Teola: Towards End-to-End Optimization of LLM-based Applications

Xin Tan[1], Yimin Jiang[2], Yitao Yang[1], Hong Xu[1]
[1]The Chinese University of Hong Kong, [2]Unaffiliated

Abstract

Large language model (LLM)-based applications consist of both LLM and non-LLM components, each contributing to the end-to-end latency. Despite great efforts to optimize LLM inference, end-to-end workflow optimization has been overlooked. Existing frameworks employ coarse-grained orchestration with task modules, which confines optimizations to within each module and yields suboptimal scheduling decisions.

We propose *fine-grained end-to-end* orchestration, which utilizes *task primitives* as the basic units and represents each query's workflow as a primitive-level dataflow graph. This explicitly exposes a much larger design space, enables optimizations in parallelization and pipelining across primitives of different modules, and enhances scheduling to improve application-level performance. We build Teola, a novel orchestration framework for LLM-based applications that implements this scheme. Comprehensive experiments show that Teola can achieve up to 2.09x speedup over existing systems across various popular LLM applications.

primitive 比 module 更 fine-grained

1 Introduction

Large language models (LLMs) and their multi-modal variants have revolutionized user query understanding and content generation. This breakthrough has transformed many traditional and emerging applications. For instance, some search engines have integrated LLMs into their query processing pipelines, enhancing user experiences [3, 13]. Additionally, AI agents, a new paradigm for human-machine interaction, have led to new applications such as emotional companionship [4] and personalized assistants [16].

Despite being the most intelligent component in the applications, LLMs by themselves often cannot satisfy the diverse and complicated user requirements. Examples include knowledge timeliness and long context understanding, for which LLMs cannot perform well due to their design. If not properly handled, these problems can easily cause the well-known hallucination issue [32]. To mitigate such problems, many techniques have been proposed, including RAG (Retrieval Augmented Generation) [40, 48], external function calls [11, 33, 38] and even multiple LLM interactions. Popular frameworks such as Langchain [9] and LlamaIndex [1] support integrating various modules and building the end-to-end pipelines mentioned above.

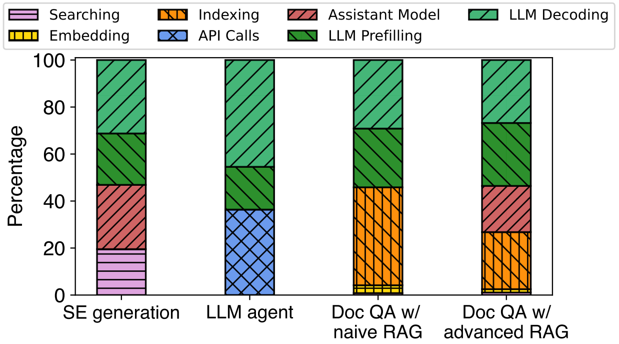
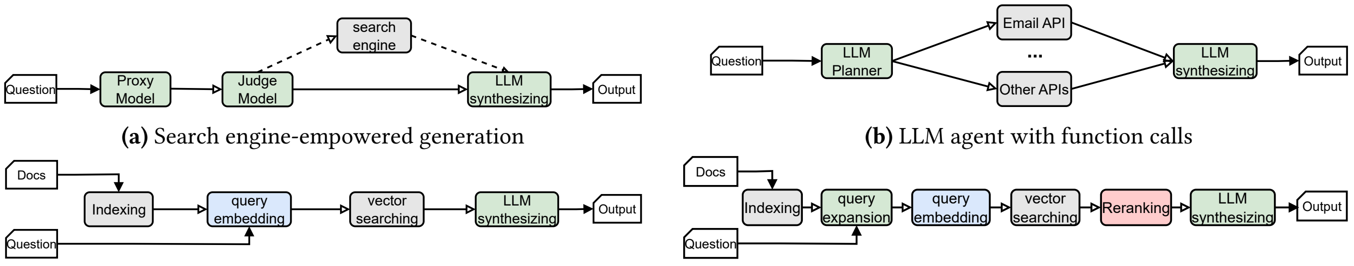


Figure 1. Latency breakdown of each task module for various applications in Figure 2 using LlamaIndex [1]. The LLM synthesizing module time is divided into prefilling and decoding.

