LUMILAKE: An Agentic Analytics Engine for AI4Science

Zhengyuan Su, Noppanat Wadlom, Junyi Shen, Peisong Zhang, Ran Luo, Dianbo Liu, Yi Hou, Yicong Huang*, Wentao Wu+, Yao Lu
National University of Singapore, Databricks*, Microsoft Research+

ABSTRACT

In the emerging *agentic analytics*, the fundamental unit of computation is no longer a SQL query or a Python script but an *agentic workflow*: a DAG of agent operations, each defined by a template that may compose sub-operations such as SQL queries, object fetches, model invocations, or multi-round subagent interactions. These workflows are computationally explosive – a single input triggers cascades of agent and subagent calls that quickly overwhelm the modest, fragmented GPU resources typical of research groups; yet existing serving systems treat each call in isolation.

We present LUMILAKE, an agentic analytics engine that models workflows as first-class, introspectable DAGs and addresses three key challenges: (i) LLM-aware query optimization that flattens the two-level template structure and consolidates shared prefixes to maximize KV-cache reuse across the explosion of agent instances, (ii) a multi-tenant runtime that dispatches heterogeneous operators across mixed-generation hardware with cross-tenant deduplication, and (iii) governance by design with semantic lineage at every agent and subagent step. Experiments on six workflows from clinical decision support, materials spectroscopy, and financial analysis show up to 3.8× speedup over Ray on commodity hardware. Our engine is open-sourced at https://github.com/mlsys-io/lumilake.

1 INTRODUCTION

For decades, data analytics engines have been defined by the primitives they execute. SQL engines process declarative queries over relational tables; Spark [40] and its successors run distributed dataflow programs written in Python or Scala. The two paradigms shared storage, often with a data lakehouse [1], but were connected only through manual hand-offs or brittle ETL. In both cases, the "blood" of the engine, i.e., the unit of work it optimizes, schedules, and governs, was a query or a script authored by a human.

We argue that a paradigm shift is underway. In the emerging *agentic analytics*, the fundamental unit of computation is an *agentic workflow*: a directed acyclic graph (DAG) whose nodes are *agent operations*, each of which spawns sub-operations such as data retrieval, model invocations, or subagent interactions. At the top level, a user submits a workflow DAG that orchestrates multiple agent operations; for example, a data-retrieval stage feeding two parallel analysis agents whose outputs are synthesized by a third. At the node level, each agent operation instantiates a sub-DAG of actions defined by a *template*: issuing a SQL query, fetching objects from storage, invoking a VLM or diffusion model, launching subagents for debate, or calling external APIs, with the agent generating the runtime parameters that bind each action to the specific input. This compositional, two-level structure distinguishes agentic workflows from both flat SQL query plans and static Spark dataflow graphs: the engine must optimize not only the top-level DAG but also the templated sub-operations within each node.

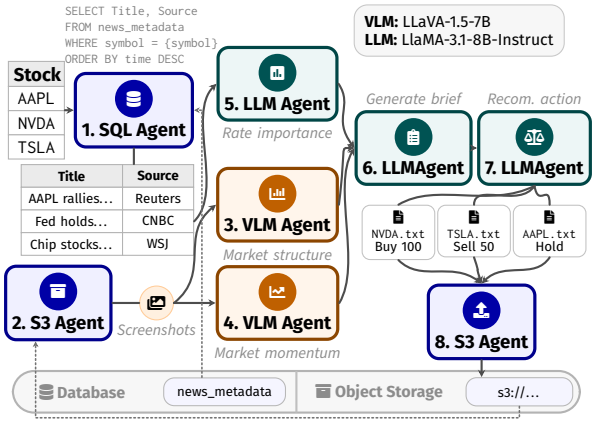


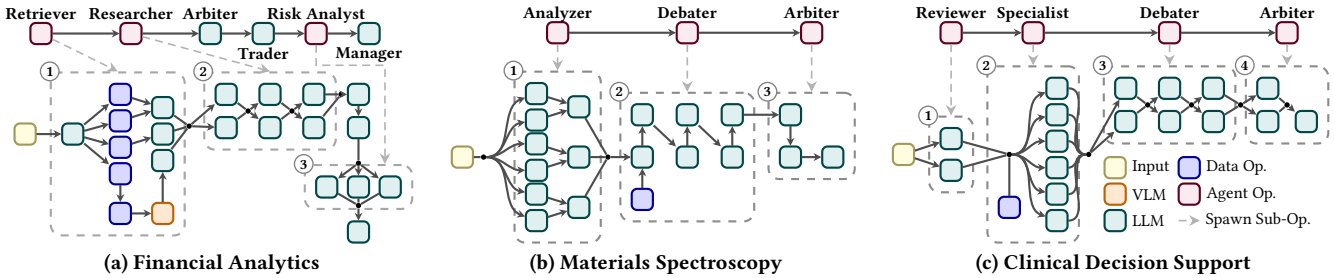
Figure 1: An agentic analytics workflow that combines SQL query, object fetch, and LLM invocations.

Just as the transition from hand-written queries to cost-based optimizers transformed relational analytics, the transition from human-authored scripts to agent-authored pipelines demands a new engine design: one that understands, optimizes, and governs agentic workflows as first-class objects.

This shift is already visible in AI4Science [16, 33] [1]. A clinical decision-support workflow chains a data-retrieval agent that issues SQL over patient records to six specialist agents, each prompted with distinct clinical guidelines, which follow a debate template of three pro/con rounds before a final arbiter synthesizes the diagnosis [3]. A materials-spectroscopy workflow dispatches three parallel expert agents, each following a template with two analyst subagents using different guided prompts, then orchestrates cross-branch critique through iterative rebuttal rounds [5]. A financial-analysis workflow retrieves structured data via SQL and unstructured snapshot images via S3, fans out to VLM subagents for chart interpretation, and funnels their outputs through a bull/bear debate among trading personas [36].

Consider the financial-analysis workflow in Figure 1: given a stock symbol, a top-level DAG of seven agent operations produces a Buy/Sell/Hold recommendation, chaining data retrieval (Nodes 1–2), parallel analysis (Nodes 3–4), and sequential reasoning (Nodes 5–7). Each agent node follows its own template: Node 1 issues a SQL query over a relational database; Node 2 fetches market snapshots from object storage; Nodes 3–4 each invoke a VLM to analyze every retrieved image; and Nodes 5–7 compose multi-step LLM calls for importance rating, synthesis, and recommendation. Running this workflow for 160 symbols instantiates the same templates 160 times, producing over 800 GPU-resident LLM/VLM calls, most of which share identical system prompts and overlapping prefixes. In

[1] Alternatively, AI4X in some cases. We use AI4Science in this paper.



(a) Financial Analytics (b) Materials Spectroscopy (c) Clinical Decision Support

Figure 2: Three representative agentic analytics workflows derived with domain experts. Edges denote data dependencies.

a Ray [18]+vLLM [12] baseline, each call is served independently with no KV-cache sharing across nodes or symbols. On a university cluster of up to tens of commodity GPUs, this already takes hours.

Challenges. The above example exposes three challenges that generalize across AI4Science applications:

(1) *Agentic workflows are computationally explosive.* An agentic workflow amplifies each input through two levels of composition: the top-level DAG fans out to multiple agent operations, and each operation's template may itself comprise subagent calls, debate rounds, and synthesis steps, e.g., nearly 20 LLM calls per patient in the clinical workflow and five GPU operators per stock symbol above. As instances follow shared templates, they naturally exhibit common system prompts and prefixes, and reusable artifacts; yet existing serving systems [12, 22, 42, 43] treat each call in isolation.

(2) *Heterogeneous operators on fragmented infrastructure.* A single DAG mixes SQL (CPU), object fetches (network), and VLM/LLM or diffusion inference (GPU), each with different resource and latency profiles. Research groups typically operate with a small number of commodity GPUs, e.g., a mix of consumer cards and occasional datacenter-grade accelerators spread across lab servers, departmental clusters, and cloud spot instances, where resources are fragmented, volatile, not abundant, and shared among competing users. Orchestration frameworks [34] rely on external LLM API calls with no cross-query batching or resource reuse; Ray [18] manages heterogeneous resources but lacks the LLM-semantic awareness to co-schedule across operator types or deduplicate across tenants.

(3) *Semantic provenance.* The two-level template structure means tracing a final output requires following both the top-level workflow and each agent's sub-operations. Unlike Spark's RDD lineage for fault tolerance, agentic analytics must record inputs, prompts, model versions, and intermediate outputs at every step for reproducibility and attribution [15, 21, 27].

Our solution. Our prior work addressed pieces of this problem: Halo [28] (DAG-level batch optimization), Helium [32] (cache-aware LLM optimization), and FlowMesh [29] (multi-tenant service fabric). Building on these, we introduce Lumilake, an analytics engine with agentic workflows as the primary compute primitive:

- *LLM-aware query optimization.* Lumilake models agentic workflows as introspectable DAGs, flattening the two-level template structure into a unified operator graph, and applies cost-based rewrites that consolidate shared prefixes, deduplicate reusable stages, and schedule execution to maximize KV-cache reuse, taming the explosion of agent instances into a coordinated plan that fits within a modest GPU budget.
- *Multi-tenant, model-aware runtime.* Workflows are decomposed into stateless, content-addressable tasks dispatched with operator-type-aware placement and cross-tenant deduplication, maximizing utilization from fragmented, mixed hardware.
- *Governance by design.* Semantic lineage (inputs, model versions, prompts, intermediate outputs) is co-designed with the operator abstraction, making provenance intrinsic to execution so that every agent and subagent step is auditable and reproducible.

