Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective

Anonymous Authors
Anonymous Institute
AnonymousEmails

Abstract

Agentic workflows are composed of sequences of interdependent Large Language Model (LLM) calls, and they have become a dominant workload in modern AI systems. These workflows exhibit extensive redundancy from overlapping prompts and intermediate results due to speculative and parallel exploration. Existing LLM serving systems, such as vLLM, focus on optimizing individual inference calls and overlook cross-call dependencies, leading to significant inefficiencies. This paper rethinks LLM and agent serving from a data systems perspective and introduces Helium, a workflow-aware serving framework that models agentic workloads as query plans and treats LLM invocations as first-class operators. Helium integrates proactive caching and cache-aware scheduling to maximize reuse across prompts, KV states, and workflows. Through these techniques, Helium bridges classic query optimization principles with LLM serving, achieving up to 1.48× speedup over state-of-the-art agent serving systems on complex workloads. Our results demonstrate that end-to-end optimization across workflows is essential for scalable and efficient LLM-based agents.

ACM Reference Format:
Anonymous Authors. 2018. Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

AI agents are autonomous LLM-based programs that act on a user's behalf [42, 65, 69]. They operate through agentic workflows [55, 61, 77, 80] that are goal-driven sequences of steps involving multiple LLM invocations (often using different prompts or tools), orchestrated to solve complex tasks. They have become a dominant workload in modern AI systems [3, 41, 68].

An emerging scenario of such workflows is that agents may explore many strategies or subtasks (e.g., trying alternative analyses) in parallel or in sequence to achieve their goal. This agentic speculation [41] is a high-throughput exploration to blend LLM serving with batch-style query processing. In other words, an agentic workflow resembles a batch analytics pipeline where LLM calls function as operators. While powerful, these workflows can issue a large volume of LLM queries, often with overlapping or repeated sub-queries, leading to significant redundancy and inefficiency.

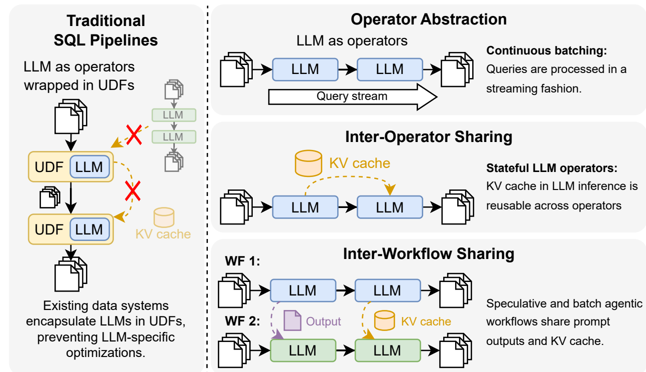

Figure 1: Three disparities between traditional SQL pipelines and agentic workflows with LLM as operators.

Recent advances in LLM serving systems have largely focused on optimizing individual LLM inference tasks. State-of-the-art serving engines such as vLLM [27] employ techniques like continuous batching [78], which dynamically group incoming requests to better utilize GPUs. These innovations, along with optimized GPU kernels and memory paging strategies [8, 25, 48, 76], have dramatically increased throughput and reduced latency for standalone LLM queries. However, existing LLM inference engines operate at the granularity of individual LLM calls, lacking visibility into the broader workflow structure. These solutions are not designed for batch agentic workflows that chain up multiple LLM calls (often with different prompts or agent roles). As a result, in complex multi-LLM pipelines, e.g., an agent that spawns several helper agents, each querying an LLM, these per-call optimizations fail to capture cross-call commonalities (e.g., shared sub-prompts or intermediate results). Recent work, such as AgentScope [15, 47], supports basic multi-agent pipelines, but they again treat each LLM call or agent as an individual unit, optimizing locally rather than globally. This gap leaves performance optimizations unexplored for batch agentic workflows.

How can we optimize agentic LLM workflows end-to-end? Many of the challenges mirror classic problems in query processing and optimization. An agent workflow can be represented as a directed acyclic graph (DAG) of operations: nodes perform data retrieval or invoke an LLM (i.e., *LLM-as-operators*), and edges represent data or prompt flow between operators. This is analogous to a complex relational query plan or a workflow in a data system. Prior work suggests that decades of database query optimization principles can be applied by modeling an agentic workflow as a query-plan

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

DAG and applying cost-based optimization. The approach of logical and physical optimization, i.e., rewriting plans, choosing operator implementations, and scheduling execution to minimize cost, contributes to the "agent-first" query processing techniques to handle the scale and complexity of LLM-driven workloads [10]. In essence, this is a familiar data systems problem framed in the context of LLM serving to eliminate redundant work by globally optimizing the workflow, much like a SQL optimizer would do for a complex relational query. However, at the same time, important new factors differentiate LLM workflows from traditional SQL pipelines. We identify and summarize the disparities of LLM-as-operators as follows and as illustrated in Figure 1:

- **Operator abstraction**: The operators in an agentic workflow wrap expensive LLM inference processes. The continuous batching technique in LLM serving systems becomes dominant to enable processing of multiple input queries in a streaming fashion without being blocked by other queries in the same batch. Within a single LLM call, the model maintains internal state (e.g., a KV cache of the prompt) and generates output token by token in an iterative fashion. In essence, the operators' batching mechanisms are no longer abstracted by classic relational functions like select or filters.
- **Inter-operator sharing**: As LLMs are inherently stateful, standard data-processing optimizations must be rethought. Across multiple calls, an agent may carry forward conversational context or reuse an earlier answer in a later prompt, along with the KV cache and model's internal state, creating stateful dependencies between operations.
- **Inter-workflow sharing**: For speculative and batch agentic workflows, prefix sharing among the queries from an input batch becomes an important technique to reuse partial or even entire agentic sub-chains across queries. This is also the case for workflows across different input batches to query common data sources (e.g., agents that query daily weather or news).

Unfortunately, current data systems are not tailored for these shifts and foresee significant challenges in efficiently processing agentic workflows. LLM operators often are wrapped in user-defined functions (UDFs) by current frameworks (e.g., Spark, Dask [54, 79]), while an LLM call is treated as a black-box UDF, which prevents introspection or special-case handling for performance. This lack of visibility precludes many optimizations: the system cannot automatically reuse common partial work (like shared prompt prefixes), pipeline LLM calls, or reorder operations based on LLM-specific cost factors. Importantly, continuous batching does not naturally happen beyond each individual UDF, causing blockage and significant performance degradation. Nevertheless, cost-based Volcano-style query optimization [19] principles still apply, as operator costs and even semantics are often readily available, but LLM-as-operators introduces a new form of batching abstraction, statefulness, and cost that traditional optimizers were never designed to handle.

To mitigate the paradigm shifts in agentic workflows and close the gap between the continuous batching abstract in LLM serving systems and the classic batching abstract in data systems, we consider a key missing jigsaw to be a new *workflow-aware LLM serving* that applies continuous batching while efficiently manages KV and prompt caches from a holistic, workflow perspective, instead of within the scope of individual UDFs. Specifically, it must simultaneously support: (i) inter-operator sharing, by enabling KV state communication across LLMs; (ii) inter-query and inter-batch sharing, by organizing caches according to prompt prefixes across multiple workflows, as well as (iii) an optimizer to maximize the chance of these sharing. While existing LLM serving systems (e.g., vLLM) employ prefix caching to reuse KV states from previously encountered prefixes, these approaches are inherently passive, optimized for online serving environments that cannot anticipate future workloads. In contrast, batch agentic workloads expose structures that can be leveraged for a more proactive caching strategy that exploits workload patterns to minimize redundancy.

In this paper, we propose Helium, a system that rethinks serving agentic workflows in modern data systems[1]. Helium introduces a novel proactive KV cache paired with a query optimizer to mitigate the drawbacks in today's passive, opportunistic KV cache sharing. By analyzing the workload during compilation, Helium recognizes shared prefixes across operators and workflows, and hence prewarms the cache to reduce redundant computation. Beyond that, Helium augments and leverages a cache-aware, cost-based query optimizer to enable rewriting query plans across the workload, maximizing KV state reuse in batch agentic workloads. Compared with state-of-the-art solutions, our prototype achieves up to a 1.48× speedup on a complex financial analysis workflow, with up to a 1.63× speedup on primitive workflows.

In summary, this paper makes the following contributions:

- We present Helium, a workflow-aware serving layer that models agentic workflows as query plans with LLM as first-class operators, enabling holistic optimization across intra- and inter-query execution.
- We introduce a novel proactive caching strategy that pre-warms KV caches for static prompt prefixes and maintains a global prompt cache to bypass redundant operators.
- We design a cost-based, cache-aware scheduling algorithm that leverages a templated radix tree to capture prompt structure and dependencies, maximizing prefix cache reuse across batch agentic workloads.
- We implement and evaluate Helium, demonstrating significant performance improvements over state-of-the-art systems while preserving exact semantics.

2 Background and Motivation

The rise of Large Language Models (LLMs) has shifted application development towards complex, multi-step *agentic workflows* [3, 61, 68]. Figure 2 demonstrates several representative workflow patterns. These workflows, which resemble traditional data processing DAGs using LLM calls as operators, often involve speculative execution, generating massive redundancy across prompts and results [41, 66, 74]. This transforms LLM serving from a simple inference into a problem that requires optimizing computational graphs where each operator is an expensive, stateful LLM call. This section first deconstructs prior work by examining several relevant pillars.

[1]Our source code is available at https://anonymous.4open.science/r/sxzFDjxv14Cz.

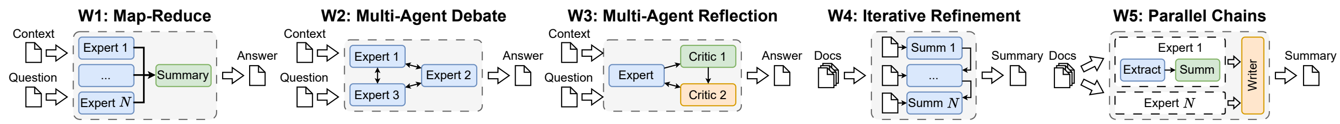

Figure 2: Each representative agentic workflow demonstrates a primitive pattern in agent interactions.

Optimizing LLM Serving for Latency. The first pillar of work focuses on optimizing the LLM operator itself in serving systems, which maximizes hardware utilization for streams of independent requests, treating each LLM call as a discrete unit of work. State-of-the-art engines like vLLM [27, 64] use techniques like *PagedAttention* to manage the key-value (KV) cache efficiently and *continuous batching* to dynamically group requests, significantly improving GPU utilization and throughput for LLM serving. To reduce redundant computation, these engines also implement prefix caching, reusing pre-computed KV cache for requests that share common prompt prefixes, often managed with an LRU eviction policy [27, 46, 81]. Aiming to improve the latency of LLM serving systems, these optimizations are fundamentally *local*, *workflow-agnostic*, and *reactive*, as the serving engine has no visibility into the workload or the broader DAG. This "operator-level myopia" prevents it from addressing cross-call inefficiencies. For instance, it cannot guarantee KV cache reuse for related queries within a workflow if they are separated by unrelated requests, as its optimization is limited to the immediate queries.