While significant efforts have been made to optimize LLM inference across various aspects [20, 24, 39, 68, 72], little attention has been paid to the end-to-end performance of LLM-based applications composed of diverse modules. Figure 1 illustrates the execution time breakdown of several popular LLM-based applications with Llamaindex [1]. The non-LLM modules account for a significant portion of the end-to-end latency, and in some cases (Document question answering with RAG) even more than 50%. Optimizing end-to-end performance, however, faces more difficulties than one would expect in current orchestration frameworks [1, 8, 9, 12]. They organize the workflow as a simple module-based chain pipeline (see Figure 3a), where each module independently and sequentially handles a high-level task using its own execution engines (e.g. vLLM [39] for LLM inference). Despite its ease of use, this coarse-grained chaining scheme significantly limits the potential for workflow-level joint optimization across modules, as they treat each module as a black-box (§2.2). Additionally, the decoupling of frontend orchestration and backend execution implies that request scheduling cannot optimize for the application's overall performance, forcing it to instead optimizing per-request performance, which may actually degrade the overall efficiency (§2.3).

In this paper, we argue for a finer-grained exposition and orchestration of LLM-based applications, which can be the bedrock of end-to-end optimization. Rather than using the module-based chaining, we orchestrate with a primitive-level dataflow graph, where the *task primitive* serves as the basic unit. Each primitive is a symbolic node in the graph responsible for a specific primitive operation, and has a metadata profile to store its key attributes (§2.2). This primitive-level



(a) Search engine-empowered generation

(b) LLM agent with function calls

(c) Document QA with naive RAG

(d) Document QA with advanced RAG

Figure 2. Real-world LLM-based application workflows, showcasing typical definition styles in current frameworks [1, 9, 12].

graph allows us to exploit each primitive's properties and their interactions to optimize the graph, identifying an execution plan with the best end-to-end latency (§2.2). Furthermore, the graph captures request correlations and dependencies among different primitives as well as their topological depths, which enables application-aware scheduling and batching with better end-to-end performance (§2.3).

Following this insight, we build Teola, a primitive-based orchestration framework for serving LLM-based application. Teola features two main components: 1) Graph Optimizer: It parses each user query into a specific primitive-level dataflow graph, incorporating the query's input data and configurations along with the developer's pre-defined coarse-grained workflow. Subsequently, targeted optimization passes are applied to the primitive graph to generate an efficient execution graph for runtime execution. 2) Runtime Scheduler: Utilizing a two-tier scheduling mechanism, the upper tier schedules each query's execution graph, while the lower tier is managed by individual engine schedulers. The lower tier batches and processes primitives from queries' execution graphs that request the same engine, taking into account the relationships between requests from each primitive to achieve application-aware scheduling.

We implement Teola's prototype primarily using Ray [52] for distributed scheduling and execution, with various libraries for the execution engines. We evaluate Teola with diverse datasets and applications, including search engine-empowered generation and document question answering using both naive and advanced RAG. Comprehensive testbed experiments demonstrate that Teola can achieve up to a 2.09x speedup in end-to-end latency compared to existing schemes, including our distributed implementation of Llamaindex [1] with Ray and its advanced version that incorporates module parallelization and enhanced LLM execution.

Our contributions are summarized as follows:

- We identify the limitations of current LLM-based orchestration frameworks, i.e. coarse-grained module-based orchestration that restricts optimization potential, and mismatch between request-level scheduling and end-to-end application performance.
- We propose a fine-grained orchestration that represents query workflows as primitive-based dataflow graphs, enabling larger design space for end-to-end optimization including graph optimization (i.e., parallelization and pipelining) and application-aware scheduling.
- We design and implement Teola to show the feasibility and benefit of our approach. Experiments using popular LLM applications demonstrate Teola's superior performance over current systems.

2 Background and Motivation

2.1 LLM-based Applications

A primer on LLM. Current LLMs are built upon transformers, which rely on the attention mechanism to effectively capture the long context in natural languages [61]. LLM inference, which this paper focuses on, is *autoregressive*: in each forward pass the model produces a single new token—the basic unit of language modeling, which the becomes part of the context and is used as input for the subsequent iterations. To avoid redundant attention computation of preceding tokens in this process, a key-value (KV) cache is used which becomes a critical source of memory pressure [39, 68].

LLM inference involves two phases: prefilling and decoding. Prefilling produces the very first output token by processing all input tokens (instruction, context, etc.), and is clearly compute-bound. After prefilling, the decoding phase iteratively generates the rest of the output based on the KV cache, and is memory-bound as in each iteration only the new token from the previous iteration needs to be processed.