We deployed Lumilake at the National University of Singapore and evaluated on six workflows spanning clinical decision support, materials spectroscopy, and multimodal financial analysis, achieving up to 3.8× speedup over Ray [18] on commodity hardware. Our results suggest that the database principles of query optimization, multi-tenancy, and data lineage provide a strong foundation for this new era of agentic analytics.

2 BACKGROUND

2.1 Agentic Analytics in AI4Science

Agentic workflows are replacing SQL queries and Python scripts as the primary analytical primitive. An agentic workflow is a DAG of agent operations, where each operation instantiates a sub-DAG of actions defined by a template, e.g., SQL queries, retrieval calls, LLM/VLM invocations, subagent interactions, composed without human intervention. Three converging trends drive this shift. First, modern LLMs serve as universal data interpreters, parsing SQL results, analyzing images via VLMs, and reasoning across modalities in one context. Second, orchestration tools [34, 41] have lowered the barrier to building multi-agent pipelines from such templates. Third, AI4Science applications inherently demand cross-modal reasoning; a clinical diagnosis needs structured lab values, unstructured reports, and imaging data; materials discovery needs property tables, spectra, and literature synthesis. We describe six workflows drawn from real scientific and industrial settings that also serve as our evaluation benchmarks in Section 4. Figure 2 illustrates three representative workflows.

Financial data analytics. The following four workflows reflect common patterns in AI4Finance analysis. (1) `Q1-Social` models a text-only analyst roundtable: given a stock symbol, agents issue four SQL retrieval queries over financial, insider-trading, market, and news data, summarize the retrieved materials, and follow a three-round discussion template among four analyst personas before a reporter synthesizes a final report. (2) `Q2-Trading`, modeled after the TradingAgents framework [36], adds multimodal analysis: in addition to structured data retrieval, a VLM agent analyzes market

snapshot images, and agents follow a multi-round debate template covering bull/bear researcher debate, risk analysis, trader decisions, and fund-manager synthesis, combining SQL, S3 retrieval, VLM, and LLM operators in a single 18-node DAG. (3) `Q3-ImageGen` captures a news-to-infographic generation scenario; given a stock symbol, the workflow retrieves related news, summarizes it, drafts competing render prompts via a critique template, invokes a diffusion model, critiques outputs with a VLM, and produces a final image artifact. (4) `Q4-Summary` represents an ETL-style ingestion pipeline commonly used by financial data vendors; it takes news images from object storage, analyzes each with a VLM, and runs three parallel drafting-and-critique branches following a reflection template before a final summarizer writes the output.

Clinical decision support. The `Q5-Healthcare` workflow is an osteoporosis decision-support pipeline developed in collaboration with experts at the National University Hospital (NUH) Singapore. Each input is a patient report containing clinical metrics drawn from the NHANES dataset [19]. The workflow begins with two micro reviews over imaging and clinical-risk evidence. The original patient report is retrieved from the lakehouse, and six specialist agents cross-reference it against the micro-review summaries from different clinical angles, each following a template with distinct clinical guidelines from practicing endocrinologists. The specialists' outputs feed into a three-round pro/con debate template, after which an arbiter produces the final diagnostic report with contradiction checking. A single patient case requires nearly 20 LLM calls.

Materials spectroscopy. The `Q6-Materials` workflow is a spectroscopy analysis pipeline developed with researchers from the NUS Department of Chemical Engineering, based on experimental data from [13]. Each input is a single-spectrum table covering either X-ray diffraction (XRD) or photoluminescence (PL) measurements; the base corpus contains 38 such spectra. The workflow launches three expert branches (feature identification, quantitative extraction, physical interpretation), each following a template with two analyst subagents using different guided prompts. A three-round cross-branch critique-and-rebuttal debate template challenges the branch analyses, with domain guidance and user insights retrieved from the lakehouse to steer the discussion. A three-node chain summarizes the debate and synthesizes the final report.

2.2 Prior Systems for Agentic Workflows

Table 1 summarizes the capabilities of existing systems along four axes that correspond to the challenges identified in Section 1: cross-query LLM-aware optimization (C1), KV-cache-aware scheduling (C2), semantic provenance (C3), and first-class support for agentic DAGs with templated sub-operations (C4). No prior system addresses all four. We discuss each category below.

LLM inference engines. vLLM [12] and SGLang [42] achieve high single-model throughput via continuous batching, paged attention, and prefix caching. Disaggregated designs such as Splitwise [22] and DistServe [43] further decouple prefill and decode phases for goodput gains. However, all operate at request granularity: they do not detect shared prefixes *across* workflow instances, reorder execution across operators, or bypass already-materialized subtrees. Because agentic workflows instantiate the same agent templates many times across inputs, the structural sharing from repeated templates (Challenge 1) is entirely invisible to these systems.

Table 1: Comparison of systems w.r.t. cross-query LLM-aware optimization (C1), KV-cache-aware scheduling (C2), semantic provenance (C3), and first-class support for agentic DAGs (C4). ✓ = supported, ○ = partial, × = not supported.

Category	System(s)	C1	C2	C3	C4
LLM Inference	vLLM [12], SGLang [42]	×	○	×	×
Orchestration	AutoGen [34], LangGraph [10]	×	×	×	○
Classic Data Sys.	Ray [18], Spark [40]	○	×	○	○
LLM Data Sys.	Palimpzest [15], LOTUS [21]	×	×	×	○
Workflow Sys.	AFlow [41], JellyBean [35]	×	×	×	✓
Ours	Lumilake	✓	✓	✓	✓

Agent orchestration frameworks. AutoGen [34], LangGraph [10], and CrewAI [9] provide expressive multi-agent abstractions, including support for composing agent operations into DAGs, but treat every model invocation as a black-box user-defined function (UDF) dispatched to an external serving endpoint. They perform no cross-query optimization (C1), have no visibility into KV-cache state (C2), and provide no built-in provenance beyond conversation logs. Multi-tenancy, i.e., batching and deduplicating across concurrent users, must be retrofitted externally. Critically, while these frameworks *define* templated agent patterns (e.g., debate rounds, reflection loops), they do not *optimize across* template instances.

Classic data systems. Ray [18] and Kubernetes-based orchestrators provide resource management and fault tolerance for heterogeneous clusters, making them the de facto baseline for deploying ML workloads on the fragmented, commodity hardware typical of research groups (Challenge 2). However, they lack semantic awareness of LLM workloads: they do not consolidate equivalent agent operations across users or track semantic provenance across the two-level template hierarchy.

LLM-integrated (semantic) data systems. Palimpzest [15] explores model selection and operator reordering across cost-quality-latency trade-offs. LOTUS [21] formalizes semantic operators with accuracy guarantees. DocETL [27] uses agent-based rewrites for document pipelines. Abacus [25] adds Cascades-style cost-based optimization to Palimpzest. These systems advance declarative optimization for tabular or document-level operations, but target flat operator pipelines rather than the compositional, two-level DAGs of agentic workflows; they do not address multi-step reasoning chains with templated subagent interactions, multimodal operators, or stateful KV dependencies spanning operator boundaries.

Workflow systems. AFlow [41] uses MCTS to discover operator compositions; JellyBean [35] optimizes ML workflows over a hybrid cloud via cost-based query optimization. These systems improve the *logical* plan by choosing which ML models or agent templates to compose and how, but delegate execution to black-box backends with no cross-query batching, KV-cache coordination, or built-in provenance. Lumilake is complementary; it takes a given workflow DAG and optimizes its *physical* execution, exploiting the template structure for prefix consolidation and cross-instance deduplication.

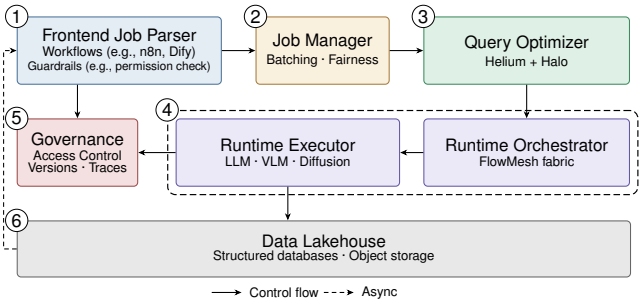


Figure 3: LUMILAKE system architecture.

Our prior work on optimizing agentic workflows. Helium [32] models an agentic workflow as a query-plan DAG and optimizes LLM serving via logical rewrites, proactive caching, and cache-aware scheduling; its greedy DFS scales linearly but assumes all nodes share one base LLM. Halo [28] removes this constraint via epoch-based dynamic programming over heterogeneous CPU/GPU operators. These systems pair one workflow template with one user's input batch. In practice, *multiple users submit different templates concurrently* to a shared cluster; this is the norm at universities where researchers across domains share the same GPUs. FlowMesh [29] proposes a multi-tenant service fabric that opportunistically deduplicates tasks across concurrent users. LUMILAKE extends FlowMesh to a multi-tenant, multi-template setting. The opportunity is substantial even when templates differ: in our deployment, workflows from different groups still share base LLMs and large system-prompt fragments, creating cross-job prefix and weight reuse that single-tenant optimizers miss. Our experiments (Section 4.4) confirm benefits from cross-tenant deduplication alone.