Orchestrating LLMs as Black-Boxes. The next pillar comprises frameworks that orchestrate the logical composition of agentic workflows. These tools provide high-level abstractions for building complex DAGs but treat the LLM operator as a black-box unit. Frameworks like LangChain [29] and LangGraph [30] simplify building multi-LLM applications by properly managing data and control flows. Traditional data systems like Spark [79] integrate LLMs as User-Defined Functions (UDFs), which can be *inference-agnostic*. By treating LLM invocations as black-box UDFs, the orchestration layer is blind to the internal mechanics of the LLM operator; critical performance factors, such as the stateful KV cache and the bimodal prefill/decode cost structure [83], are hidden from the optimizer, preventing the system from making intelligent, cost-based decisions.

Challenges and Our Ideas. Prior LLM serving and data systems were originally designed under different contracts (classic vs continuous batching). We observe the following critical challenges:

- *KV cache in single operators:* In multi-agent debates [11, 37], each turn builds on shared conversational history, yet current systems redundantly reprocess the entire context. An ideal serving layer should instead extend the KV cache, converting costly recomputation into lightweight incremental updates.
- *Passive prefix caching*. Existing prefix caching is passive and opportunistic in reusing KV states, as the incoming queries are unpredictable during online serving. Prior LLM serving systems did not leverage workload patterns in batch agentic workflows.
- *Cost modeling and optimization:* Integrating LLMs in UDFs causes a significant disconnect: the query optimizer is blind to physical execution, while the execution engine (LLM serving systems) is blind to the logical plan.

Rethinking the abstract of LLM serving in data systems, our key ideas are two-fold. First, instead of employing passive, opportunistic KV cache sharing in prior LLM serving solutions, we propose a holistic, proactive caching mechanism in batch agentic workflows to reuse KV cache across operators and workflows. Next, Helium augments and leverages a cost-based query optimizer to pair with the proactive cache, thus enabling query plan rewrite and maximizing cache reuse across operators and workflows. These techniques combined contribute to workflow-aware LLM serving tailored to modern data systems for agentic workflows.

Scope. We employ the following scope and simplifying assumptions in our solution:

- *Semantic preserving*: Optimized executions produce the same results as naive ones. We avoid approximations (e.g., proxy models) that trade accuracy for speed, focusing on unstructured data analytics rather than SQL-generation tools for structured data.
- *On-premise deployment*: We study on-prem execution with multiple GPUs, enabling fine-grained scheduling, memory, and resource control, instead of using off-the-shelf cloud APIs.
- *Simplifying assumptions*: We limit our scope of agents to calling LLMs and performing local data operations only, without remote API calls. Also, we constrain the agentic workflows used in this work to use the same base LLM. They can configure with different prompts to simulate different agentic roles.

3 System Overview

Helium adopts a classic multi-stage query processing architecture [18, 21], dividing execution into parsing, optimization, and processing phases. Each *agentic workflow* is represented using a procedural language that specifies the individual LLM or agent calls and their dependencies. The system treats each workflow definition as a template that will be evaluated over a batch of input instances.

Parsing Agentic Workflow DSL. We use a domain-specific language (DSL) to represent batch agentic workflows that share the structure but with different input prompts. Our parser translates them into a DAG. Each operator in the DAG corresponds to a prompt-related action, such as invoking a model with a given prompt, retrieving data, or running a custom transformation. Helium's design is inspired by the dataflow representation in TensorFlow [1] that first constructs a symbolic graph of operators, using placeholders to mark where actual prompt tokens or data will be fed in at execution time. During execution, each query and the prompts *flow* through the DAG. Unlike TensorFlow, Helium allows cross-operator continuous batching such that ready outputs can be forwarded to the next operators without blockage. More details come in Section 6.

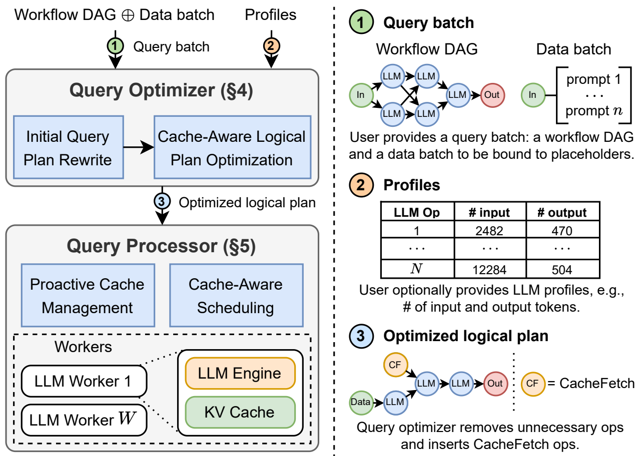

Figure 3: Overview of Helium's architecture.

Logical Plan Optimization. Helium 's query optimizer rewrites the logical DAG to eliminate redundancy and exploit shared computation across the batch workload. It applies rewrite rules to prune and merge operators. Redundant nodes (e.g., identity operators or unused branches) are removed, and identical subgraphs that occur across queries are consolidated. This is akin to common subexpression elimination in databases, such that redundant subplans are detected and computed only once. After structural optimization, the optimizer performs cache substitution: for each operator, the optimizer checks if a matching entry exists in a global prompt cache that maps the inputs of deterministic operators to their outputs. On a cache hit, the corresponding LLM operator is replaced by a lightweight `CacheFetch` operator. This transformation converts a computational dependency into a simple data retrieval.

Execution Planning and Processing. Helium then constructs a *templated radix tree* (TRT) over the optimized logical plans to capture the prompt structure and identify commonalities. This TRT serves as the primary input for Helium's cache-aware scheduling that uses a cost-based model to assign operators to workers and determine an execution order that balances load and maximizes KV cache reuse for shared prompt prefixes. Helium's query processor then executes the plan with our proactive caching mechanism. Specifically, for static prompt prefixes identified by the TRT, proactive caching pre-computes and stores these states in the GPU memory during the first execution; in subsequent batches, workers can directly reuse these tensors to avoid prefill. Meanwhile, a global proactive prompt cache is maintained to store the full outputs of deterministic operators, allowing the system to bypass entire operator executions for repeated inputs. These efforts allow Helium to accelerate batch agentic workflows with unchanged outputs.

4 Query Optimizer Design

The query optimizer identifies sharing opportunities at multiple granularities, from entire sub-workflows to prompt prefixes, thus constructs the logical plan for execution. This process bridges the gap between workflow-agnostic LLM serving engines and inference-agnostic orchestration. The optimization process comprises two stages.

Initial Plan Pruning. User-defined agentic workflows, especially those with speculative execution, often contain structural redundancies and inefficiencies such as dead code or duplicated computations [4, 41, 74]. This stage produces a clean graph representation that simplifies the problem space for subsequent optimizations.

- *Operator Pruning*. Analogous to dead code elimination in compilers, this transformation removes operators that do not contribute to the final output. The optimizer performs a backward traversal from the designated output nodes, identifying and pruning any operator whose output is not consumed on a valid execution path. In multi-agent workflows, this efficiently removes speculative branches that are rendered eventually.
- *Common Subgraph Elimination*. Next, the optimizer applies common subgraph elimination (CSE) [56], to identify and merge structurally identical subgraphs that share the same inputs. Subgraphs are hashed based on their topology and input node identifiers to detect duplicates efficiently. This ensures a computation with specific inputs is executed only once, and the prompt prefixes associated with this computation can be uniquely identified. For example, in the Map-Reduce pattern in Figure 2, multiple agents can be initialized with the same but repeatedly defined context. Without CSE, the preparation of this context incurs redundant computations, and the shared prompt prefixes containing this context cannot be identified.

Logical Plan Optimization with Prompt Cache. Next, Helium leverages a global prompt cache to bypass redundant computation at the operator level, as agentic workflows often repeat entire tasks within or across queries. This cache maps the inputs of deterministic operators to their previously computed outputs. The optimizer performs a recursive, bottom-up traversal of the DAG. For each operator, it computes a signature from its type and the values of its materialized inputs (i.e., constants or values resolved from the cache); the signature is used to probe the prompt cache. Upon a cache hit, the operator is marked; during the resolution stage, the optimizer replaces it with a `CacheFetch` operator. This lightweight operator stores a pointer to the cached result. This transformation fundamentally alters the data flow; a computational dependency (e.g., a `Summarizer` agent depending on an `Expert` agent) is rewired into a simple data retrieval dependency. This allows workers to pull the results from the cache instead of pulling from the original worker. To ensure unchanged results and reproducibility, this caching mechanism is restricted to deterministic operators, such as non-LLM operators and LLM operators with greedy sampling (e.g., zero temperature), and uses LRU as the eviction policy.

These transformations produce an optimized *logical* plan to express the *intent* to fetch from a cache or execute in parallel, without binding operations to specific workers or timelines. This allows the query processor to generate physical plans based on runtime resource availability and updated cost models.

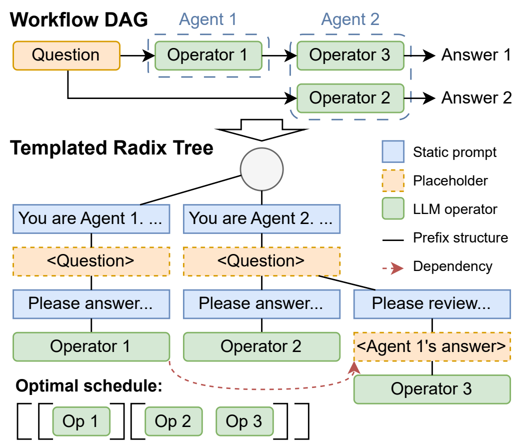

Figure 4: A workflow DAG (top) and the corresponding templated radix tree with cache-aware schedule (bottom)

5 Query Processor Design

Modeling Prompt Structure with a Templated Radix Tree. By representing an agentic workflow as a DAG, Helium can infer the structure of input prompts among operators and identify opportunities for prefix cache reuse. This is done with a *templated radix tree* (TRT), a novel data structure upon the standard radix tree [45]. The TRT represents the prefix structure of both static prompt components and dynamic parts derived from the outputs of other operators. It also captures the dependencies among its leaf nodes.

Formally, a TRT $T = (V, E, E')$ consists of a set of nodes V, a set of edges E representing the prefix structure, and a set of directed edges E' connecting the leaves $L \subset V$. Let r denote the root of T. Each intermediate node $v \in V \setminus (L \cup \{r\})$ is associated with a sequence of tokens and placeholders, denoting token sequences to be filled by other operators' output. Each leaf $l \in L$ is associated with an LLM operator whose input prompt structure is defined by the path from the root r to its parent. The leaves L and dependency edges E' form a DAG $G = (L, E')$, representing the dependencies among the operators. Specifically, an edge $(l_1, l_2) \in E'$ indicates that the operator at l_2 depends, directly or indirectly, on the output of the operator at l_1. The TRT effectively captures the prefix structure of the operators' input prompts and their dependencies. We describe the algorithm for constructing the TRT in Appendix A.