LLM apps are more than just LLM. Despite their great generation capabilities, LLMs are not a panacea. Their training datasets are inevitably not up-to-date, leading to knowledge gaps and hallucination issues [8, 40]. They also lack abilities to interact directly with the environment, that is they are not directly capable of sending an email though it can draft the email message [30, 56, 63]. Thus real-world applications often need to integrate additional tools with LLMs to be practically usable.

We show in Figure 2 four typical LLM-based applications (apps). Figure 2a demonstrates a search engine-empowered generation app, where the LLM utilizes the search engine to answer questions that are beyond its knowledge scope [3, 35, 58]. It employs a proxy and a judge model to determine if the search engine needs to be called. Figure 2b illustrates a generic LLM agent, where the LLM interacts with various tool APIs to execute the plan it formulates (e.g. draft and send emails using the user's account credentials) [30, 63]. Figures 2c and 2d showcase document question answering (QA) with naive and advanced RAG, respectively. RAG is arguably the most popular technique to enhance LLM apps with many production uses [8, 12]. Here the documents uploaded by users are ingested as chunks into the vector database as the domain knowledge base [8, 12, 40], after processed by the embedding models. This step is known as indexing. The LLM uses the relevant chunks retrieved from the vector database to generate answers. The advanced version (Figure 2d) leverages LLM-based query expansion to refine and broaden new queries [26, 34], thereby enhancing search accuracy, and subsequently reranks all retrieved chunks for the expanded queries to synthesize a precise final answer. As seen before in §1, the LLM may not be the only performance bottleneck of these complex application pipelines. Carefully orchestrating the various components of the workflow is thus critical.

2.2 Fine-grained Orchestration of LLM Apps

Many frameworks such as LlamaIndex [1], Langchain [9], and enterprise solutions such as PAI-RAG [12] and Azure-RAG [8] have emerged to facilitate the creation and orchestration of LLM applications. They naturally adopt *module-level orchestration* in the sense that each app is defined and scheduled as a simple chain of modules, as depicted in Figure 3a. Each module is executed independently with backend engines. Coarse-grained module-level chaining is easy to use, but inherently limited for optimizing the complex workflows for best performance. It overlooks the larger design space of jointly optimizing the modules, especially by exploiting the intricate dependencies among the internal operations of the individual modules.

The central thesis of this paper is to advocate for fine-grained exposition and orchestration of LLM apps in order to improve end-to-end performance. Consider an alternative representation of the same app workflow (Figure 3a) shown in Figure 3b. Instead of working with modules, we decompose each module into fine-grained *primitives* as the basic unit of orchestration (i.e. nodes in the graph). The indexing module, for example, is decomposed into embedding creation and data ingestion primitives, and query expansion is decomposed into prefilling and decoding primitives just like the LLM synthesizing module.

Moreover, the dependency among these primitives are explicitly captured in this dataflow graph, enabling the exploration of more sophisticated joint optimizations across primitives and modules. As a simple example, it is apparent that the embedding creation and data ingestion primitives

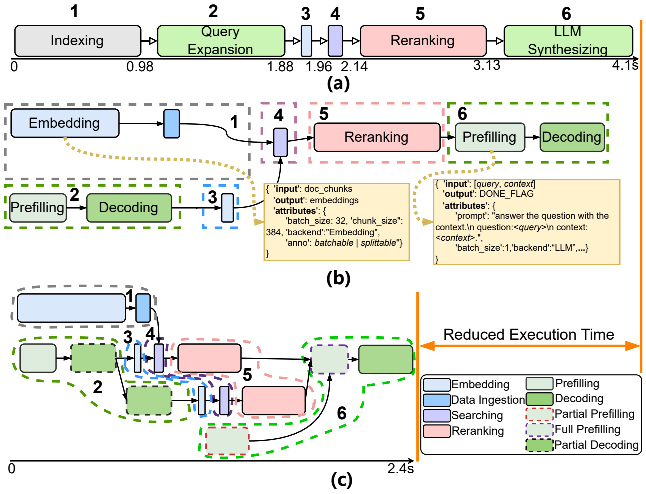


Figure 3. Workflow expression and execution comparison of existing schemes and Teola. (a) Module-level workflow in current schemes. Note the arrows here merely indicate the execution order, not the dependency. (b) Primitive-based dataflow graph in Teola (limited metadata of nodes shown in the interest of space). Arrows here represent the dependency between primitives. (c) Execution graph after optimization in Teola.

