-quality audits are catalogued under their own protocols; finite-budget progress uses the matched training records. Transition and pipeline results retain their stated repetition ranges. These distinctions are preserved in the manuscript and planned release metadata.

Learning-curve reconstruction records. The training-progress analysis archive retains the original run hashes, per-update losses, checkpoint observations, fixed-work records, and plotting script under `jointloom-training-progress-analysis-20260921-v1`. Figure 5's cached-prefetch series uses the three complete logging reruns under `jointloom-strong-cpu-learning-20260921-v1`; their update, loss, and validation records are kept together in every panel. The renderer and its source-hash ledger distinguish these runs from the earlier ring-512 strong-baseline cohort. Guided-search records retain their deployment-version identifiers. The learning curves use scheduled checkpoints and frozen validation observations from the training cohorts.