In modern LLM inference engines, the cache of common prefix tokens across different calls can be *reused* to improve prefill efficiency [27, 81]. Due to limited GPU memory, the prefix KV cache may be prematurely evicted to make space for newly scheduled calls. Prior work [81] proposed organizing the KV cache as a radix tree and scheduled LLM calls in order of their shared prefix length. While this online strategy maximizes prefix cache reuse when all LLM calls are available upfront, it is suboptimal for agentic workflows. Consider a workflow with two agents (the top panel of Figure 4): Agent 1 generates an answer (Operator ①); Agent 2 answers the same question while provideing feedback on Agent 1's answer (Operator ② and ③). Based on its templated radix tree, an optimal schedule would be ① → ② → ③, as this enables full reuse of the KV cache for Agent 2's system prompt between Operator ② and ③. However, the online algorithm might yield a suboptimal schedule like ② → ① → ③. In this case, the KV cache for Agent 2's prompt might be partially evicted to process Operator 1, leading to recompute and reduced efficiency for Operator 3. To address this, we require a new solution to capture the prompt structure across the entire workflow and a scheduling algorithm that leverages this structure to minimize execution cost.

Proactive Cache Management. Batch agentic workflows are highly redundant within and across batches. Queries in a batch often share the same structure, repeating prompt prefixes and intermediate results. Because agents repeatedly probe similar information, many prompts and outputs remain unchanged across runs. For example, a trading agent summarizing daily company reports sees near-identical inputs, duplicating LLM computation. Helium improves efficiency through a proactive KV caching mechanism.

Specifically, Helium 's query processor uses the TRT, constructed during scheduling, to identify static prompt prefixes that are invariant across batches. During the first execution of a workflow, the query processor precomputes and stores the KV cache for these static token sequences in GPU memory. In subsequent batches, the LLM engines can directly use these precomputed KV tensors, avoiding redundant prefill computations.

Scheduling Problem Formulation. To map each operator to the workers, we model our scheduling task as a multi-worker scheduling problem to find an assignment of calls to workers and an execution order that optimizes the *makespan* (i.e., wall-clock latency) of the execution. Specifically, Helium leverages the *token usage* for each LLM call, and the total *token step* of the entire workflow. The former is the total number of tokens it occupies across all its inference steps. A token step serves as the unit of time in our model, where each LLM call consumes a number of token steps proportional to its token usage. This formulation allows us to model the effect of prefix cache sharing by reducing a call's token usage by the number of shared prefix tokens. To account for batching parallel calls for a single inference step, we impose *precedence delay constraints* that define the minimum number of token steps that must pass between a call and another call that depends on its output. This incentivizes the scheduler to intersperse dependent calls with independent ones, thereby increasing the potential batch size. We consider prefix sharing only in consecutive LLM calls on the same worker. Our cost model uses a call-level TRT, where each leaf corresponds to an individual LLM call rather than an operator.

Let $T = (V, E, E')$ be a TRT with root r and leaf set L, where each $l \in L$ corresponds to an LLM call. Let $G = (L, E')$ be the dependency DAG among the calls. Each node $v \in V$ has an associated weight $\omega(v)$, where $\omega(v) = 0$ if $v \in L \cup \{r\}$, and otherwise $\omega(v)$ is the number of tokens in the prompt segment associated with v. Let W be the set of LLM workers, and let M_i be the KV cache capacity (in tokens) of worker $i \in W$. We partition the set of leaves L into $|W|$ disjoint subsets $L_1, \ldots, L_{|W|}$ corresponding to the workers. We denote a permutation (schedule) of the calls assigned to worker i as $\sigma_i = (l_1^i, \ldots, l_{|L_i|}^i)$, where $l_k^i \in L_i$. The overall schedule is $\sigma = (\sigma_1, \ldots, \sigma_{|W|})$. The cost of each LLM call is modeled by its token usage and precedence delays to capture batching effects:

- *Token usage.* Let $u(i,j)$ be the sequence-dependent token usage of the j-th call, l_j^i, in worker i's schedule. We decompose $u(i,j)$ into prefill usage $u_p(i,j)$ and decode usage $u_d(i,j)$. The prefill usage is the number of new tokens to be processed, not shared with the previous call l_{j-1}^i:

$$u_p(i,j) = \begin{cases} \sum_{v \in \mathrm{path}(r, l_j^i)} \omega(v), & \text{if } j = 1 \\ \sum_{v \in \mathrm{LCApath}(l_{j-1}^i, l_j^i)} \omega(v), & \text{otherwise} \end{cases}$$

where $\mathrm{path}(v_1, v_2)$ is the set of nodes on the path from v_1 (exclusive) to v_2 (inclusive), and $\mathrm{LCApath}(v_1, v_2)$ is the set of nodes on the path from the lowest common ancestor of v_1 and v_2 (exclusive) to v_2 (inclusive). The decode token usage models the cumulative token count over all generation steps:

$$u_d(i,j) = \frac{\mathrm{len}_{\mathrm{out}}(l_j^i)\left(\mathrm{len}_{\mathrm{out}}(l_j^i) + 1\right)}{2}$$

where $\mathrm{len}_{\mathrm{out}}(l)$ is the estimated number of output tokens for the call associated with leaf l. The total token usage is then:

$$u(i,j) = \alpha_i \left(\mathrm{len}_{\mathrm{out}}(l_j^i) \times u_p(i,j) + u_d(i,j)\right)$$

where α_i is a normalization constant accounting for worker performance differences. If all workers use identical hardware and models, we can set $\alpha_i = 1/M_i$.
- *Precedence delay.* To model the batching effect, we define a precedence delay $d(i,j)$. Any call that depends on the output of call l_j^i must be scheduled at least $d(i,j)$ token steps after l_j^i completes. This delay is given by:

$$d(i,j) = \alpha_i M_i \times \mathrm{len}_{\mathrm{out}}(l_j^i)$$

The total token step $T(\sigma)$ for a schedule σ is the token step at which the last LLM call finishes, analogous to the makespan in classical scheduling problems.

Putting these together, the scheduling problem is therefore to find a schedule σ that minimizes the total token step. Let $b(i,j)$ and $c(i,j)$ be the start and completion token steps, respectively, of call l_j^i. The problem is formulated as:

$$\text{Minimize } T(\sigma) = \max_{\substack{i=1,\ldots,|W| \\ j=1,\ldots,|L_i|}} c(i,j) \text{ such that}$$

$$\begin{aligned} b(i,j) &\geq c(i',j') + d(i',j'), && \forall (l_{j'}^{i'}, l_j^i) \in E' \\ c(i,j) &= b(i,j) + u(i,j), && i \in \{1,\ldots,|W|\}, j \in \{1,\ldots,|L_i|\} \end{aligned}$$

Here, the first constraint enforces dependency requirements, ensuring a call starts only after its predecessors are completed and the precedence delay has passed. The second constraint defines the completion step of each call. The above optimization problem is NP-hard. This can be shown by a reduction from the parallel machine scheduling problem for makespan minimization, which is a well-known NP-hard problem [32].

Solver by Cache-Aware Scheduling. We propose a cost-based, cache-aware greedy algorithm to guide operator scheduling. Our approach is inspired by prior work in which a DFS traversal of a radix tree yields an optimal schedule for prefix cache reuse [81]. However, a direct application at the LLM call level is impractical due to: (1) runtime dynamics, such as unknown output lengths and complex batching behaviors in LLM engines, make the cost model inaccurate and can lead to costly pipeline stalls; and (2) the scheduling complexity would scale with the batch size, which can be unbounded. Our algorithm instead works on an operator-level TRT derived from the logical plan; the complexity thus depends on the workflow structure rather than the batch size. It takes the optimized DAG and offline-profiled operator statistics (e.g., average output token counts) as input and produces a *soft schedule.* This schedule consists of nested sequences of LLM operators for each worker, allowing reordering at runtime to adapt to system dynamics.

Algorithm 1 Helium's Cache-Aware Scheduling

```
function SCHEDULE(workflow_dag)
    partitioned_dag ← PARTITIONWORKFLOW(workflow_dag)
    tree ← BUILDSCHEDULINGTREE(partitioned_dag)
    while not RECURSE(tree, tree.root, false) do
        RECURSE(tree, tree.root, true)   ▷ Force progress if stuck
    return tree.GETFINALSCHEDULE()

function RECURSE(tree, node, force)
    if node.is_leaf then
        if tree.CANSCHEDULE(node, force) then
            tree.SCHEDULENODE(node)
    else
        for each child in SELECTCHILDREN(node, force) do
            if RECURSE(tree, child, force) then
                node.REMOVECHILD(child)
            tree.UPDATESTATE(node.children)
            force ← false                ▷ Only force first child
    return node.ISEMPTY()
```

Algorithm 1 partitions workflow operators across LLM workers, replicating operators per assignment to balance load and shrink the search space. From the partitioned DAG, we build the TRT and a mirrored scheduling tree to track dependencies and states. A DFS-style recursion chooses the next child via a critical-path heuristic [26], prioritizing the subtree with the greatest aggregated dependency depth and earliest schedulable token step. At leaves, operators are scheduled when data and precedence delays allow; if blocked, we force the operator with the earliest start to ensure progress. The query processor then dispatches schedules; workers execute best-effort, issuing ready LLM calls for the query with the most pending calls to saturate GPUs while preserving cache-friendly ordering.

It is notable that the nested sequence structure is crucial for maximizing cache reuse across different levels of prompt sharing. For instance, in Figure 4, while Operators ② and ③ share a static system prompt, the subsequent question prompt is unique to each query in a batch. A simple operator-by-operator schedule would thrash the cache by alternating between different queries' question prompts. Our algorithm mitigates this by grouping operators that share static prefixes into inner sequences. This structure allows workers to process LLM calls query-by-query *within* an inner sequence (to reuse per-query prefix) and operator-by-operator *across* inner sequences (to reuse static prefix). An intermediate buffer is used to collect operators, which are released as a complete inner

```python
from helium import graphs, helium, ops
# Define the agentic workflow
q = ops.placeholder("q")
answer = ops.llm([ops.Msg("user", q)])
revise_prompt = ops.fmt("Revise answer...", q, answer)
fin_answer = ops.llm([ops.Msg("user", revise_prompt)])
# Build and compile the DAG
graph = graphs.build([fin_answer]).compile(q=["How many
    inches is 1 meter?"])
# Execute the DAG with Helium runtime
print(helium.invoke(graph))
```

Listing 1: Example usage of Helium**'s DSL**

Op	Type	Description
Data	IO	Encapsulates a data batch as a node in the DAG.
Input	IO	Placeholder for data provided at compile time.
Output	IO	Materializes data as a final output of the workflow.
Format	Prompt	Formats input text using specified arguments.
Lambda	Prompt	Applies an arbitrary Python function to the input data.
LLM	Model	Generates LLM responses for input conversation contexts.

Table 1: Examples of Helium's primitive operators.

sequence to the final schedule once all operators sharing the same static prompt have been emitted. Our algorithm has a time complexity of $O(|V_{int}| \cdot c_{max}^3 + |E'| \cdot d_{max})$, where $|V_{int}|$ is the number of internal TRT nodes, c_{max} is the maximum branching factor, $|E'|$ is the number of dependency edges, and d_{max} is the maximum TRT depth. The proof is detailed in Appendix B.