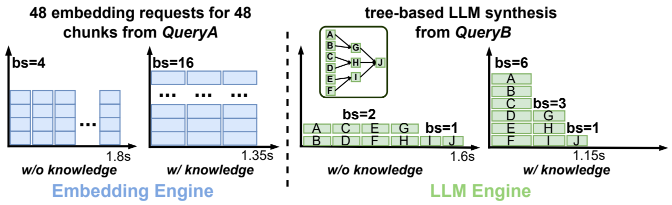
can be executed in parallel with prefilling and decoding primitives for query expansion in Figure 3b, since their inputs are independent. Each primitive's input/output relationship, along with other key information (see §2.3 for some examples), is encoded as node attributes in the graph.

We can then optimize this primitive-level dataflow graph to identify the best execution plan of the entire workflow with the lowest end-to-end latency. Figure 3c shows the optimized execution graph corresponding to the dataflow graph in Figure 3b. Specifically, given that query expansion creates multiple new queries, the corresponding decoding primitive can run in a pipeline fashion with multiple *partial decoding* primitives, each generating a new query and sending it to the subsequent primitive (embedding creation) right away without waiting for all queries to come out. By the same token, the prefilling primitive of LLM synthesizing can be divided into a partial prefilling first that operates on the system instruction and user query, which can run in parallel with indexing before search and reranking.

Thus, primitive-level dataflow graph allows us to explore various parallelization and pipelining opportunities across primitives that are not visible in existing module-based orchestration. The gain is significant: in our example (Figure 3), the overall execution time is reduced from 4.1s to 2.4s.

2.3 Application-Aware Scheduling and Execution

Another limitation of current LLM application orchestration is the request-level optimization of the backend execution



(a) Batching for embedding engine. **(b)** Batching for LLM engine.

Figure 4. Comparison between request-level and application-level scheduling and execution.

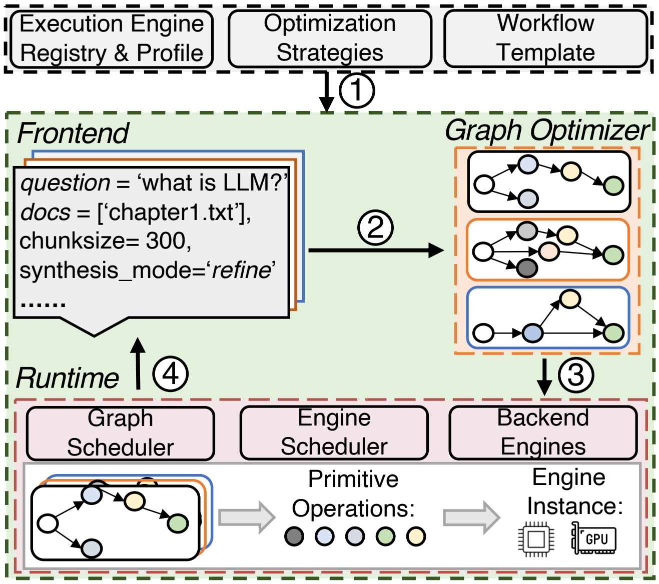


Figure 5. System Overview of Teola.

engines, which is a mismatch with the application-level performance that the user perceives. Suppose that Triton [17], a popular serving engine, is used to serve the embedding model for the indexing module. The Triton engine treats each request uniformly with a fixed batch size of 4 as shown in Figure 4a. Without any application-level information, we can only optimize for the per-request latency with reasonable but sub-optimal GPU utilization.

Now given that these embedding requests come from the same module, it is obvious that the execution engine should optimize for the total completion time instead of per-batch latency. Therefore, a better strategy is to use a larger batch size of say 16 to fully utilize the GPU. With 48 requests in total (for 48 document chunks), the total completion time is reduced from 1.8s to 1.35s, a 1.3x speedup as seen in Figure 4a, even though the per-batch latency is slightly higher.

The above toy example utilizes request correlation of an individual primitive. Another type of information we can exploit is request dependency across primitives. Consider the LLM synthesizing module, which makes a series of LLM calls in a tree-based synthesis mode in Figure 4b. These requests are executed with a batch size of 2 in conventional request-level scheduling, again to optimize per-request latency. In contrast, given that they form a dependency tree of depth of 2, the LLM execution engine can process requests at the same depth with varying batch sizes, leading to a 1.4x speedup overall albeit longer per-batch latency.