3 SYSTEM DESIGN

LUMILAKE is an agentic data analytics engine built upon a data lakehouse [1], combining structured databases and object storage. Each submitted job consists of a workflow defined in N8N JSON [31], an input specified as either a literal string list or a location in the lakehouse, and an output location in the lakehouse. As shown in Figure 3, LUMILAKE is modular and consists of six components:

① Workflow Parser: parses incoming jobs, checks against guardrails, and sends validated jobs to the job manager.
② Job Manager: manages job queues, selects pending jobs to form a batch to dispatch, and assigns them to specific workers.
③ Query Optimizer: optimizes the workflow DAG of the batch, and generates a plan that maximizes resource reuse.
④ Runtime Processor (Orchestrator + Executor): executes the optimized workflow following the generated schedule on the given hardware, while storing metadata for governance.
⑤ Governance Layer: stores metadata for data and execution traces, used for access control and auditing.
⑥ Storage: contains all structured and unstructured data, including governance metadata.

Running example. Throughout this section, we use a simplified variant of the `Q2-Trading` workflow (Section 2.1), shown in Figure 4. The seven-node DAG takes a stock symbol via placeholder `{symbol}` and produces a Buy/Sell/Hold recommendation: Node 1 issues a SQL query over `news_metadata`; Node 2 fetches snapshot images from S3; Nodes 3–4 are parallel VLM agents for market-structure and momentum analysis; Nodes 5–7 are sequential LLM agents for importance rating, synthesis, and recommendation. A user submits three inputs (`AAPL`, `NVDA`, `TSLA`) with output at s3.

In a multi-tenant scenario, a second user may concurrently submit a different workflow over overlapping symbols; LUMILAKE batches and deduplicates shared retrieval and prefix computations (Section 4.4). As shown in Figure 4, workflow B (`Q1-Social`) shares the same S3 and VLM prefix as workflow A, so LUMILAKE reuses the intermediate results and only executes the divergent LLM nodes.

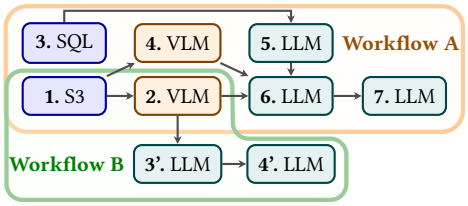


Figure 4: Running example for financial analysis workflow.

Scope and Limitations. LUMILAKE targets *template-based* agentic workflows whose top-level DAG and per-node sub-DAGs are fully specified before execution, enabling the optimizer to inspect, rewrite, and schedule the entire operator graph ahead of time. This covers the dominant pattern in AI4Science, where workflows are authored once by domain experts and instantiated over many inputs (patient records, spectra, stock symbols, etc.). LUMILAKE does not support dynamically generated DAGs in which agents decide at runtime which sub-operations to spawn (e.g., adaptive tool-use agents); this would require online re-optimization, left to future work. All data is assumed to reside in or be accessible through the lakehouse; external API calls are modeled as opaque CPU operators with profiled latency but are excluded for prefix optimization.

3.1 Workflow Parser

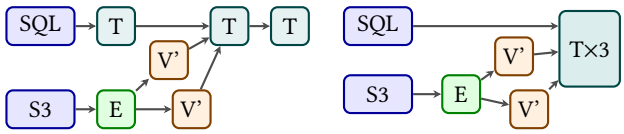
The frontend exposes a low-code interface for defining and submitting workloads. A workload consists of three parts: (1) a *workflow template* defined via our N8N-based visual editor or as a JSON file, (2) an *input* that binds each placeholder in the template to a lakehouse path or a literal string list, and (3) an *output location* in the lakehouse. Figure 1 shows an example: the template contains seven agent operations with curly-brace placeholders (e.g., `{symbol}`); the input binds `symbol` to a list of stock tickers read from `s3://.../stock_list.txt`; the output is written to `s3://.../financial-suggestion/`. Users may also assign a priority to each workload, which influences scheduling order.

Upon receiving a job, the parser aggregates all data-access nodes in the workflow and checks that the submitting user has the required permissions via the governance layer (Section 3.5).

3.2 Job Manager

The job manager maintains a pool of queued jobs and a pool of CPU and GPU workers, and controls batch construction and dispatch.

Input slicing. Workers are partitioned into groups at setup time, each group serving as the unit for job dispatch. Upon ingestion, a



(a) After optimization. (b) After contraction.

Figure 5: Intermediate graphs by the query optimizer for our running example. E, V', and T represent the embedding, VLM reasoning, and LLM text generation nodes, respectively.

job's input is split along the input dimension, e.g., 160 stock symbols are divided into slices of 1 each, while the workflow DAG remains intact for every slice. Each slice becomes an independent scheduling unit. This design has two benefits: (i) slices have roughly uniform cost, preventing head-of-line blocking from a single large job; and (ii) lightweight jobs consume fewer workers, enabling elasticity.

Priority queues. Pending job slices are maintained in per-priority, per-user queues. Each scheduling cycle draws a configured quantum from each priority level (higher priorities receive larger quanta) to form a candidate pool. Within each level, round-robin over user subqueues prevents any single user from monopolizing a priority lane. To avoid starvation, each job carries a miss counter that increments whenever it is considered but not selected; once the counter exceeds a threshold N_s, the job is pinned into the next batch.

Clustering-based batch selection. From the candidate pool, the manager selects a batch of size B that maximizes intra-batch affinity—promoting downstream prefix deduplication and reducing the number of distinct model weights that must be co-resident. For workflows i, j, affinity is measured by $D(i,j) = 1 - (\alpha\, S_{\text{model}} + \beta\, S_{\text{data}} + \gamma\, S_{\text{size}})$ with $\alpha+\beta+\gamma = 1$, where $S_{\text{model}} = |M_i \cap M_j|/|M_i \cup M_j|$ is the Jaccard similarity over model-identifier sets, $S_{\text{data}} = |T_i \cap T_j|/|T_i \cup T_j|$ over normalized system-prompt sets, and $S_{\text{size}} = 1 - |N_i - N_j|/\max(N_i, N_j)$ penalizes mismatched graph sizes. We set $\alpha = 0.7$, $\beta = 0.2$, $\gamma = 0.1$ to prioritize model reuse.

Given these distances, LUMILAKE runs agglomerative hierarchical clustering (average linkage) over the candidates. After each merge, a cluster is eligible only if it contains all starved jobs and has size $\geq B$. For eligible clusters larger than B, we greedily select a B-subset by seeding with the starved set and iteratively adding the nearest workflow. The globally best subset, minimizing average intra-batch distance, is emitted. In the running example, with batch size 2, the first two input slices (`AAPL` and `NVDA`) are selected.

3.3 Query Optimizer

The selected batch is sent to the query optimizer, which (1) rewrites the batch of workflow graphs into a single optimized graph and (2) generates an execution schedule maximizing resource reuse.

Graph Rewriting. The optimizer first aggregates slices from the same job (they share identical dataflow and benefit from batched execution), then applies three semantics-preserving rewrites to the merged DAG:

Rule 1: VLM splitting. Each VLM node is decomposed into an image-embedding node and a VLM-reasoning node with the embedding as input. This exposes the embedding computation for deduplication: when multiple agents analyze the same image, the embedding nodes are identical and can be deduplicated by Rule 3.

Rule 2: Node absorption. Lightweight formatting and code-execution nodes that perform no GPU computation are absorbed into their immediate successors by prepending the formatting logic to the successor's input specification. This reduces the node count and simplifies the DAG without affecting semantics.

Rule 3: Subgraph deduplication. We compute a *structural signature* for each node by normalizing its operator specification (backend, model, data template, inference parameters) and recursively including the signatures of its parents. Nodes with identical signatures within and across jobs are merged; their downstream dependents are rewired to the single canonical copy.

In the running example, Rule 1 splits each VLM node into an embedding and a reasoning node; Rule 3 then merges the two identical embedding nodes into one shared copy, producing the optimized graph in Figure 5a.

Scheduling Objective is defined as follows.

Definition 1 (Agentic Workflow DAG)**.** An *agentic workflow DAG* is a directed acyclic graph $G = (V, E)$ where each node $v \in V$ is annotated with an operator type

$$\tau(v) \in \{\textsc{Text},\ \textsc{Embed},\ \text{VLM},\ \textsc{Diff},\ \text{DB},\ \text{S3}\}$$

and an optional model identifier $m(v)$. The vertex set is partitioned into GPU operators $V_{\text{GPU}} = \{v \mid \tau(v) \in \{\textsc{Text}, \textsc{Embed}, \text{VLM}, \textsc{Diff}\}\}$ and CPU operators $V_{\text{CPU}} = V \setminus V_{\text{GPU}}$. An edge $(u, v) \in E$ denotes a data dependency: v cannot execute until u completes.

Definition 2 (Worker Pool)**.** A *worker pool* is a set $W = W_{\text{GPU}} \cup W_{\text{CPU}}$. Each GPU worker $w \in W_{\text{GPU}}$ executes GPU operators; each CPU worker $w \in W_{\text{CPU}}$ executes CPU operators. A node v must be assigned to a worker of the matching type: $v \in V_{\text{GPU}} \Leftrightarrow \pi(v) \in W_{\text{GPU}}$ and likewise for CPU. Each GPU worker maintains mutable state (m_w, v_w)—the currently loaded model and the last executed node—which determines model-switching and KV-cache reuse costs.