6 Implementation

Agentic Workflows DSL. We implement the DSL as a Python front end that, under a lazy dataflow model, constructs a *symbolic* DAG of primitive operators (Table 1); operator calls record nodes and edges instead of executing. Listing 1 shows a minimal example of the DSL. Each call to an `ops` primitive creates a node with its operator kind and arguments, and passing symbolic handles as arguments links nodes via dependency edges. Inputs are declared explicitly with `ops.placeholder()`, which creates a named input node that can be referenced by downstream operators (e.g., as a message argument in Listing 1). Users author workflows by composing operators and supplying the terminal nodes to `graphs.from_ops()`; this constructor traverses dependencies from the terminals to materialize the dependency graph. Therefore, compilation binds inputs to the graph without executing the workflow. Specifically, `compile()` binds placeholders *by name* to a concrete input batch while preserving the graph's symbolic structure, yielding a compiled graph that is ready for optimization and scheduling. Finally, `invoke()` submits the compiled graph to the runtime, which applies graph rewriting, generates a cache-aware execution plan, and dispatches work to the assigned workers (Sections 4 and 5).

LLM Engine Integration. Helium is built on top of vLLM v0.8.5 [64]. Each worker manages a dedicated vLLM engine instance, each running in a separate process and communicating with the worker via IPC message queues. To implement our proactive caching strategy, we augment vLLM to support pinning the KV caches of precomputed prefixes. During a precomputation phase, the Helium query processor dispatches specially annotated prefill requests to the vLLM engine. The engine executes these requests to populate the KV cache and retains the resulting state in GPU memory for reuse in subsequent batches. While these precomputed caches are prioritized, they may be evicted under high memory pressure and are recomputed at the beginning of the next batch.

Job Profiling. Helium requires statistics of LLM operators, such as their average output token counts, to make effective scheduling decisions. Helium provides APIs for users to profile their agentic workflows and submit these statistics along with their jobs. For our experiments, we profile all benchmark workflows offline using a different query set. In practice, the profiling overhead is minimal compared to the execution cost with a small sample of queries. This one-time cost can be amortized across workflows later on.

Workflow	Datasets	Configuration
MapRed	MMLU, TAT-QA	14 experts for MMLU, 7 experts for TAT-QA with 7 distinct roles
Debate	MMLU, TAT-QA	3 agents with distinct roles, 2 rounds of debate
Reflect	TAT-QA	1 expert, 2 critics
Iterative	Amazon	6 review chunks, 10 reviews each
Parallel	Amazon	7 experts, each processing 6 review chunks

Table 2: Configurations for primitive workflows.

7 Evaluation

We evaluate Helium to assess its performance and efficiency in orchestrating real-world agentic workflows. Our experiments are guided by the following targets:

- RQ1 How does Helium perform on microbenchmark agentic workflows that exhibit primitive patterns?
- RQ2 How does Helium's end-to-end performance compare to state-of-the-art solutions on complex agentic workflows?
- RQ3 How do Helium's key components contribute to the overall system performance? We conduct an ablation study.
- RQ4 How does Helium perform under different configurations and workload constraints? We conduct a sensitivity study.

In the following subsections, we first investigate RQ1 using five primitive agentic workflows illustrated in Figure 2. We then evaluate our solution on more sophisticated workflows that combine multiple primitive patterns to demonstrate RQ2. This is followed by ablation and sensitivity studies to cover RQ3 and RQ4.

7.1 Microbenchmarks

Primitive Workflows. We consider five primitive agentic workflows illustrated in Figure 2. Table 2 illustrates more details on the configurations.

- Map-Reduce (*MapRed*): This workflow consists of multiple expert agents that concurrently process input contexts and questions, followed by a summarizer agent that aggregates the experts' outputs to produce the final answer.
- Multi-Agent Debate (*Debate*): Following [11], this workflow involves multiple agents debating over a given context and question before arriving at a final answer.

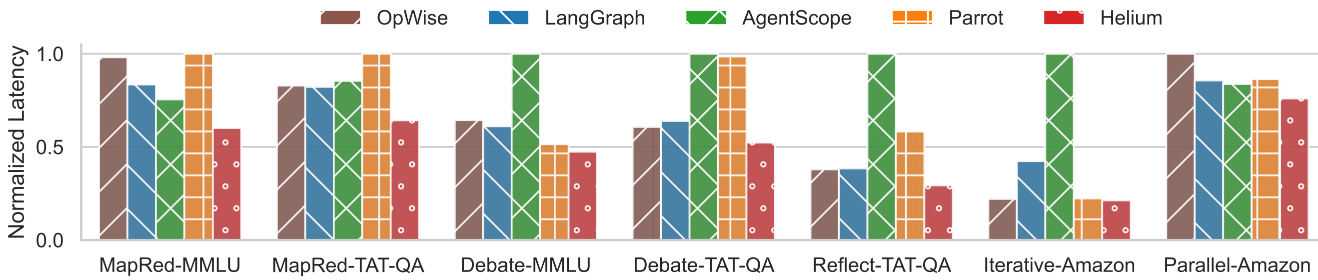

Figure 5: Normalized end-to-end latency of Helium **and baselines, excluding *vLLM*, across representative workflows and datasets with Qwen3-8B. Values are normalized within each workload so that 1.0 equals the slowest system (lower is better).**

- Multi-Agent Reflection (*Reflect*): This workflow uses an expert agent to draft an answer, followed by multiple critic agents that provide feedback to refine the initial answer.
- Iterative Refinement (*Iterative*) [28]. Input documents are divided into smaller chunks, and a summarizer agent processes each chunk iteratively, refining the previous summary before producing the final result.
- Parallel Chains (*Parallel*). The workflow consists of multiple expert agents with distinct roles that extract insights from separate input chunks (e.g., customer reviews), followed by a writer agent that aggregates the insights into a coherent report.

We use datasets from MMLU [22], TAT-QA [84], and Amazon Reviews [24]. For *MapRed* and *Debate*, we sample 200 questions from MMLU and 100 contexts from TAT-QA (each with 6 questions). For *Reflect*, we use 200 contexts from TAT-QA (each with 6 questions). For *Iterative* and *Parallel*, we sample 200 and 100 items from Amazon Reviews, respectively, each with 60 reviews divided into 6 chunks. Since we evaluate performance on a single batch, only proactive KV caching and cache-aware scheduling are enabled.

Baselines. We compare Helium against baselines and state-of-the-art LLM serving systems and agentic workflow orchestration:

- *vLLM* [27]: a high-throughput LLM inference engine. This baseline implements agentic workflows naively by executing each query's operators *sequentially* on an unmodified vLLM backend. This setup is the same as that in Helium to ensure fairness.
- *OpWise*: executes the workflow DAG *operator by operator* across the batch, similar to classical batch analytics systems (e.g., Spark, Dask [54, 79]). In contrast to the *vLLM* baseline's query-wise execution, *OpWise* traverses the DAG in topological order and executes the operator for each query in the batch before moving to the next operator, thereby improving data-parallel efficiency.
- *LangGraph* [30]: a graph-based orchestration framework for long-running, stateful agents. We map our workflows directly to its graph abstraction and execute on the full batch using LangChain's Runnable interface for asynchronous batch execution [29].
- *AgentScope* [15]: a multi-agent platform with an *actor-based distributed* execution mechanism that parallelizes agent runs and exchanges messages among decentralized agent servers [47]. We implement the workflows in AgentScope v0.1.6, passing the entire data batch as the initial inputs.

System	Relative Latency (× vs. Helium)		
	Average	Min	Max
vLLM	78.11	43.39	112.45
OpWise	1.30	1.04	1.63
LangGraph	1.37	1.13	1.99
AgentScope	2.26	1.10	4.69
Parrot	1.48	1.05	1.99
Helium	**1.00**	**1.00**	**1.00**

Table 3: Relative latency of each system normalized to Helium **across representative workflows (lower is better). Values** > 1 **indicate higher latency than** Helium**.**

- *Parrot* [38]: an LLM service system that exposes application-level knowledge via *Semantic Variables* and performs *dataflow analysis* across requests. For each workflow, we submit all requests for all agents and batch items up front, allowing Parrot to optimize over the whole workload.

Models and Testbed. Our evaluation employs the Qwen3-8B and Qwen3-14B models [72]. All experiments are conducted on a machine equipped with an AMD EPYC 9554 64-Core Processor and two 94GB NVIDIA H100 NVL GPUs. We use vLLM v0.8.5 [27], with automatic prefix caching and chunked prefill enabled, as the LLM inference engine throughout the experiments. Each model instance is deployed on a single GPU by spawning a separate vLLM instance.

Since the baseline frameworks do not natively support integration with vLLM v0.8.5, we deploy a standalone LLM inference server and interface with the baselines through the OpenAI API. For multi-engine experiments, because the baselines, except for *Parrot*, lack built-in request routing, we implement a lightweight wrapper engine that load-balances requests across worker engines based on request queue length. For reproducibility, we employ greedy sampling for all LLM calls across all experiments.

Evaluations. The results are shown in Figure 5 and summarized in Table 3, demonstrating Helium's robust performance across all representative workflows. As expected, Helium achieves a speedup of up to 112.45× over the naive *vLLM* baseline, as it was not tailored for agentic workflows. Helium is up to 1.63× faster than *OpWise*, whose operator-by-operator execution model becomes a bottleneck in highly parallel workflows (*MapRed*, *Parallel*) and is inefficient for

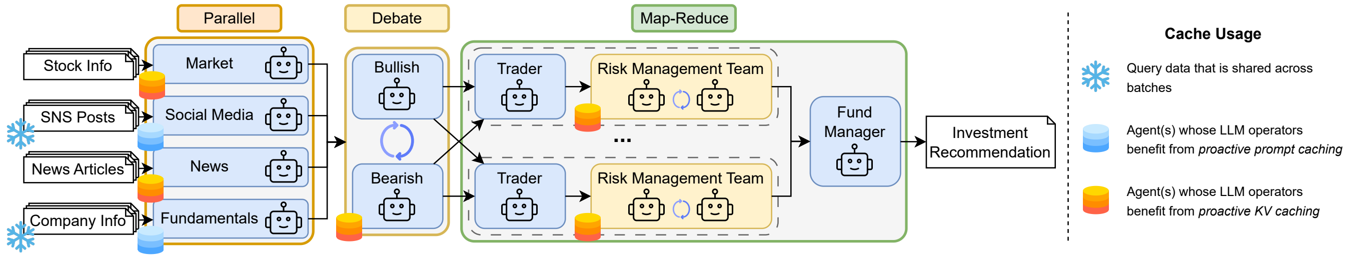

Figure 6: The *Trading* workflow used for end-to-end evaluation, combining the *Parallel*, *Debate*, and *Map-Reduce* patterns. Agent annotations indicate opportunities for proactive KV or prompt caching, applied to prefixes over 200 tokens.

workloads with cache patterns that favor a query-by-query execution order. Helium outperforms *LangGraph* by up to 1.99× on workflows with high operator parallelism or long dependency chains (*MapRed*, *Iterative*), exposing the limitations of generic graph execution. Helium is up to 4.69× faster than *AgentScope*, whose agent-level parallelism is ill-suited for workflows where a few agents are invoked repeatedly (*Debate*, *Reflect*, *Iterative*). Finally, Helium is up to 1.99× faster than *Parrot*. Although *Parrot* is prefix-aware, its scheduling heuristics cause severe load imbalance on workflows with specific prompt patterns (e.g., *MapRed*, *Debate* with TAT-QA, and *Reflect*). Helium outperforms the baselines in all cases.