To sum up, fine-grained orchestration also bridges the gap from the execution engine's request-level optimization and enables application-aware scheduling, by using request correlation and dependency information from the primitive dataflow graph (as node attributes) to further optimize end-to-end performance.

3 Design Overview

3.1 Architecture

Teola is a novel orchestration framework to optimize the execution of LLM-based applications with primitive operations as the basic unit.

Figure 5 depicts Teola's architecture. In the offline stage ①, developers register execution engines for an app, such as those for embedding models, LLMs, and database operations, along with their latency profiles for various input sizes (e.g., batch size and sequence length). They also provide a workflow template that outlines the app's components (e.g., query expansion and LLM generation) and their execution sequence, similar to tasks modules in current frameworks [1, 9]. Optionally, developers may specify optimization strategies for certain primitive operations. Once the app is configured and deployed, the system is ready for online serving.

In the online stage, upon receiving a query with specific input data and workflow configurations, Teola creates a primitive-based dataflow graph (*p-graph*) ②, applies relevant optimizations to generate the execution graph (*e-graph*), and submits the e-graph to the runtime ③. The runtime accurately tracks and efficiently schedules the execution of the e-graph's primitives on the appropriate backends. Finally, the results are returned to the frontend upon completion ④.

3.2 APIs

Listing 1 presents a simplified usage example of Teola, highlighting its main components as described below.

Execution engines. Execution engines handle requests for models or operations from workflow components (line 5). They can be model-free or model-based. Model-free engines, such as databases, are primarily CPU-based and do not involve DNN models. On the other hand, model-based engines can deploy various DNN models, including BERT-family models [25] for embedding and LLMs for generation. A single engine can serve multiple components with different purposes, such as the shared LLM engine for query expansion and LLM synthesizing in Figure 2d.

```python
1  from Teola.app import APP,Node
2  from Teola.executor import Engine
3  from Teola.graph import OPT_Pass,Graph
4  # Register executor engines (e.g. LLM).
5  LLM=Engine("LLM",executable=llm_exec,config=config_llm,
       resource={"GPU":2},instances=2)
6  # Register optimization passes.
7  OPT_Pass.register("pipelining",pipline_pass)
8  app=APP.init() # Init an application
9  # Resiger template components with specification
10 query_expand=Node("LLM",in_kwargs,out_kwargs,
11     anno="splitable",config=config_expand)
12 embedding=Node("Embedding",in_kwargs,out_kwargs,
13     anno="batchable",config=config_embed)
14 # Omit other components...
15 generation=Node("LLM",in_kwargs,out_kwargs,
16     anno=None,config=config_gen)
17 # Declare the dependency.
18 query_expand>>embedding>>...>>generation
19 # Update the app's workflow template.
20 app.update_template([query_expand,...])
21 # Construct and optimize graph based on query and config.
22 e_graph=Graph.optimize(app,query,config,OPT_Pass)
23 # Sumbit to runtime and schedule.
24 e_graph.schedule()
```

Code Listing 1. Simplified usage example of Teola.

Workflow template. A workflow template defines the essential components for an app and their execution flow (line 8-20). Developers specify components involved in a workflow, outlining required engines, roles, and input-output configurations. These components can be further annotated with optimization hints, such as `batchable` (for batched inputs that operate independently) and `splittable` (for output that can be divided into independent partial outputs). The >> operator establishes execution sequence between components, ensuring dataflow correctness. The resulting template serves as the basis for constructing and optimizing a finer-grained graph for queries with different configurations.

Graph optimization. For each query, a finer-grained p-graph with primitive nodes is constructed based on the query-specific data, configuration, and predefined workflow template (line 22). Teola then utilizes built-in optimization passes for primitive operations and patterns to identify an optimized execution plan and generate an e-graph (§4) for execution. Developers can also register custom optimizations through a provided interface (line 7).

Declarative query. A declarative interface is offered for submitting queries to a deployed app. Beyond specifying queries (i.e., question and context), users can customize the workflow (Figure 5), allowing for parameter tuning of components (e.g., document chunk size for indexing, LLM prompt template and LLM synthesis mode) to meet performance expectations.

4 Graph Optimizer

Graph optimizer generates a fine-grained, per-query representation (*p-graph*) by combining query information and workflow template. This p-graph, composed of symbolic primitive nodes, enables optimization strategies to produce an efficient *e-graph* for execution.