Problem 1 (Heterogeneous DAG Scheduling (HDS))**.** Given a workflow DAG $G = (V, E)$ and a worker pool W, find a worker assignment $\pi : V \to W$ respecting operator types and an execution ordering that respects all data dependencies, minimizing a weighted combination of GPU makespan, total GPU load, and CPU load.

THEOREM 1 (NP-HARDNESS). *The Heterogeneous DAG Scheduling problem (Problem 1) is NP-hard.*

PROOF. To show that HDS is NP-hard in general, we only need to prove that a special case—a single operator type, a single model, two identical GPU workers, and independent nodes (no edges)—is NP-hard, for which we reduce from MULTIPROCESSOR SCHEDULING ($m \geq 2$ machines) [4]. Given n jobs with processing times $p_1, \ldots, p_n$ and two identical machines, construct an HDS instance with $G = (V, \varnothing)$ of n independent TEXT nodes (same model, cost p_i) and $|W_{\text{GPU}}| = 2$. With no CPU nodes and only two identical GPU workers, the scheduling objective reduces to minimizing the maximum load across the two workers, i.e., $\max(L_1, L_2)$—exactly the two-machine makespan in MULTIPROCESSOR SCHEDULING. □

Since HDS is intractable, we introduce two reasonable assumptions that reduce the search space. The first assumption follows

from a common practice also adopted in Halo [28], which discretizes the search space into epochs.

Assumption 1 (Epoch-based scheduling)**.** Rather than optimizing over continuous start times, we partition execution into discrete *epochs*: a partition of V into ordered groups $\sigma = (E_1, \ldots, E_T)$ such that for every edge $(u, v) \in E$, u is not assigned to a later epoch than v. Within each epoch, all assigned nodes execute concurrently on their respective workers.

With this assumption, the search space is parameterized by a 3-tuple (σ, π, ρ): the epoch partition σ, the worker assignment π, and the per-worker execution order ρ within each epoch (which determines KV-cache reuse).

The second assumption constrains how contracted components are scheduled:

Assumption 2 (Atomic component execution)**.** Each contracted homogeneous component executes in a dedicated epoch: its nodes are distributed across all available GPU workers, and no other GPU nodes share that epoch.

It is reasonable because grouping nodes that share the same base LLM maximizes the chance of resource reuse across nodes.

Cost Model. Under Assumption 1, the total schedule cost is:

$$C(\sigma, \pi, \rho) = \sum_{t=1}^{T} \Big[\underbrace{P(t)}_{\text{epoch penalty}} + \underbrace{w_{\max} \cdot \max_{w \in W_{\text{GPU}}} L_w(E_t)}_{\text{GPU makespan}} + \underbrace{w_{\Sigma} \cdot \sum_{w \in W_{\text{GPU}}} L_w(E_t)}_{\text{GPU total load}} + \underbrace{w_{\text{CPU}} \cdot \sum_{w \in W_{\text{CPU}}} L_w(E_t)}_{\text{CPU load}} \Big], \tag{1}$$

where $L_w(E_t)$ is the execution load on worker w in epoch t (defined below), $P(t) = 1 + 0.1 \cdot t$ is an increasing epoch penalty, and $w_{\max}$, w_{Σ}, w_{CPU} balance GPU makespan, total GPU load, and CPU overlap (weight choices in Section A).

The per-worker load $L_w(E_t)$ aggregates operator execution cost, model initialization overhead, and a KV-cache reuse bonus:

$$L_w(E_t) = \sum_{v \in E_t :\, \pi(v) = w} \big[\mu(v, w) \cdot C_{\tau(v)}(v) + C_{\text{init}}(v, w) \big]. \tag{2}$$

$C_{\tau(v)}(v)$ is the estimated execution time of node v, linear in model size $|m(v)|$ and input count b (the number of inputs routed to node v): $C_{\tau(v)}(v) = \alpha_\tau \cdot |m(v)| + \beta_\tau \cdot b$, with per-class coefficients for text, embedding, VLM, and diffusion.

$C_{\text{init}}(v, w)$ is the overhead of loading model weights onto worker w:

$$C_{\text{init}}(v, w) = \begin{cases} \frac{1}{2}\lambda \cdot |m(v)|, & \text{cold start (no prior model)}, \\ 0, & m_w = m(v) \text{ (reusing resident model)}, \\ \lambda \cdot |m(v)|, & \text{full model switch}, \end{cases} \tag{3}$$

where λ is the per-billion-parameter load coefficient. Cold starts pay half because no eviction is needed.

$\mu(v, w)$ is the KV-cache reuse bonus: 0.9 when the last node on worker w is a parent of v (the child reuses the parent's cached key-value state), and 1.0 otherwise.

Algorithm 1 Hybrid Helium + Halo Scheduling

Require: Rewritten DAG $G = (V, E)$, worker pool W
Ensure: Schedule (σ, π, ρ)

Phase 1: Helium scheduling with cycle-safe contraction
$C \leftarrow$ same-model, text-only connected components of G
Remove components below size/path threshold
Split any $C \in C$ that would create a cycle upon contraction
for each surviving component C **do**
  $order_C \leftarrow \textsc{Helium}(C)$ ▷ [32]
  Contract C into super-node $\hat{v}_C$; remap external edges
end for
Build contracted graph $\hat{G}$

Phase 2: Halo DP on contracted graph
$(\hat{\sigma}, \hat{\pi}) \leftarrow \textsc{Halo}(\hat{G}, W)$ ▷ [28]

Phase 3: Schedule expansion
for each super-node $\hat{v}_C$ in $\hat{\sigma}$ **do**
  Expand $\hat{v}_C$ into $order_C$; shard across GPU workers
end for
return merged schedule (σ, π, ρ)

For CPU workers, C_{init} and μ do not apply (no model loading, no KV-cache), so L_w reduces to $\sum C_{\tau(v)}(v)$. For DB nodes, $C_{\text{DB}}(v) = \gamma \cdot c_{\text{explain}}(v)$, where $c_{\text{explain}}(v)$ is the base cost from PostgreSQL `EXPLAIN` [28] and $\gamma \in [0.5, 1.0]$ discounts for data reuse when queries in the same epoch access overlapping tables. Each query carries a data-access footprint mapping table names to access weights; γ decreases with the weighted overlap ratio between footprints. For S3 nodes, $C_{\text{S3}}(v) = c_f \cdot n_f + c_t \cdot s_t$, where n_f is the number of objects fetched and s_t the total transfer size. Concrete coefficient values appear in Section A.

Hybrid Scheduling Algorithm. The rewritten graph may contain 30–50 nodes with heterogeneous operator types. Helium [32] scales linearly via greedy DFS over a templated radix tree but assumes all nodes share one base LLM. Halo [28] co-schedules heterogeneous CPU/GPU operators via dynamic programming but is exponential in graph size and struggles beyond ~20 nodes. Our key observation is that multimodal agentic workflows still contain large homogeneous subgraphs (all text-generation nodes sharing one LLM), with heterogeneous operators appearing mainly at DAG boundaries. Based on this observation, we propose a hybrid scheduling algorithm that combines Helium and Halo to get the best of both worlds. The algorithm proceeds in three phases and is summarized in Algorithm 1.

Phase 1: Helium scheduling with cycle-safe contraction. We identify connected components of same-model, text-only LLM nodes; components below a size threshold are left uncontracted. Before contracting, we check cycle safety: contracting a region is valid only if no external node has both an incoming and outgoing dependency on it (see Figure 6b for an example of a cycle that would arise otherwise). Any violating component is split at the ancestor boundary and re-partitioned. In this way, we can avoid the cycle issue, as formally proved in Theorem 2.

THEOREM 2 (CONTRACTION SAFETY). *The cycle-avoidance procedure guarantees that the contracted graph $\hat{G}$ is a valid DAG.*

PROOF. Suppose for contradiction that $\hat{G}$ contains a cycle. Since G is acyclic, the cycle must pass through at least one super-node $\hat{v}_C$, so there exist external nodes $u, w \notin C$ with $u \to \hat{v}_C \to w$ and a path from w to u in $\hat{G}$. Expanding $\hat{v}_C$, the edge $u \to \hat{v}_C$ becomes $u \to v'$ for some $v' \in C$, and $\hat{v}_C \to w$ becomes $w' \to w$ for some $w' \in C$. Combined with the path from w to u, the external node u has both an ancestor $w' \in C$ (via $w' \to w \to \cdots \to u$) and a descendant $v' \in C$ (via $u \to v'$). This is exactly the condition detected and split above. Contradiction. □

We then apply Helium [32] to each surviving component to obtain a cache-aware execution order. Each component is contracted into a super-node $\hat{v}_C$ whose cost encapsulates the Helium-scheduled internal execution:

$$C_{\text{super}}(\hat{v}_C) = \sum_{i=1}^{k} \mu_i \cdot C_{\text{TEXT}}(n_i) + M \cdot C_{\text{init}}(n_1, w), \tag{4}$$

where $M = |W_{\text{GPU}}|$, $\mu_i = 0.9$ when the preceding node in the Helium order is a parent of n_i (reflecting KV-cache reuse) and 1.0 otherwise, and $C_{\text{init}}(n_1, w)$ is the per-worker model initialization cost (identical across workers since all load model $m(C)$; n_1 triggers the load). Dependencies are remapped so that edges to members of C now target $\hat{v}_C$.