7.2 End-to-End Benchmark

Setups. To demonstrate Helium's performance on a realistic task and answer RQ2, we construct a complex agentic workflow named *Trading*, combining multiple primitive patterns from the microbenchmark. The workflow aims for investment recommendation and combines *MapRed*, *Debate*, and *Parallel*, mirroring the workflow of a financial trading application proposed in prior work [70]. Specifically, the *Trading* workflow has three stages: *analyst*, *research*, and *decision*. The analyst stage follows the *Parallel* pattern, with four agents (market, social media, news, and fundamentals) that analyze and summarize different documents; i.e., the market analyst processes stock price history, while the fundamentals analyst reviews company financial statements. The research stage uses the *Debate* pattern, where two researcher agents debate the analysts' findings, mediated by a manager. The final decision stage resembles the *MapRed* pattern but with nested complexity. It branches into eight chains, each containing a trader agent with a specific behavior (e.g., value trader, news trader [20]). The trader makes an initial decision, which is then evaluated by three risk management agents in a multi-turn debate. A fund manager agent aggregates the outputs from all chains to produce the final investment recommendation.

The models used and the experiment environment are the same as those in the previous microbenchmark section. Inspired by the task in [70], we created a financial dataset using data from a few finance data sources [13, 62, 71]. We collected data for 100 stocks over two consecutive days. The first day's data is used to warm up the system, while the second day's data is used for evaluation. This setup simulates a daily trading scenario where some data, such as company fundamentals, is relatively static across batches. The workflow, with its branching and repeated sub-tasks, simulates a complex and computationally intensive workload.

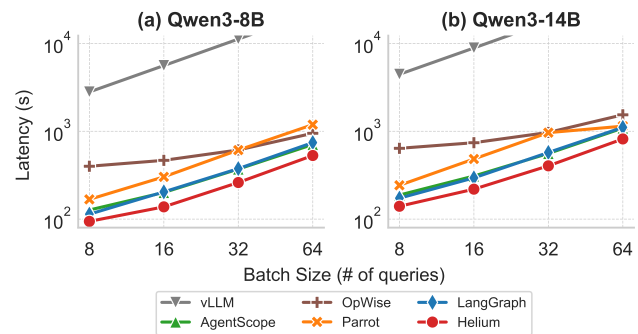

Figure 7: End-to-end latency of Helium and baselines on the *Trading* workflow across batch sizes. Left: Qwen3-8B; Right: Qwen3-14B.

Comparisons. The end-to-end latency results are summarized in Figure 7. Helium achieves a significant performance improvement over the naive *vLLM* baseline, reducing latency by up to 44.05×. This substantial gap highlights the critical need for a workflow-aware orchestration layer that can exploit parallelism. Compared with *OpWise*, which models the workflow as a DAG but executes it operator-by-operator, Helium still achieves up to a 4.57× speedup. *OpWise*'s scheduling model is limited; it cannot parallelize independent LLM requests and results in poor prefix cache utilization. Helium achieves speedups of up to 1.48× over *LangGraph* and up to 1.46× over *AgentScope*. Despite that our load balancer allows these frameworks to achieve a high degree of parallelism, they are unaware of redundant requests or shared prompt prefixes. Helium's proactive caching and cache-aware scheduling allow it to exploit these redundancies. Notably, Helium achieves speedups of up to 2.39× compared to *Parrot*. Although *Parrot* also performs workflow analysis, it is not designed for batch agentic workflows and makes suboptimal scheduling decisions. For instance, its heuristic of dispatching requests to engines based on cached prefixes leads to severe load imbalance and a significant performance penalty, highlighting the importance of Helium's cache-aware scheduling. More discussion on prefix cache utilization can be found in Appendix C.

7.3 Ablation Study

To validate the individual contributions of Helium's key components (RQ3), we perform an ablation study on the *Trading* workflow

System	Configurations	Latency (s)	Perf. delta (%)
Helium	Full	137.94	0.00
	w/o KV	145.98	-5.83
	w/o PC	156.62	-13.54
	w/o CAS	158.74	-15.08

Table 4: Ablation study comparing Helium's performance without proactive KV caching (KV), prompt caching (PC), and cache-aware scheduling (CAS) on the *Trading* workflow (batch size = 16).

Scheduling method	Latency (s)	Cache hit rate (%)
QueryWise	760.67 (4.85×)	44.4 (-19.7%)
OpWise	466.86 (2.98×)	39.5 (-28.7%)
Random	204.28 (1.30×)	49.5 (-10.6%)
LSPF	198.75 (1.27×)	50.6 (-8.5%)
Helium (w/ only CAS)	**156.89**	**55.4**

Table 5: Performance of different scheduling strategies on the *Trading* workflow (batch size=16). Caching is disabled to isolate scheduling effectiveness.

Workflow	Datasets	Configuration
MapRed	MMLU, TAT-QA	3 experts, batch sizes of 4 for MMLU and 2 for TAT-QA
Debate	MMLU, TAT-QA	3 agents, 2 debate rounds, batch sizes of 4 for MMLU and 2 for TAT-QA
Reflect	TAT-QA	1 expert, 2 critics, batch size of 2
Iterative	Amazon	2 review chunks, batch size of 2
Parallel	Amazon	2 review chunks, batch size of 2

Table 6: Small-scale workflow configurations used for the scheduling optimality experiment.

Sched. method	Total token step ($\times 10^6$)			Optimality gap (%)		
	Avg	Min	Max	Avg	Min	Max
QueryWise	21.6 ± 8.6	10.2	34.2	72.4 ± 39.3	26.7	149.2
OpWise	15.3 ± 6.9	7.0	26.4	17.6 ± 9.9	3.6	30.7
Random	15.2 ± 7.2	7.0	27.7	16.3 ± 8.1	6.0	30.3
LSPF	14.9 ± 6.9	7.0	26.9	14.5 ± 8.7	2.8	30.5
Helium	**13.4 ± 6.5**	**6.0**	**24.1**	**0.9 ± 1.4**	**0.0**	**3.6**

Table 7: Total token steps and optimality gaps of Helium's scheduling algorithm compared to the baselines.

with a batch size of 16 using Qwen3-8B. We evaluate three variants of Helium: one without proactive KV caching (w/o KV), one without prompt caching (w/o PC), and one without cache-aware scheduling (w/o CAS). The results, summarized in Table 4, show the performance delta relative to the full system. We observe that disabling cache-aware scheduling causes the largest performance drop (15.08%), emphasizing the value of intelligent task ordering for cache reuse. Removing prompt caching increases latency by 13.54%, showing that avoiding redundant operator runs is key to efficiency. Proactive KV caching has a smaller but still meaningful effect (5.83% delta), confirming that reusing precomputed KV states for static prefixes reduces overhead. Overall, these results demonstrate that Helium 's gains stem from the synergy of its caching mechanisms and cache-aware scheduling.

Scheduling Effectiveness and Optimality. Next, we study the effectiveness of Helium's cost-based cache-aware scheduling by comparing it against four well-known scheduling strategies. To isolate the impact of scheduling, we disabled proactive KV and prompt caching for this experiment. The baselines include: *QueryWise*, which executes queries sequentially (implemented using LangGraph without its batch API); *OpWise*, which executes operator-by-operator across the batch and is identical to our previous baseline; *Random*, which dispatches any ready request, reflecting the default behavior of LangGraph's batch execution; and *LSPF* [81], a state-of-the-art online prefix-aware strategy, which we implemented by modifying vLLM to sort its request queue.

The results are summarized in Table 5, showing that Helium significantly outperforms all other strategies in both latency and cache efficiency. It achieves a 4.85× and 2.98× speedup over *QueryWise* and *OpWise*, respectively. While *QueryWise* achieves a decent cache hit rate, it fails to exploit inter-query parallelism, which severely underutilizes the GPU. Conversely, *OpWise* processes queries in parallel but thrashes the prefix cache by not grouping requests with shared prefixes across different operators. The *Random* baseline, which is equivalent to our *LangGraph* baseline, improves performance over the naive strategies by maximizing parallelism. However, Helium's cost-based scheduling still achieves a 1.30× speedup by optimizing for both parallelism and cache reuse. More importantly, Helium is 1.27× faster than *LSPF*. Although *LSPF* is designed to maximize cache hits, its online, workflow-agnostic approach is not as effective as Helium's global optimization. This is reflected in the cache hit rates: *LSPF* provides only a 2% improvement over *Random*, whereas Helium improves the hit rate by 8.5% by using the TRT to make more informed scheduling decisions.

Additionally, we evaluate Helium's scheduling optimality by comparing it against the optimal schedule derived from our scheduling problem formulation (Section 5). Specifically, we reformulate the problem as a Mixed Integer Linear Program (MILP) and apply an off-the-shelf solver [14] to find a schedule that minimizes the cost function (total token step).

We evaluate Helium and the baseline scheduling strategies on 7 combinations of datasets and primitive workflows (Section 7.1). To obtain the exact schedule of each strategy, we execute the workflows and collect the trace of LLM requests' processing order. Then, we compute the total token step cost of each strategy, using the cost model described in Section 5. To allow the MILP solver to arrive at optimal, or near-optimal, solutions, we use smaller variants of the dataset-workflow configurations, as detailed in Table 6, and consider only a single LLM worker. We set the time limit for solving each workflow instance to 6 hours, which we found sufficient to arrive at near-optimal solutions.

We quantify the suboptimality of a given schedule σ using the *optimality gap*, defined as the percentage cost increase over the reference optimal schedule σ^*: Optimality gap (%) $= \frac{T(\sigma)-T(\sigma^*)}{T(\sigma^*)} \times 100\%$, where $T(\cdot)$ is the total token step. The raw token step costs and their corresponding optimality gaps are summarized in Table 7.

Overall, Helium achieves a significantly lower optimality gap compared to all baselines, with average and maximum optimality gaps of only 0.9% and 3.6%, respectively. This near-optimal performance is also reflected in its total token step cost, which is the lowest on average. Notably, *QueryWise*, unable to leverage inter-query parallelism, exhibits an extremely high optimality gap of up to 149.2%. While *OpWise* and *Random* address this problem, their non-prefix-aware nature leads to substantial optimality gaps, exceeding 30% at maximum. Although *LSPF* helps improve prefix cache reuse on average compared to *Random*, its online approach still leads to suboptimal schedules, with its optimality gap also exceeding 30%, in sharp contrast to Helium's 3.6%. This collectively demonstrates that Helium's cache-aware scheduling is essential to close the gap with the optimal schedule.

7.4 Sensitivity Analysis

To address RQ4, we conduct a sensitivity analysis to evaluate Helium's performance under various workload and system configurations. For these experiments, we compare Helium against *LangGraph*, a strong baseline in our prior experiments.