Phase 2: Halo DP on contracted graph. The contracted graph typically has 5–15 nodes, small enough for Halo's exact DP solver [28]. Halo produces an optimal epoch-based schedule and worker assignment for this graph.

Phase 3: Schedule expansion. Each super-node in the DP schedule is replaced by its Helium-ordered members, sharded across eligible GPU workers. Cross-node dependencies are preserved in the merged schedule.

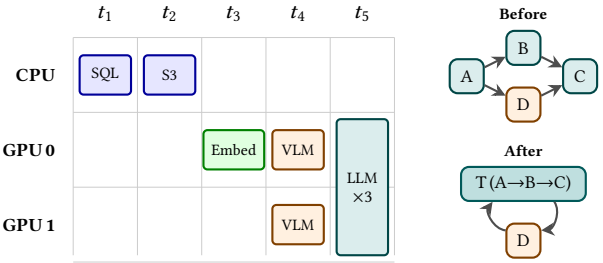
In the running example, the three sequential LLM nodes are contracted into one super-node; the retrieval, embedding, and VLM nodes remain explicit (Figure 5b), yielding the schedule in Figure 6a.

Optimality Analysis. The hybrid optimizer provides per-level optimality guarantees. Within each contracted region, Helium's DFS over the templated radix tree maximizes shared prefix length between consecutive nodes, achieving optimal cache reuse for tree-structured workflows [32]. At the inter-region level, Halo's DP exhaustively explores frontier subsets and worker placements, producing an optimal schedule for the contracted graph under its cost model [28]. By partitioning the problem at homogeneous-region boundaries, the hybrid approach extends the reach of both solvers: Helium operates within its optimal regime on arbitrarily large single-model subgraphs, while Halo handles the remaining heterogeneous structure at a scale its DP can solve exactly, enabling end-to-end optimization of 50+ node DAGs with <2% overhead (§4.5), well beyond either optimizer's standalone capacity. The formal proof of the optimality of the schedule generated by our hybrid optimizer under Assumptions 1 and 2 can be found in Section A.2.

Complexity Analysis. Let the original graph be (V, E) with $N_m = |\{m(v)|v \in V\}|$ distinct models and $N_t = |\{v \in V|\tau(v) = \text{TEXT}\}|$ text generation nodes. Similarly, let the contracted DAG be $(\hat{V}, \hat{E})$ with $\hat{N}_m = |\{m(v)|v \in \hat{V}\}|$ distinct models. Let $M = |W_{\text{GPU}}|$ be the number of GPU workers available. Phase 1 builds the templated radix tree in $O(|V| + |E| + N_t \cdot L)$ where L is the maximum prompt length, then runs Helium's cache-aware scheduling in $O(|V_{\text{int}}| \cdot c_{\max}^3 + |E'| \cdot d)$ where $|V_{\text{int}}|$ is the number of internal TRT nodes, $c_{\max}$ the maximum branching factor, $|E'|$ the dependency edges, and d the TRT depth (Theorem B.1 in [32]). Phase 2 runs Halo's DP [28] in $O(2^{|\hat{V}|} \cdot (\hat{N}_m\,|\hat{V}|)^M)$. Phase 3 expands in $O(|V|)$. The overall complexity is therefore

$$O\big(|V| + |E| + N_t \cdot L + |V_{\text{int}}| \cdot c_{\max}^3 + |E'| \cdot d \;+\; 2^{|\hat{V}|} \cdot (\hat{N}_m\,|\hat{V}|)^M\big).$$

For a fully homogeneous workflow ($|\hat{V}| = 1$, $\hat{N}_m = 1$), the Halo term vanishes and the cost is $O(|V| + |E| + N_t \cdot L + |V_{\text{int}}| \cdot c_{\max}^3 + |E'| \cdot d)$, identical to Helium's standalone complexity. For a fully heterogeneous workflow ($|\hat{V}| = |V|$, $\hat{N}_m = N_m$), no contraction occurs ($|V_{\text{int}}| = |E'| = 0$) and it recovers Halo's $O(2^{|V|} \cdot (N_m|V|)^M)$.



(a) Schedule of the running example. **(b) Contraction cycle.**

Figure 6: Left: The schedule of the contracted example graph on a worker group with one CPU and two GPU workers. Right: A, B, C are LLM nodes (teal) sharing the same base LLM; D is a VLM node (orange). Contracting A, B, C into a super-node T creates a cycle $T \to D \to T$.

3.4 Runtime Processor

The runtime comprises an orchestrator and an executor and is built on FlowMesh [29], extending scheduling hints and additional operator support.

Orchestrator. LUMILAKE passes the optimizer's output to FlowMesh as two hints: a *hard* task-to-worker assignment (pinning each task to a specific worker) and a *soft* per-worker execution order (used as a tiebreaker among ready tasks). The soft constraint allows FlowMesh to merge tasks destined for different execution slots when doing so increases the inference batch size, improving per-node throughput.

Executor. We extend FlowMesh's executor pool beyond LLM inference to include: SQL and S3 data retrieval from the lakehouse, image embedding and VLM understanding, diffusion-based image generation, and data profiling (collecting cost estimates for the optimizer). Splitting image embedding from VLM reasoning enables deduplication when multiple agents analyze the same image.

3.5 Governance Layer

Agentic workflows compose many autonomous steps, each with its own model, prompt, and intermediate output. Auditing a final result, therefore, requires end-to-end semantic provenance across

all agent and subagent operations—a requirement absent from traditional data systems, where lineage serves fault tolerance rather than attribution. LUMILAKE's governance layer addresses this by recording provenance as a by-product of execution rather than as a separate post-hoc system. The layer is backed by an in-memory DuckDB [24] instance (periodically checkpointed to the lakehouse) and provides two services:

Access control and versioning. Role-based access control maps users to roles and roles to per-table permission IDs via a bridge table; the workflow parser checks these permissions before admitting a job (Section 3.1). Row-level versioning is maintained via a version column on every structured table; unstructured objects in MinIO are versioned by the storage service itself.

Execution traces. Every executor reads upstream results from the governance layer, records timestamped events during execution (model initialization, generation start/end, intermediate outputs), and writes final results back. These traces enable two kinds of auditing: (i) *semantic*: reconstructing which inputs, prompts, model versions, and intermediate outputs contributed to a given result; and (ii) *performance*: extracting critical-path latency, per-stage processing time, and networking overhead. In the running example, the user can trace exactly which news articles were retrieved, how each VLM agent interpreted them, and which LLM version produced the final recommendation, or diagnose a latency bottleneck.

3.6 Storage

LUMILAKE stores all data in a lakehouse combining a relational database (PostgreSQL or TimescaleDB) and an object storage (MinIO or S3), partitioned into public read-only data (subject to per-user permissions) and private per-user storage with read-write access. Governance metadata (permission tables, execution traces) is hidden from all users. During execution, LUMILAKE polls task status to synchronize fine-grained progress. Completed results are fetched from FlowMesh, traces are collected from the governance layer, and outputs are written to the user-specified lakehouse location for downstream analytics.

4 EVALUATION

We investigate the following aspects to justify LUMILAKE's efficacy:

- RQ1. LUMILAKE outperforms off-the-shelf solutions across different agentic analytics workloads and hardware generations.
- RQ2. LUMILAKE effectively scales across workflow and data sizes.
- RQ3. LUMILAKE schedules multi-tenant workloads according to their priority while exploiting overlapping opportunities.
- RQ4. LUMILAKE's query optimization and governance layer introduce low overhead.
- RQ5. The governance layer provides actionable execution traces for performance diagnosis.

Experiment setup. LUMILAKE is deployed at the National University of Singapore in collaboration with domain experts in finance, clinical medicine, and materials science.

- *Resources.* A setup representative of a university research group: four local NVIDIA H200 GPUs, with up to eight on-demand NVIDIA RTX 5090 GPUs from a cloud marketplace.

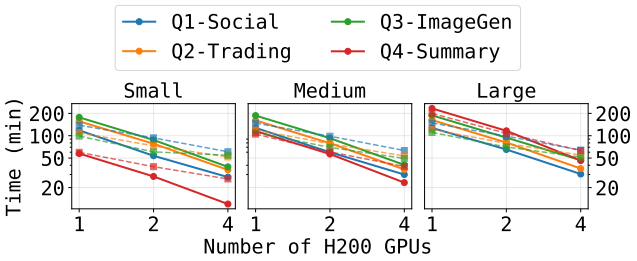


Figure 7: End-to-end time for AI4Finance workloads across data scales and GPU counts. Solid: LUMILAKE; dashed: Ray.

- *Workloads.* The six workflows Q1-Q6 described in Section 2.1: four AI4Finance workflows (text-only analysis, multimodal analysis, image generation, ETL) and two AI4Science workflows (clinical decision support, materials spectroscopy).
- *Datasets.* For AI4Finance workloads, we prepare three data splits of 1 GB (small), 10 GB (medium), and 100 GB (large); structured tables occupy one-third of each split, with the remainder consisting of raw HTML files and market snapshot images. Data size primarily affects SQL query performance. For healthcare, we collect reports for 320 patients from the NHANES 2017–2020 Pre-pandemic Data Files [19]. For materials science, we use up to 160 variant spectra from [13].
- *Baseline.* We compare with Ray [18], a widely used distributed ML framework. Each workflow is mapped to a Ray Data pipeline: LLM/VLM operators run as Ray Actor pools with one GPU per actor and vLLM [12] as the inference backend; CPU operators use standard `map`/`map_batches` transforms. A stage planner topologically sorts the DAG and schedules GPU-intensive stages within a configurable GPU budget.
- *Metrics.* End-to-end wall-clock time (lower is better) for all experiments; job progress over time for multi-tenant case studies; percentage overhead for optimization and governance.

4.1 End-to-end Performance

AI4Finance workloads Q1–4 (Figure 7). For the ETL workflow we sample 800, 1600, and 3200 news HTML files (batches of 100); the other three workflows process 160 stock symbols (batches of 10). We vary the number of H200 GPUs from one to four. With a single GPU, Ray is competitive on some workloads because continuous batching via vLLM streams all inputs concurrently, yielding a larger effective batch size and higher per-operator GPU utilization. LUMILAKE partitions inputs into smaller batches for cross-node scheduling, which reduces per-operator batch size on one GPU. However, as GPUs increase, LUMILAKE's schedule-driven batching enables better reuse of model weights and KV-cache prefixes across nodes within each worker; with four GPUs, LUMILAKE outperforms Ray on all four workloads, averaging 1.7× speedup (up to 2.2×).

AI4Science workloads Q5–6 (Figure 8). We run the healthcare workflow over 80, 160, and 320 patients, and the materials workflow over 40, 80, and 160 spectra (batches of 10 in both cases). LUMILAKE scales linearly with GPU count for both workloads, and the advantage over Ray grows with available compute, averaging 2.2× speedup (up to 3.8×) on four H200s. With four H200s, LUMILAKE

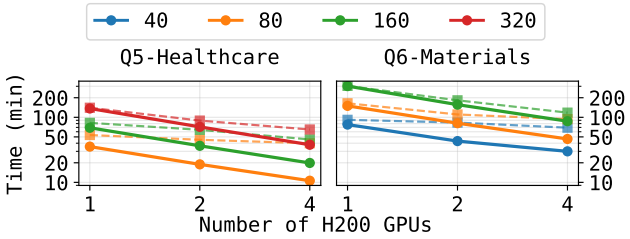


Figure 8: End-to-end time for AI4Science workloads across data scales and H200 GPU counts.

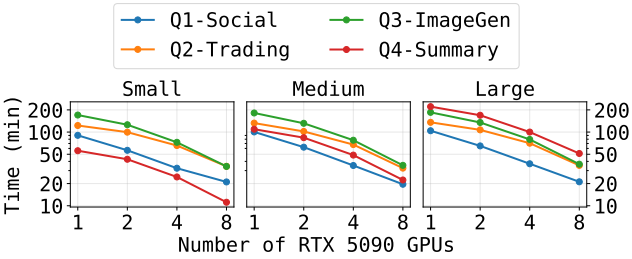


Figure 9: End-to-end time for AI4Finance workloads on RTX 5090 GPUs.

Table 2: Optimizer ablation on workflows of various sizes.

	10-node	18-node	50-node
# Runtime Nodes	16	24	80
End-to-end Time (Halo)	49.4 min	–	–
End-to-end Time (Ours)	62.9 min	79.2 min	168.6 min

diagnoses over 300 patients or analyzes 100 spectra within one hour—demonstrating that complex, high-quality agentic analyses are practical on modest hardware.

Alternative hardware (Figure 9). We repeat the financial workload experiments on up to eight NVIDIA RTX 5090 GPUs from a cloud marketplace. The scaling pattern mirrors the H200 results. Variance is higher because different GPU counts are provisioned on different cloud instances, which differ slightly in CPU performance, network speed, and GPU throughput. We don't run the Ray baseline on RTX 5090s due to budget constraints.

4.2 Optimizer Ablation

To demonstrate the necessity of graph contraction for scaling Halo to large workflows, we compare our hybrid optimizer against Halo on variants of `Q2-Trading` with different sizes (Table 2). Although Halo produces a better schedule on the small 10-node graph, the DP solver fails to exhaust the search space within the 30-minute timeout on larger variants which contain more than 20 runtime nodes. In contrast, our hybrid optimizer keeps the total optimizer overhead at ~1.4 s per batch across all variants by contracting homogeneous regions and applying Halo to the contracted graph, trading a modest optimality gap for better scalability.

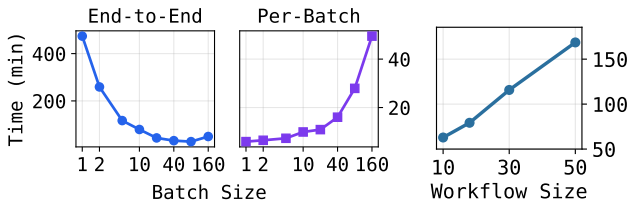


Figure 10: Left and Mid: End-to-end time and per-batch execution time vs. batch sizes on `Q2-Trading` (2×H200, medium). Right: End-to-end time vs. workflow sizes (GPU nodes).

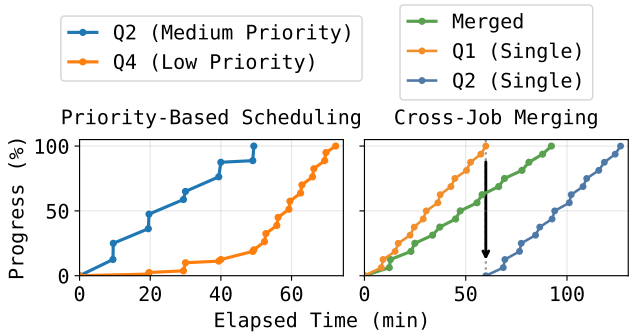


Figure 11: Multi-tenant progress on 2xH200. Left: priority scheduling (medium vs. low). Right: cross-job merging.

4.3 Scalability and Sensitivity Analysis

Batch size (Figure 10 Left and Mid). We vary the batch size from 1 to 160 on `Q2-Trading` with 160 inputs, two H200s, and the medium data split. Smaller batches yield lower per-batch latency but higher end-to-end time because cross-input reuse is limited. As batch size grows, the optimizer can consolidate shared prefixes and deduplicate retrieval across more inputs per batch, sharply reducing end-to-end time: batch size 10 takes 79.2 min (6.0× faster than batch size 1), and batch size 40 takes 32.0 min (14.8×). However, per-batch execution time rises from 5.9 min to 16.0 min over this range as each batch contains more work. Note that with two H200s, LUMILAKE can process two batches concurrently. At a batch size of 160, all inputs are processed in a single batch, leaving one GPU idle and yielding a worse end-to-end time than a batch size of 80.

Workflow size (Figure 10 Right). We construct variants of the `Q2-Trading` workflow with 10, 18 (original), 30, and 50 LLM nodes by varying the number of debate rounds, and run each on two H200s with the medium data split. LUMILAKE scales linearly with node count, confirming that the hybrid optimizer remains efficient as workflows grow.

4.4 Multi-tenant Case Studies

Priority scheduling and starvation avoidance (Figure 11, left). Two users submit a medium-priority `Q2-Trading` job and a low-priority `Q4-Summary` job concurrently. Both jobs make progress from the start, but the medium-priority job advances significantly

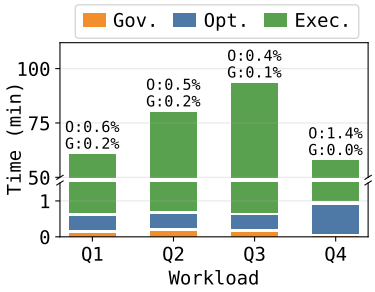


Figure 12: Per-workload overhead breakdown: execution, optimization, and governance time (two H200s, medium split).

faster. Once it completes, the ETL job accelerates to finish, confirming that Lumilake respects priorities, avoids starvation, and elastically reallocates freed resources.

Cross-job merging (Figure 11, right). Two users submit high-priority jobs over the same 160 stock symbols with overlapping workflows: `Q1-Social` and `Q2-Trading-Text` (a text-only variant of `Q2-Trading` that shares the data-retrieval stage with `Q1-Social`). As a baseline, we run each workload sequentially with a batch size of 10 (dashed lines); items from only one job appear per batch, so the optimizer cannot deduplicate shared stages. Sequential execution totals 126.5 minutes. When submitted concurrently with batch size 20, Lumilake deduplicates shared retrieval and analysis stages across the two workflows and completes both in 92.3 mins (1.37×).

4.5 Overhead Analysis

We measure the overhead of query optimization and governance on the medium data split with two H200s (Figure 12). Across all four financial workloads, the combined overhead is below 2% of end-to-end time, confirming that Lumilake's optimization and provenance tracking are inexpensive relative to the end cost of the workflows.

Table 3: Critical-path breakdown of one `Q2-Trading` batch (10 inputs, 2×H200, medium). 15/24 tasks lie on the critical path.

	LLM gen.	Model init.	Queue	Preproc.	Comm.	Cri. path
sec	275.0	110.3	85.4	61.6	39.0	571.3
%	48	19	15	11	7	100

4.6 Execution Trace Analysis

We examine the execution trace of a single `Q2-Trading` batch (Table 3). The governance layer records 1,210 timestamped events across 24 tasks and 265 data dependencies without user instrumentation. Of the 24 tasks, 15 lie on the critical path (571.3 s); the remaining 9 (SQL and S3 retrievals) execute on the CPU worker in parallel with GPU inference. LLM generation and model initialization account for 67% of the critical path, validating the optimizer's focus on KV-cache reuse and reducing model switches.