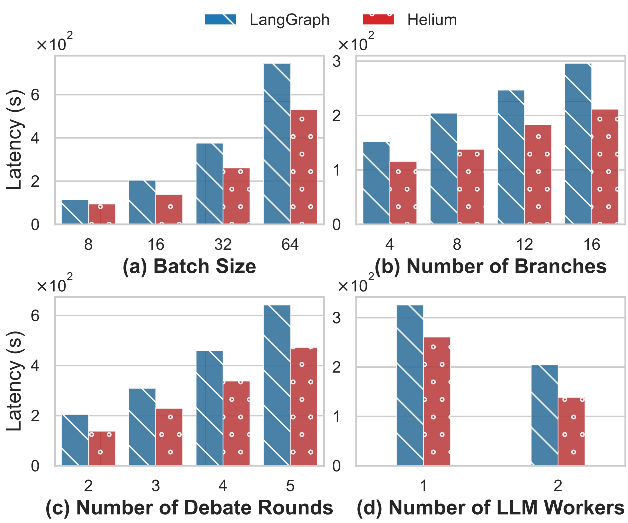

Figure 8: Helium**'s performance across different (a) batch sizes, (b) numbers of parallel branches, (c) numbers of debate rounds, and (d) numbers of LLM workers on the *Trading* workflow compared to *LangGraph*.**

Workload Scalability. We first analyze how Helium's performance scales with different workload characteristics, as shown in Figure 8. Using the *Trading* workflow with Qwen3-8B, we vary the batch size from 8 to 64 (Figure 8a). Helium's performance advantage widens with larger batches, demonstrating that its caching and scheduling strategies effectively exploit the increased inter-query sharing opportunities in larger workloads, whereas *LangGraph*'s black-box model cannot leverage such cross-query optimizations.

Next, we scale the number of parallel branches in the decision stage from 4 to 16 (Figure 8b). Helium consistently maintains its performance advantage across all configurations, efficiently handling the computational demands of highly parallel workflows. We also vary the number of debate rounds in the research stage and risk management module from 2 to 5 (Figure 8c). As the workflow becomes more sequential with additional rounds, Helium's performance advantage continues to grow due to its effective cache-aware scheduling that maximizes reuse across dependent operators. Finally, we evaluate scalability with varying numbers of LLM workers (Figure 8d). While both systems benefit from additional computational resources, Helium achieves superior scaling efficiency through its cache-aware scheduler, which optimizes operator placement to maximize cache reuse and minimize end-to-end latency.

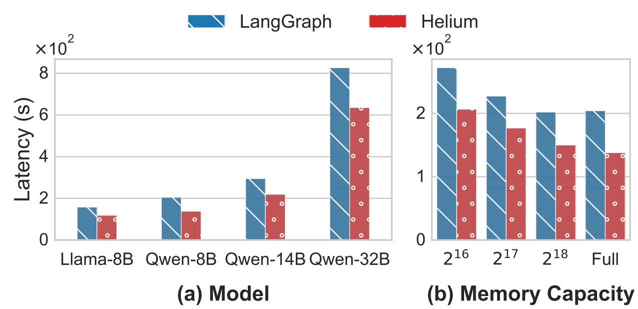

Figure 9: Helium**'s performance across different (a) LLMs and (b) memory capacities on the *Trading* workflow compared to *LangGraph*. The memory capacity is the size of the KV cache in terms of the number of tokens; Full indicates the KV cache occupies the entire GPU.**

System and Model Variations. We then assess Helium's robustness to changes in the underlying system and LLM, with results in Figure 9. We evaluate performance on the *Trading* workflow with four models: Llama-3.1-8B, Qwen3-8B, Qwen3-14B, and Qwen3-32B (Figure 9a). As expected, latency increases with model size for both systems. However, Helium's performance advantage also grows. The cost of redundant computation, which Helium is designed to remove, is greater with larger models. Helium's advantage grows because it achieves greater savings as the underlying inference becomes more resource-intensive. In addition, larger models increase memory pressure due to a larger KV cache size per token. Helium's scheduler helps with this by maximizing prefix reuse, making better use of the limited cache space.

Lastly, we simulate environments with constrained GPU memory by artificially limiting the KV cache size (Figure 9b). As memory pressure increases, both systems experience performance degradation from more frequent cache evictions. However, Helium is more resilient; its cache-aware scheduler anticipates this pressure and organizes the execution plan to make the best use of the available cache, showing better performance in resource-constrained systems.

7.5 Case Study

To provide a deeper insight into the dynamic behavior of Helium's optimization and scheduling strategies, we conduct a case study on the *Trading* workflow.

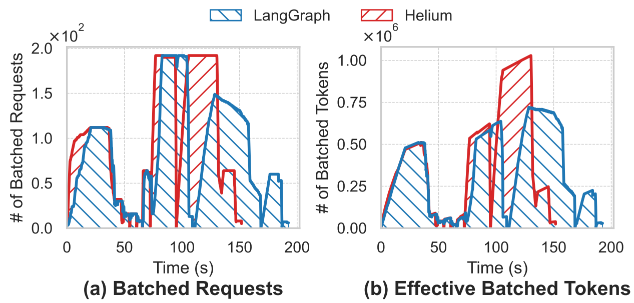

Figure 10: LLM inference batching dynamics over time for Helium **and** ***LangGraph*****, showing (a) the number of batched requests and (b) the effective number of batched tokens.**

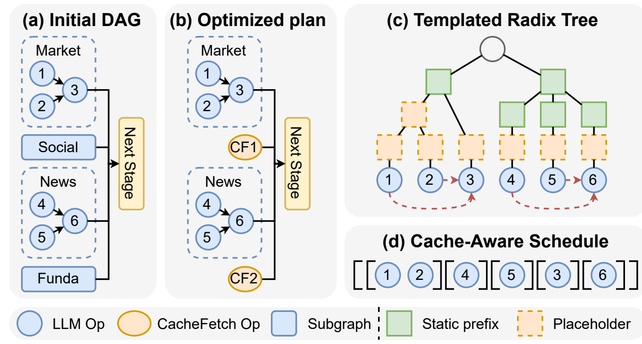

Figure 11: Helium**'s optimization and scheduling process for the** ***Trading*** **workflow: (a) initial DAG, (b) optimized logical plan, (c) TRT construction, and (d) the cache-aware schedule.**

Effective LLM Batch Sizes. First, we investigate the dynamic behavior of Helium's scheduling by monitoring the LLM inference batch sizes during execution. We run a batch of 16 queries on two LLM workers and observe two key metrics over time: the number of requests in each inference batch and the total number of tokens (including shared prefixes). To isolate the impact of scheduling on KV cache reuse, we disable proactive prompt caching for Helium and compare its performance against *LangGraph*.

The results, shown in Figure 10, demonstrate that Helium's cache-aware scheduling leads to more efficient batching. On average, Helium processes 1.20× more requests per batch compared to *LangGraph*. This improved batching directly contributes to better GPU utilization and lower end-to-end latency. Furthermore, Helium achieves a 1.43× higher peak and a 1.19× higher average number of effective batched tokens. By scheduling requests with shared prefixes consecutively on the same worker, Helium's scheduler not only maximizes KV cache reuse but also increases the effective batch size. This confirms that Helium's scheduling strategy translates cache efficiency into larger, more effective batches, further improving overall system throughput.

Optimization and Scheduling in Action. To illustrate Helium's optimization and scheduling strategies operate in practice, we trace the execution of a batch of 16 queries on the *Trading* workflow. Figure 11 visualizes this process, focusing on a simplified subgraph of the *analyst* stage for clarity. The process begins with the initial workflow DAG (Figure 11a). First, Helium's query optimizer probes the proactive prompt cache and identifies that the outputs for the fundamentals and social media agents are already cached. It rewrites the logical plan by replacing these subgraphs with `CacheFetch` operators, effectively pruning two branches from the execution graph (Figure 11b).

The query processor then receives this optimized logical plan, constructs a TRT to capture the prefix structure (Figure 11c), and applies its cache-aware scheduling algorithm. The resulting execution schedule (Figure 11d) demonstrates how the system balances parallelism and prefix cache reuse. The scheduler first groups the execution of `Op1` and `Op2` (market agent) to maximize the reuse of their shared prompt prefix across the query batch. Then, instead of idling while waiting for the dependent operator (`Op3`), the scheduler interleaves the independent operators from the news agent (`Op4` and `Op5`). This cost-based decision hides the dependency latency of `Op3` and maximizes GPU utilization. Then, `Op3` and `Op6` are scheduled once their inputs are ready. This demonstrates how Helium's holistic approach produces an efficient execution plan that is unattainable by systems that lack a global view of the workflow.

8 Related Work

LLM Inference Optimizations. Autoregressive generation makes LLM serving compute- and memory-intensive, motivating optimizations across kernels, batching, memory, system architecture, and decoding algorithms. Kernel-level work reduces attention I/O and improves GPU utilization [7, 8, 25, 48, 57, 76]. System-side batching co-schedules requests to enhance throughput while controlling latency [2, 78, 82]. KV cache management enables larger effective batches and longer contexts [27, 52]. Model and pipeline parallelism remain standard for scaling [60]. Disaggregated serving splits prefill and decode across machines to reduce interference and tailor resources [51, 83]. Speculative decoding accelerates autoregressive generation by having a lightweight draft predictor propose candidate continuations that are then verified by the full model [5, 33, 44, 59]. Beyond pure prefix caching, several techniques reuse or fuse cached states across requests [17, 73, 75] and explore non-prefix reuse or offloading [31, 73]. SGLang further organizes cached states in a radix tree to amplify prefix reuse under structured executions [81]. These advances substantially improve workflow-agnostic serving stacks; however, unlike Helium, they do not eliminate the redundancy in agentic workflows, nor do they schedule LLM calls across queries to maximize cache reuse.

LLM-based Agentic Workflows. Agentic workflows compose multiple LLM calls to solve complex tasks across domains such as software engineering, social simulation, and finance [6, 12, 16, 23, 34–36, 50, 53, 70]. Frameworks like LangChain, LangGraph, and AutoGen simplify development and orchestration but treat LLM calls as black-box units and do not optimize for system-level performance [29, 30, 67]. Recent systems introduce workflow awareness: Ayo represents workflows as primitive-level dataflow graphs to inform scheduling [63]; Parrot captures inter-call dependencies via Semantic Variables to optimize server-side execution [38]; Autellix infers workflow characteristics non-clairvoyantly to improve serving [43]; KVFlow coordinates KV cache management using workflow signals to avoid premature eviction [49]; and Halo performs cost-based batch DAG planning to expose shared computation and KV reuse [58]. These advances improve orchestration and cache utilization, but they do not aim at eliminating prompt and prefix redundancy across queries and batches, nor do they perform workflow-level, cache-aware scheduling to systematically maximize prefix reuse, which is the focus of Helium.

LLMs in Data Systems. Data systems increasingly expose LLMs through UDFs and SQL extensions for batch analytics [9, 79]. Prior work optimizes LLM-driven analytics over tables by reordering rows and fields to increase shared-prefix opportunities and reduce prefill cost [40]. Palimpzest treats LLM-centric semantics as first-class and introduces cost models to trade off time, cost, and output quality for semantic analytics [39]. These techniques target relational analytics and often involve performance–quality trade-offs. Helium differs by treating LLM invocations as first-class operators within a workflow DAG, extracting prompt structure with a templated radix tree, and using cache-aware scheduling and materialization to preserve exact semantics while amplifying cache reuse across batch agentic workflows.