5 RELATED WORK

Section 2 positioned Lumilake against LLM inference engines, orchestration frameworks, classic and semantic data systems, and workflow optimizers. We discuss four additional themes below.

Shared computation and prefix optimization. SGLang's RadixAttention [42] organizes cached prefixes in a radix tree so that requests sharing a common prefix automatically reuse stored KV states. Prompt Cache [6] generalizes this by pre-computing attention states for reusable prompt modules, and CacheBlend [37] goes further by blending pre-computed and fresh KV states for partially overlapping contexts. The scope of reuse has grown from exact prefixes to partial overlaps, but all of these operate within a single serving endpoint. Lumilake schedules execution across an entire workflow DAG to maximize prefix sharing across nodes, inputs, and tenants.

Multi-tenant LLM serving. Orca [38] introduced continuous batching so that new requests join an in-flight batch at every decoding step; vLLM [12] complemented this with paged KV-cache management. At the model-placement level, AlpaServe [14] co-locates multiple models on shared GPU clusters via statistical multiplexing, and S-LoRA [30] serves thousands of LoRA adapters from a single base model through unified paging of adapter weights. All of these schedule at request granularity and cannot detect, for instance, that two tenants' workflows share an identical retrieval-plus-analysis subgraph. Lumilake adds a workflow-aware layer that batches and deduplicates at the DAG level (Section 4.4).

Heterogeneous DAG scheduling. Cluster schedulers have evolved from multi-dimensional bin packing (Tetris [7]) to DAG-aware packing (Graphene [8]) to learned policies via reinforcement learning over graph neural networks (Decima [17]). Sia [11] further adds heterogeneity awareness, jointly optimizing placement and parallelism across mixed GPU types. None of these model LLM-specific costs such as model-switch overhead, KV-cache reuse, or prompt structure. Lumilake's hybrid scheduler fills this gap by combining cache-aware ordering within homogeneous subgraphs with epoch-based DP over the heterogeneous remainder.

Data provenance and experiment tracking. LIMA [23] traces lineage inside ML pipelines at dataframe, column, and cell granularity, enabling deduplication of intermediate results. MLflow [39], Weights & Biases [2], and Neptune [20] take a lighter-weight approach, logging parameters, metrics, and artifacts for model training. LLM data systems such as LOTUS [21] and Palimpzest [15] have begun tracking semantic operator metadata. All of these cover flat pipelines; none capture the nested provenance of agentic workflows, where a top-level DAG triggers sub-DAGs of agent and subagent operations. Lumilake records provenance at agent-step granularity as a by-product of execution, without user instrumentation.

6 DISCUSSION AND CONCLUSION

What makes agentic analytics different? A key lesson from our experiments is that the optimization surface of agentic workflows is unlike that of SQL or Spark: the bottleneck is not data shuffling or I/O, but the sheer volume of LLM calls and model switches, and the primary lever is exploiting the structural regularity that templates create across inputs and users. This suggests that future agentic

engines should be co-designed with the serving layer, treating KV-cache state, model residency, and prompt structure as first-class objects in the optimizer's cost model, rather than layered on top of general-purpose infrastructure.

AI4Science, auto research, and agentic analytics. Scientific research is inherently analytics-heavy for querying databases, cross-referencing multi-modal evidence, and iterating through expert review; agentic workflows automate exactly these central activities. The emerging Auto Research paradigm [26], in which agents autonomously formulate hypotheses, run experiments, and synthesize findings, amplifies this by orders of magnitude: a single research campaign may launch thousands of agentic analytics runs. Efficient, governed execution of such workloads is precisely the problem Lumilake addresses.

Conclusion. We present Lumilake, an agentic analytics engine that treats template-based workflows as first-class DAGs over a data lakehouse. By combining LLM-aware query optimization, multi-tenant scheduling, and built-in governance, Lumilake outperforms Ray across all workloads on multi-GPU setups, scales to 50+ node DAGs with under 2% overhead, and achieves 1.37× speedup from cross-tenant deduplication—demonstrating that database principles can tame the computational explosion of agentic analytics on commodity hardware.

REFERENCES

[1] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. Online.

[2] Lukas Biewald. 2020. Experiment Tracking with Weights & Biases. https://www.wandb.com/

[3] Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous Chemical Research with Large Language Models. *Nature* 624, 7992 (2023), 570–578.

[4] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[5] Alireza Ghafarollahi and Markus J. Buehler. 2025. SciAgents: Automating Scientific Discovery through Bioinspired Multi-Agent Intelligent Graph Reasoning. *Advanced Materials* 37, 22 (2025), 2413523.

[6] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of Machine Learning and Systems (MLSys)*.

[7] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the ACM SIGCOMM Conference*. ACM, Chicago, IL, USA, 455–466.

[8] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Savannah, GA, USA, 81–97.

[9] CrewAI Inc. 2023. CrewAI: Fast and Flexible Multi-Agent Automation Framework. https://github.com/crewAIInc/crewAI

[10] LangChain Inc. 2024. LangGraph: Low-level Orchestration Framework for Building Stateful Agents. https://github.com/langchain-ai/langgraph

[11] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023. Sia: Heterogeneity-aware, Goodput-optimized ML-Cluster Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. ACM, Koblenz, Germany, 642–657.

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. ACM, Koblenz, Germany, 611–626.

[13] Nengxu Li, Xiuxiu Niu, Zijing Dong, Jingcong Hu, Ran Luo, Shuihua Yang, Qilin Zhou, Zhuojie Shi, Jinxi Chen, Xinyi Du, et al. 2025. Optimal perovskite vapor partitioning on textured silicon for high-stability tandem solar cells. *Science* 390, 6779 (2025), eadz3698.

[14] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 663–679.

[15] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. CIDR, Amsterdam, Netherlands.

[16] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, et al. 2026. Supporting Our AI Overlords: Redesigning Data Systems to Be Agent-First. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. Santa Cruz, CA, USA.

[17] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM SIGCOMM Conference*. ACM, Beijing, China, 270–288.

[18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, USA, 561–577.

[19] National Center for Health Statistics (NCHS). 2022. *National Health and Nutrition Examination Survey (NHANES) 2017–March 2020 Prepandemic File: Sample Design, Estimation, and Analytic Guidelines*. Technical Report. Centers for Disease Control and Prevention. https://wwwn.cdc.gov/nchs/nhanes/analyticguidelines.aspx

[20] Neptune Labs. 2019. Neptune: Experiment Tracking and Model Registry. https://neptune.ai/

[21] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4171–4184.

[22] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st International Symposium on Computer Architecture (ISCA)*. IEEE, Buenos Aires, Argentina, 118–132.

[23] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. ACM, Virtual Event, China, 1426–1439.

[24] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. ACM, Amsterdam, Netherlands, 1981–1984.

[25] Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *arXiv preprint arXiv:2505.14661* (2025).

[26] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. 2025. Agent Laboratory: Using LLM Agents as Research Assistants. In *Findings of the Association for Computational Linguistics: EMNLP 2025*. Association for Computational Linguistics, Suzhou, China, 5977–6043.

[27] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3035–3048.

[28] Junyi Shen, Noppanat Wadlom, and Yao Lu. 2025. Batch Query Processing and Optimization for Agentic Workflows. *arXiv preprint arXiv:2509.02121* (2025).

[29] Junyi Shen, Noppanat Wadlom, Lingfeng Zhou, Dequan Wang, Xu Miao, Lei Fang, and Yao Lu. 2025. FlowMesh: A Service Fabric for Composable LLM Workflows. *arXiv preprint arXiv:2510.26913* (2025).

[30] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2024. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. In *Proceedings of Machine Learning and Systems (MLSys)*.

[31] Padmanabhan Venkiteela. 2025. n8n: An Open-Source Workflow Automation Platform for Enterprise Integration and AI-Driven Orchestration. *International Journal of Computer Applications* 187, 63 (Dec 2025), 1–11. https://doi.org/10.5120/ijca2025926031

[32] Noppanat Wadlom, Junyi Shen, and Yao Lu. 2026. Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective. *arXiv preprint arXiv:2603.16104* (2026).

[33] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, et al. 2023.

Scientific Discovery in the Age of Artificial Intelligence. *Nature* 620, 7972 (2023), 47–60.
[34] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In *Proceedings of the Conference on Language Modeling (COLM)*. Philadelphia, PA, USA.
[35] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. 2022. Serving and Optimizing Machine Learning Workflows on Heterogeneous Infrastructures. *Proceedings of the VLDB Endowment* 16, 3 (2022), 406–419.
[36] Yijia Xiao, Edward Sun, Di Luo, and Wei Wang. 2025. TradingAgents: Multi-Agents LLM Financial Trading Framework. arXiv:2412.20138 [q-fin.TR] https://arxiv.org/abs/2412.20138
[37] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys)*. ACM, Rotterdam, Netherlands, 94–109.
[38] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, USA, 521–538.
[39] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin* 41, 4 (2018), 39–45.
[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, San Jose, CA, USA, 15–28.
[41] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2025. AFlow: Automating Agentic Workflow Generation. In *The Thirteenth International Conference on Learning Representations (ICLR)*.
[42] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. Curran Associates, Inc., Vancouver, Canada, 62557–62583.
[43] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Santa Clara, CA, USA, 193–210.

A FORMAL ANALYSIS

A.1 Cost Model Coefficients

In this section, we list the concrete coefficient choices for the cost model in Section 3.3. These can be obtained by profiling each operator class on the target hardware; we fix them to representative values in our experiments.