9 Conclusion

This paper introduces Helium, a workflow-aware serving system that models agentic LLM workloads as query plans with LLMs as first-class operators. By combining proactive caching with cost-based, cache-aware scheduling, Helium removes redundant computation and boosts hardware efficiency. Experiments show substantial speedups over existing frameworks, demonstrating that applying query-optimization principles to LLM serving enables scalable, end-to-end efficiency for agentic AI systems.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 7, 18 pages.

[3] Zain Asgar, Michelle Nguyen, and Sachin Katti. 2025. Efficient and Scalable Agentic AI with Heterogeneous Systems. arXiv:2507.19635 [cs.LG] https://arxiv.org/abs/2507.19635

[4] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michał Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: solving elaborate problems with large language models. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24)*. AAAI Press, Article 1972, 9 pages. https://doi.org/10.1609/aaai.v38i16.29720

[5] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. MEDUSA: Simple LLM inference acceleration framework with multiple decoding heads. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 203, 27 pages.

[6] Jaroslaw A. Chudziak and Michal Wawer. 2024. ElliottAgents: A Natural Language-Driven Multi-Agent System for Stock Market Analysis and Prediction. In *Proceedings of the 38th Pacific Asia Conference on Language, Information and Computation*, Nathaniel Oco, Shirley N. Dita, Ariane Macalinga Borlongan, and Jong-Bok Kim (Eds.). Tokyo University of Foreign Studies, Tokyo, Japan, 961–970. https://aclanthology.org/2024.paclic-1.91/

[7] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=mZn2Xyh9Ec

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 16344–16359. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf

[9] Databricks. [n.d.]. Databricks: Leading Data and AI Solutions for Enterprises — databricks.com. https://www.databricks.com/. [Accessed 22-09-2025].

[10] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multi-agent debate. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*. https://arxiv.org/abs/2402.14034

[11] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 467, 31 pages.

[12] Sorouralsadat Fatemi and Yuheng Hu. 2024. FinVision: A Multi-Agent Framework for Stock Market Prediction. In *Proceedings of the 5th ACM International Conference on AI in Finance* (Brooklyn, NY, USA) *(ICAIF '24)*. Association for Computing Machinery, New York, NY, USA, 582–590. https://doi.org/10.1145/3677052.3698688

[13] Finnhub.io. [n.d.]. Finnhub - free realtime apis for stock, Forex and cryptocurrency. https://finnhub.io/

[14] John Forrest, Ted Ralphs, Stefan Vigerske, Haroldo Gambini Santos, John Forrest, Lou Hafer, Bjarni Kristjansson, jpfasano, EdwinStraver, Jan-Willem, Miles Lubin, rlougee, a-andre, jpgoncal, Samuel Brito, h-i-gassmann, Cristina, Matthew Saltzman, tosttost, Bruno Pitrus, Fumiaki Matsushima, Patrick Vossler, Ron @ Swgy, and to-st. 2024. Coin-or/CBC: Release releases/2.10.12.

[15] Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. AgentScope: A Flexible yet Robust Multi-Agent Platform. *arXiv preprint arXiv:2402.14034* (2024). https://arxiv.org/abs/2402.14034

[16] Shen Gao, Yuntao Wen, Minghang Zhu, Jianing Wei, Yuhan Cheng, Qunzi Zhang, and Shuo Shang. 2024. Simulating Financial Market via Large Language Model based Agents. arXiv:2406.19966 [cs.CL] https://arxiv.org/abs/2406.19966

[17] 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*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 325–338. https://proceedings.mlsys.org/paper_files/paper/2024/file/a66caa1703fe34705a4368c3014c1966-Paper-Conference.pdf

[18] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. https://doi.org/10.1145/152610.152611

[19] G. Graefe. 1994. Volcano-An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135. https://doi.org/10.1109/69.273032

[20] Larry Harris. 2002. Trading and exchanges.

[21] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends Databases* 1, 2 (Feb. 2007), 141–259. https://doi.org/10.1561/1900000002

[22] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *International Conference on Learning Representations*. https://openreview.net/forum?id=d7KBjmI3GmQ

[23] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VtmBAGCN7o

[24] Yupeng Hou, Jiacheng Li, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. 2024. Bridging Language and Items for Retrieval and Recommendation. *arXiv preprint arXiv:2403.03952* (2024).

[25] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. *POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference*. Association for Computing Machinery, New York, NY, USA, 897–912. https://doi.org/10.1145/3676641.3715996

[26] James E. Kelley and Morgan R. Walker. 1959. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference* (Boston, Massachusetts) *(IRE-AIEE-ACM '59 (Eastern))*. Association for Computing Machinery, New York, NY, USA, 160–173.

https://doi.org/10.1145/1460299.1460318

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[28] LangChain. [n.d.]. How to summarize text through iterative refinement | LangChain — python.langchain.com. https://python.langchain.com/docs/how_to/summarize_refine/. [Accessed 26-08-2025].

[29] LangChain. [n.d.]. LangChain: Building applications with LLMs through composable abstractions. https://python.langchain.com/docs/introduction/. Accessed: 2025-09-16.

[30] LangChain. [n.d.]. LangGraph: Graph-based orchestration for LLM workflows. https://langchain-ai.github.io/langgraph/. Accessed: 2025-09-16.

[31] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[32] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. 1977. Complexity of Machine Scheduling Problems. In *Studies in Integer Programming*, P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser (Eds.). Annals of Discrete Mathematics, Vol. 1. Elsevier, 343–362. https://doi.org/10.1016/S0167-5060(08)70743-X

[33] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 795, 13 pages.

[34] Nian Li, Chen Gao, Mingyu Li, Yong Li, and Qingmin Liao. 2024. EconAgent: Large Language Model-Empowered Agents for Simulating Macroeconomic Activities. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15523–15536. https://doi.org/10.18653/v1/2024.acl-long.829

[35] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* 1, 1 (Oct. 2024).

[36] Yuan Li, Bingqiao Luo, Qian Wang, Nuo Chen, Xu Liu, and Bingsheng He. 2024. CryptoTrade: A Reflective LLM-based Agent to Guide Zero-shot Cryptocurrency Trading. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 1094–1106. https://doi.org/10.18653/v1/2024.emnlp-main.63

[37] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 17889–17904. https://doi.org/10.18653/v1/2024.emnlp-main.992

[38] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 929–945. https://www.usenix.org/conference/osdi24/presentation/lin-chaofan

[39] Chunwei Liu, Matthew Russo, Michael J. Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael J. Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A Declarative System for Optimizing AI Workloads. *CoRR* abs/2405.14696 (2024). https://doi.org/10.48550/arXiv.2405.14696

[40] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. 2025. Optimizing LLM Queries in Relational Data Analytics Workloads. In *Eighth Conference on Machine Learning and Systems*. https://openreview.net/forum?id=R7bK9yycHp

[41] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. *arXiv preprint arXiv:2509.00997* (2025).

[42] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. 2025. Large Language Model Agent: A Survey on Methodology, Applications and Challenges. *CoRR* abs/2503.21460 (March 2025). https://doi.org/10.48550/arXiv.2503.21460

[43] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. arXiv:2502.13965 [cs.LG] https://arxiv.org/abs/2502.13965

[44] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932–949. https://doi.org/10.1145/3620666.3651335

[45] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. https://doi.org/10.1145/321479.321481

[46] NVIDIA. [n.d.]. Welcome to TENSORRT LLM's documentation! https://nvidia.github.io/TensorRT-LLM/

[47] Xuchen Pan, Dawei Gao, Yuexiang Xie, Zhewei Wei, Yaliang Li, Bolin Ding, Ji-Rong Wen, and Jingren Zhou. 2024. Very Large-Scale Multi-Agent Simulation in AgentScope. *CoRR* abs/2407.17789 (2024). https://doi.org/10.48550/arXiv.2407.17789

[48] Zaifeng Pan, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding. 2025. FastTree: Optimizing Attention Kernel and Runtime for Tree-Structured LLM Inference. In *Eighth Conference on Machine Learning and Systems*. https://openreview.net/forum?id=BwvHcHZ3kJ

[49] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. arXiv:2507.07400 [cs.DC] https://arxiv.org/abs/2507.07400

[50] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 2, 22 pages. https://doi.org/10.1145/3586183.3606763

[51] 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 *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[52] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1133–1150. https://doi.org/10.1145/3669940.3707256

[53] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15174–15186. https://doi.org/10.18653/v1/2024.acl-long.810

[54] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling.. In *SciPy*, Kathryn Huff and James Bergstra (Eds.). scipy.org, 126–132. http://dblp.uni-trier.de/db/conf/scipy/scipy2015.html#Rocklin15

[55] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2026. AI Agents vs. Agentic AI: A Conceptual taxonomy, applications and challenges. *Information Fusion* 126 (Feb. 2026), 103599. https://doi.org/10.1016/j.inffus.2025.103599

[56] Timos K. Sellis. 1988. Multiple-query optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[57] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 68658–68685. https://proceedings.neurips.cc/paper_files/paper/2024/file/7ede97c3e082c6df10a8d6103a2eebd2-Paper-Conference.pdf

[58] Junyi Shen, Noppanat Wadlom, and Yao Lu. 2025. Batch Query Processing and Optimization for Agentic Workflows. arXiv:2509.02121 [cs.DB] https://arxiv.org/abs/2509.02121

[59] Yuhao Shen, Junyi Shen, Quan Kong, Tianyu Liu, Yao Lu, and Cong Wang. 2025. Speculative Decoding via Hybrid Drafting and Rollback-Aware Branch Parallelism. arXiv:2506.01979 [cs.DC] https://arxiv.org/abs/2506.01979

[60] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https:

//arxiv.org/abs/1909.08053
[61] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. 2024. Enhancing AI Systems with Agentic Workflows Patterns in Large Language Model. In *2024 IEEE World AI IoT Congress (AIIoT)*. 527–532. https://doi.org/10.1109/AIIoT61789.2024.10578990
[62] Lawrence Stewart. [n.d.]. Winddude/reddit_finance_43_250k · datasets at hugging face. https://huggingface.co/datasets/winddude/reddit_finance_43_250k
[63] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. *Towards End-to-End Optimization of LLM-based Applications with Ayo*. Association for Computing Machinery, New York, NY, USA, 1302–1316. https://doi.org/10.1145/3676641.3716278
[64] vLLM Team. 2023. vLLM: Easy, Fast, and Cheap LLM Serving. https://github.com/vllm-project/vllm. Accessed: 2025-09-15.
[65] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18, 6 (December 2024), 186345. https://doi.org/10.1007/s11704-024-40231-1
[66] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=1PL1NIMMrw
[67] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*. https://openreview.net/forum?id=BAakY1hNKS
[68] Yunjia Xi, Jianghao Lin, Yongzhao Xiao, Zheli Zhou, Rong Shan, Te Gao, Jiachen Zhu, Weiwen Liu, Yong Yu, and Weinan Zhang. 2025. A Survey of LLM-based Deep Search Agents: Paradigm, Optimization, Evaluation, and Challenges. arXiv:2508.05668 [cs.IR] https://arxiv.org/abs/2508.05668
[69] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. 2025. The rise and potential of large language model based agents: a survey. *Sci. China Inf. Sci.* 68, 2 (2025). https://doi.org/10.1007/s11432-024-4222-0
[70] Yijia Xiao, Edward Sun, Di Luo, and Wei Wang. 2025. TradingAgents: Multi-Agents LLM Financial Trading Framework. In *The First MARW: Multi-Agent AI in the Real World Workshop at AAAI 2025*. https://openreview.net/forum?id=4QPrXwMQt1
[71] Yahoo! [n.d.]. https://finance.yahoo.com/
[72] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388
[73] 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 Twentieth European Conference on Computer Systems*. 94–109. https://doi.org/10.1145/3689031.3696098
[74] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 11809–11822. https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf
[75] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 11608–11620. https://doi.org/10.18653/v1/2024.acl-long.623
[76] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. *CoRR* abs/2501.01005 (January 2025). https://doi.org/10.48550/arXiv.2501.01005
[77] Chaojia Yu, Zihan Cheng, Hanwen Cui, Yishuo Gao, Zexu Luo, Yijin Wang, Hangbin Zheng, and Yong Zhao. 2025. A Survey on Agent Workflow – Status and Future. In *2025 8th International Conference on Artificial Intelligence and Big Data (ICAIBD)*. 770–781. https://doi.org/10.1109/ICAIBD64986.2025.11082076
[78] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu
[79] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664
[80] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. AFlow: Automating Agentic Workflow Generation. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=z5uVAKwmjf
[81] 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. 2025. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.
[82] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2025. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. arXiv:2412.03594 [cs.CL] https://arxiv.org/abs/2412.03594
[83] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin
[84] Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. TAT-QA: A Question Answering Benchmark on a Hybrid of Tabular and Textual Content in Finance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 3277–3287. https://doi.org/10.18653/v1/2021.acl-long.254

A Templated Radix Tree Construction

The templated radix tree (TRT) is built from the optimized workflow DAG to capture the shared prefix structure and data dependencies among all LLM operators. The construction process, outlined in Algorithm 2, translates the operator-level graph into the TRT's prefix-based representation.

First, we perform a topological sort of the workflow's operators to ensure that an operator's dependencies are processed before the operator itself (Line 5). The algorithm then iterates through the sorted operators (Line 6). For each LLM operator, it constructs a *prefix template*, a sequence of static tokens and dynamic placeholders that represent inputs from antecedent operators (Line 8). This template is generated by recursively traversing the operator's inputs within the DAG: static strings are appended directly, while dependencies on other operators are encoded as placeholders.

With the prefix template and its set of LLM dependencies identified (Line 8), the operator is inserted into the TRT (Line 9). The `tree.add` method implements standard radix tree insertion, traversing from the root to find the longest common prefix. If the template diverges from an existing path, the relevant node is split to create a new branch. The LLM operator is then added as a new leaf node. Its dependencies on other LLM operators are recorded as directed edges that connect it to the corresponding leaf nodes retrieved from the `node_map`. This map maintains the correspondence between operator IDs and their nodes in the TRT, ensuring that dependencies are correctly resolved as the tree is built (Line 12).

The time complexity of this procedure is determined by the initial graph traversal and the subsequent insertion of each LLM operator into the tree. Let N be the number of operators and $|E|$ be the number of dependency edges in the workflow DAG. The topological sort takes $O(N + |E|)$ time. The total work to generate all prefix templates is also bounded by $O(N+|E|)$, as it involves traversing the DAG. Let $N_{llm} \leq N$ be the number of LLM operators and L be the maximum length of a prefix template. Inserting one template takes $O(L)$ time. The total complexity is therefore $O(N+|E|+N_{llm} \cdot L)$. In a connected DAG, $|E|$ is at least $N-1$, so the complexity simplifies to $O(|E| + N_{llm} \cdot L)$.

Algorithm 2 Templated Radix Tree Construction

1: **Input:** A workflow DAG, workflow_dag; a worker assignment map, worker_assignment.
2: **Output:** A templated radix tree, tree.

3: tree ← TemplatedRadixTree()
4: node_map ← empty map from operator ID to TRT node
5: sorted_ops ← TopologicalSort(workflow_dag)
6: **for** op **in** sorted_ops **do**
7:     **if** op **is** LLM op **then**
8:         template ← GetPrefixTemplate(op)
9:         deps ← GetLLMDependencies(op, node_map)
10:         worker ← worker_assignment[op.id]
11:         node ← tree.Add(template, worker, op.id, deps)
12:         node_map[op.id] ← node
13: **return** tree

B Complexity Analysis of the Scheduling Algorithm

In this section, we provide a formal proof for the time complexity of our cache-aware scheduling algorithm, as detailed in Algorithm 1. We demonstrate that the algorithm has a polynomial time complexity in the size of the TRT and the dependency graph.

Definitions Let $T = (V, E)$ be the input TRT, where $|V|$ is the number of nodes. Let $L \subset V$ be the set of leaf nodes, which represent the LLM operators to be scheduled. The dependencies among these operators are given by the DAG $G = (L, E')$, where $|E'|$ is the number of dependency edges. Let $|V_{int}| = |V| - |L| - 1$ be the number of internal nodes in T, let $d = \text{depth}(T)$ be the maximum depth of the TRT, and let c_{max} be the maximum branching factor (i.e., number of children) of any node in T.

Proof of Complexity

Theorem B.1. *The time complexity of the scheduling algorithm (Algorithm 1) is* $O(|V_{int}| \cdot c_{max}^3 + |E'| \cdot d)$.

Proof. The algorithm consists of two primary phases: an initialization phase and a recursive scheduling phase. We analyze the complexity of each phase separately.

1. Initialization Phase: The initialization phase constructs the internal scheduling tree data structure from the input TRT and the dependency graph G. This involves three main steps:

(1) Traversing the TRT (T) to create corresponding scheduling nodes, which takes $O(|V|)$ time.
(2) Processing the dependency graph $G = (L, E')$ to compute the depth of each LLM operator (leaf node). This is equivalent to a topological sort on G, which costs $O(|L| + |E'|)$.
(3) Building the sibling dependency maps by performing a post-order traversal of the TRT. During this traversal, each dependency in E' is examined to determine if it is internal to a subtree or connects sibling subtrees. This step has a complexity of $O(|V| + |E'|)$.

Therefore, the time complexity of the initialization phase is dominated by the tree and graph traversals, resulting in $O(|V| + |E'|)$.

2. Recursive Scheduling Phase: The core of the algorithm lies in the `Schedule` function, which invokes the recursive function `Recurse`. The complexity is dominated by the work done within all calls to `Recurse`. We analyze the work done at a single internal node $u \in V_{int}$. The work at an internal node u is driven by the `while` loop that iterates until all of its children are scheduled. Let c_u be the number of children of node u. The loop runs at most c_u times. Within each iteration, the most computationally intensive operations are `SelectBestChild` and the subsequent state updates.

- **Cost of `SelectBestChild`**: This function implements our cost-based heuristic. To determine the optimal child to schedule next (based on progress and readiness), the function must analyze the inter-dependencies among the currently unscheduled sibling subtrees. This requires constructing a temporary dependency graph among the k remaining children and analyzing its structure (e.g., by finding strongly connected components to determine topological depth). Such an analysis has a complexity of at least $O(k^2)$. The total work for the `while` loop at node u is therefore the sum of this cost over its iterations:

$$\sum_{k=1}^{c_u} O(k^2) = O(c_u^3)$$

Summing this cost over all internal nodes in the tree gives the first term of our complexity: $O(|V_{int}| \cdot c_{max}^3)$.

- **Cost of State Updates**: After scheduling a child, the `UpdateState` function is called on the remaining sibling children. This propagates the effects of newly resolved dependencies. Over the entire execution of the algorithm, a single dependency edge in E' may be re-evaluated each time a scheduling decision is made that could affect its status. The maximum number of times an edge can be re-evaluated is proportional to its depth in the TRT, which is bounded by d. Therefore, the total cumulative work for all state updates across the entire algorithm is bounded by $O(|E'| \cdot d)$.

Combining the costs from the initialization and scheduling phases, the dominant factor is the recursive scheduling phase. The total time complexity is the sum of the costs of iterative child selection and cumulative state updates.

$$O(|V| + |E'|) + O(|V_{int}| \cdot c_{max}^3 + |E'| \cdot d)$$

As $|V|$ and $|E'|$ are subsumed by the terms in the scheduling phase for any non-trivial workflow, the final complexity is $O(|V_{int}| \cdot c_{max}^3 + |E'| \cdot d)$. □

C Prefix Cache Utilization

In this section, we describe additional prefix cache utilization results following our end-to-end benchmark. We evaluate Helium and the baselines on the *Trading* workflow with a batch size of 16 using the Qwen3-8B and Qwen3-14B models. The results are summarized in Table 8.

System	Cache hit rate (%)	
	Qwen3-8B	Qwen3-14B
vLLM	43.3 (-17.4%)	43.3 (-12.4%)
OpWise	39.5 (-24.6%)	41.0 (-17.1%)
LangGraph	49.5 (-5.5%)	47.3 (-4.3%)
AgentScope	40.1 (-23.4%)	37.0 (-25.1%)
Parrot	54.7 (+4.5%)	60.5 (+22.4%)
Helium	52.4	49.4

Table 8: Prefix-cache hit rates on the *Trading* workflow (batch size = 16) for Qwen3-8B and Qwen3-14B, with relative differences vs Helium.

Overall, Helium significantly improves the prefix cache hit rates over all baselines except *Parrot*. Compared to *vLLM*, which executes the workflow query by query, Helium achieves up to 17.4% higher hit rate by exploiting the prefix shared across queries in the batch, which is unattainable by *vLLM*. On the other hand, *OpWise* achieves even lower cache hit rate than *vLLM*, as the *Trading* workflow exhibits more prefix sharing across operators (e.g., system prompts shared across agents). By capitalizing on this cross-operator prefix sharing, Helium achieves up to 24.6% improvement.

Meanwhile, *LangGraph*'s hit rate matches closely with Helium, demonstrating that random scheduling can achieve reasonable cache utilization. Nonetheless, with cache-aware scheduling, Helium is able to further enhance cache hits, achieving up to 5.5% higher hit rate. In contrast, Helium achieves the highest hit rate improvement of up to 25.1% over *AgentScope*, whose agent-level parallelism, without cache-aware scheduling, severely harms cache utilization.

Notably, *Parrot* achieves up to 22.4% higher hit rate compared to Helium. This is because *Parrot*'s scheduling heuristics dispatch LLM requests sharing the same prefix to the same engine. While this strategy significantly improves cache hit rate, it leads to severe load imbalance that substantially increases end-to-end latency as shown in Section 7.2, demonstrating the importance of global optimization to maximize end-to-end performance. This experimental result collectively highlights the advantage of Helium's cost-based approach to cache-aware scheduling.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009