Objective weights. With multiple GPUs we set $w_{\max} = 0.93$, $w_{\Sigma} = 0.07$, prioritizing makespan with a small load-balancing term; with a single GPU, $w_{\max} = 1$, $w_{\Sigma} = 0$. w_{CPU} is scaled by the fraction of CPU nodes in the workload.

Coefficient values. For GPU operators with model size $|m(v)|$ (billions of parameters) and input count b:

$$C_{\text{TEXT}}(v) = \alpha_{\text{T}} \cdot |m(v)| + \beta_{\text{T}} \cdot b, \tag{5}$$

$$C_{\text{EMBED}}(v) = \alpha_{\text{E}} \cdot |m(v)| + \beta_{\text{E}} \cdot b, \tag{6}$$

$$C_{\text{VLM}}(v) = \alpha_{\text{V}} \cdot |m(v)| + \beta_{\text{V}} \cdot b, \tag{7}$$

$$C_{\text{DIFF}}(v) = (\alpha_{\text{D}} \cdot |m(v)| + \beta_{\text{D}} \cdot b) \cdot \tfrac{N}{30} \cdot \tfrac{H \cdot W}{768^2}, \tag{8}$$

where N is the number of diffusion steps and (H, W) the output resolution. We use $\alpha_{\text{T}} = 0.12$ s/B, $\beta_{\text{T}} = 0.03$ s/input for text inference and $\alpha_{\text{V}} = 0.18$ s/B, $\beta_{\text{V}} = 0.07$ s/input for VLM; embedding and diffusion coefficients follow the same structure. For S3 retrieval, $c_f = 0.005$ s and $c_t = 0.02$ s. The model initialization coefficient is $\lambda = 0.8$ s/B.

A.2 Optimality Proof

Definition 3 (Component-respecting schedule)**.** A schedule (σ, π, ρ) for G is *component-respecting* w.r.t. contracted components C if for every $C = (n_1, \ldots, n_k) \in C$ (ordered by Helium):

(1) *(Epoch dedication)* All nodes of C are assigned to a single epoch E_t, and E_t contains no other GPU node: $E_t \cap V_{\text{GPU}} = \{n_1, \ldots, n_k\}$. CPU nodes may share the epoch.
(2) *(Internal order)* ρ orders C's nodes as $(n_1, \ldots, n_k)$, i.e., in Helium order.

We write $\mathcal{S}_{\text{CR}}$ for the set of all component-respecting schedules, $\text{Opt} = \min_{(\sigma,\pi,\rho)} C(\sigma, \pi, \rho)$ for the unrestricted optimum, and $\text{Opt}_{\text{CR}} = \min_{(\sigma,\pi,\rho) \in \mathcal{S}_{\text{CR}}} C(\sigma, \pi, \rho)$ for the optimum within this class.

The optimality proof relies on the optimality guarantee of Helium and Halo, plus two lemmas.

Lemma 1 (Helium optimality [32]). *Let $S \subseteq V_{GPU}$ be a connected subgraph of same-model, text-only nodes. Among all topological orderings of S, the Helium order $(n_1, \ldots, n_k)$ minimizes the internal execution cost*

$$L_S = \sum_{i=1}^{k} \left[\mu_i \cdot C_{TEXT}(n_i)\right] + M \cdot C_{init}(n_1, w),$$

where the init term accounts for one model load per worker (n_1 triggers it; subsequent nodes in S share the same model). When S is sharded across M workers, each shard follows Helium order independently.

Lemma 2 (Halo optimality [28]). *Given a DAG $\hat{G}$ and worker pool W, Halo's epoch-based DP computes an optimal schedule*

$$(\hat{\sigma}^\star, \hat{\pi}^\star) = \arg\min_{(\hat{\sigma},\hat{\pi})} \hat{C}(\hat{\sigma}, \hat{\pi}).$$

Lemma 3 (Component-respecting optimum). *Under Assumption 2 and Lemma 1, for any feasible schedule (σ, π, ρ) there exists $(\sigma', \pi', \rho') \in \mathcal{S}_{CR}$ with $C(\sigma', \pi', \rho') \le C(\sigma, \pi, \rho)$. In particular, $Opt_{CR} = Opt$.*

Proof. Fix a contracted component $C = (n_1, \ldots, n_k) \in C$. We construct (σ', π', ρ') from (σ, π, ρ) in two steps, each of which does not increase the total cost. Step 1 aligns σ' (epoch partition); Step 2 aligns ρ' (execution order). Under Assumption 2, π' is determined: each component's nodes are distributed across all GPU workers, so π' requires no separate alignment.

Step 1 (Epoch dedication — aligning σ'). Suppose C's nodes are scattered across multiple epochs and may share those epochs with other GPU nodes. Move all nodes of C into a single epoch E_t, and move any non-C GPU nodes out of E_t into adjacent epochs. Nodes in different components use different models or have disjoint prefixes ($\mu = 1.0$ across components), so relocating the non-C GPU nodes does not change their cache bonuses. Within E_t, all nodes now share model $m(C)$, requiring only a single model load per worker. Hence $C(\sigma', \pi', \rho) \le C(\sigma, \pi, \rho)$.

Step 2 (Internal order — aligning ρ'). The nodes of C now occupy a dedicated epoch; set ρ' to the Helium order $(n_1, \ldots, n_k)$. By

Lemma 1, Helium order minimizes the total GPU load in C's dedicated epoch:

$$\sum_{w \in W_{\text{GPU}}} L_w(E_t)\Big|_{\rho'} \;\le\; \sum_{w \in W_{\text{GPU}}} L_w(E_t)\Big|_{\rho}. \tag{9}$$

By Assumption 2, costs of all nodes outside C are unaffected (they are in separate epochs). Applying Steps 1–2 to every $C \in \mathcal{C}$ yields $(\sigma', \pi', \rho') \in \mathcal{S}_{\text{CR}}$ with $C(\sigma', \pi', \rho') \le C(\sigma, \pi, \rho)$. □

Lemma 4 (Cost-preserving bijection). *Under Assumption 2, there exist inverse maps $\phi\colon \mathcal{S}_{CR} \to \mathcal{S}(\hat{G})$ and $\psi\colon \mathcal{S}(\hat{G}) \to \mathcal{S}_{CR}$ such that $C(\sigma, \pi, \rho) = \hat{C}(\phi(\sigma, \pi, \rho))$ and $\hat{C}(\hat{\sigma}, \hat{\pi}) = C(\psi(\hat{\sigma}, \hat{\pi}))$.*

Proof. *Contraction ϕ.* Given $(\sigma, \pi, \rho) \in \mathcal{S}_{\text{CR}}$, define $(\hat{\sigma}, \hat{\pi}) = \phi(\sigma, \pi, \rho)$ on $\hat{G}$ as follows. For uncontracted nodes, $\hat{\pi}(v) = \pi(v)$. For each component $C \in \mathcal{C}$, the super-node $\hat{v}_C$ is placed in C's dedicated epoch. In the original schedule, C's nodes are sharded across all GPU workers (Assumption 2); in the contracted graph, Halo treats $\hat{v}_C$ as a single unit and assigns it to one representative worker $\hat{\pi}(\hat{v}_C)$. This assignment is a bookkeeping device—Phase 3 expansion re-shards the super-node across all GPU workers, so the choice of representative does not affect the actual cost. For an epoch dedicated to super-node $\hat{v}_C$, the representative worker's load is $C_{\text{super}}(\hat{v}_C)$; for epochs containing only uncontracted nodes, the per-worker load follows the usual $\mu \cdot C_\tau + C_{\text{init}}$ formula. By Assumption 2, a component's dedicated epoch contains no other GPU nodes, so $C_{\text{super}}(\hat{v}_C)$ equals the total GPU load of that epoch in the original schedule. Uncontracted terms are identical, so $C(\sigma, \pi, \rho) = \hat{C}(\hat{\sigma}, \hat{\pi})$.

Expansion ψ. Given $(\hat{\sigma}, \hat{\pi})$ on $\hat{G}$, expand each $\hat{v}_C$ into $(n_1, \ldots, n_k)$ in a dedicated epoch, distributed across all GPU workers. The result is component-respecting by construction, and by Assumption 2, $C(\psi(\hat{\sigma}, \hat{\pi})) = \hat{C}(\hat{\sigma}, \hat{\pi})$. Since $\psi \circ \phi = \text{id}$ and $\phi \circ \psi = \text{id}$, the maps are inverses. □

Theorem 3 (Hybrid optimality). *Under Assumptions 1 and 2, the schedule $(\sigma^\star, \pi^\star, \rho^\star)$ produced by Algorithm 1 is optimal:*

$$C(\sigma^\star, \pi^\star, \rho^\star) = \textsc{Opt}.$$

Proof.

$$\begin{aligned}
\textsc{Opt} &= \textsc{Opt}_{\text{CR}} && \text{(Lemma 3)} \\
&= \min_{(\hat{\sigma}, \hat{\pi})} \hat{C}(\hat{\sigma}, \hat{\pi}) && \text{(Lemma 4)} \\
&= \hat{C}(\hat{\sigma}^\star, \hat{\pi}^\star) && \text{(Lemma 2)} \\
&= C(\sigma^\star, \pi^\star, \rho^\star). && \text{(Lemma 4: } \psi \text{ preserves cost)}
\end{aligned}$$

□