Type	Description
Reranking	Compute and rank the relevance scores for the query and context pairs.
Ingestion	Store embedding vectors into vector database
Searching	Perform vector searching in the database
Embedding	Create embedding vectors for docs or questions
Prefilling	The prefilling part of LLM inference
Decoding	The decoding part of LLM inference
Partial Prefilling	Prefilling for partial prefix of a prompt (e.g. instruction, context, question)
Full Prefilling	Prefilling for rest part of a prompt after a partial prefilling
Partial Decoding	Part of full decoding for partial output
Condition	Decide the conditional branch
Aggregate	Aggregate the results from multiple primitives

Table 1. Primitive examples in Figure 2d. White backgrounds denote common operations, blue for decomposed operations, and gray for control flow operations.

4.1 p-Graph

Primitives. Relying solely on high-level components, as discussed in §2.2, can oversimplify the intricate relationships between operations and expose limited information and flexibility. To address this, we introduce a refined abstraction: the task primitive (primitive for short). Akin to the operation nodes in TensorFlow [18], symbolic primitives at the workflow level enhance granularity in representation and provide valuable information for optimization prior to execution.

Specifically, as shown in Table 1, a primitive can correspond to the functionality of a standard operation within a registered execution engine (e.g., embedding creation in embedding engines or context ranking in reranking engines) or represent a fine-grained decomposed operation. For instance, LLM inference is decomposed into `Prefilling` and `Decoding`, with `Partial Prefilling` and `Full Prefilling` constituting LLM prefilling, and `Partial Decoding` managing different parts of full decoding. Additionally, primitives can be control flow operations such as aggregation or conditional branching (i.e., `Aggregate` and `Condition`). Each primitive includes a metadata profile detailing its inputs, outputs, parent nodes, and child nodes, forming the basis for graph construction. This profile also contains key attributes such as batch size for DNNs or prompts for LLMs, as well as the target execution engine.

p-Graph construction. The optimizer converts the original workflow template $\mathcal{T} = (\mathcal{T}_N, \mathcal{T}_E)$ with query-specific configuration $C = (\mathcal{T}_N, C_N)$ into a more granular p-graph $\mathcal{G} = (\mathcal{V}_N, \mathcal{V}_E)$ as outlined in Algorithm 1, where $\mathcal{T}_N$ represents components, $\mathcal{T}_E$ dependencies, and C_N user configurations. The process decomposes each template component into explicit symbolic primitives based on the configuration, creating a sub-primitive-level graph with well-defined dependencies. For instance, the LLM synthesizing module in *refine* mode with 3 context chunks is transformed into a subgraph where 3 pairs of `Prefilling` and `Decoding` primitives are chained and configured with corresponding metadata. The final resulting p-graph preserves the original workflow

Algorithm 1 Graph transformation and optimization

```
function GraphTransform($\mathcal{T}, C$)
    $\mathcal{V}_N \leftarrow \{\}$; $\mathcal{V}_E \leftarrow \{\}$ ▷ primitives and their data dependency into
    # Decompose each template component with configuration into
    # a sub-graph with primitives and maintain sub-graph dependency
    for each $t \in \mathcal{T}_N$ do
        $Prims, Edges \leftarrow$ DecomposeComponent($t, C$)
        $Prims \leftarrow$ Configure($Prims, C$)
        $\mathcal{V}_N$.extend($Prims$); $\mathcal{V}_E$.extend($Edges$)
    # Maintain template's original component dependency
    for each $(t_i, t_j) \in \mathcal{T}_E$ do
        $tailp \leftarrow$ GetTailPrim($t_i$); $headp \leftarrow$ GetHeadPrim($t_j$)
        $\mathcal{V}_E$.append(($tailp, headp$))
    return $\mathcal{G}_p = (\mathcal{V}_N, \mathcal{V}_E)$            ▷ return primitive-level p-graph
function GraphOpt($\mathcal{G}_p$,$\mathcal{P}$)
    # $\mathcal{G}_p$ for p-graph, $\mathcal{P}$ for profile of execution engines
    $\mathcal{G}_e \leftarrow$ PrunDependency($\mathcal{G}_p$)                  ▷ Pass 1
    $\mathcal{G}_e \leftarrow$ StageDecompose($\mathcal{G}_e, \mathcal{P}$)              ▷ Pass 2
    $\mathcal{G}_e \leftarrow$ PrefillingSplit($\mathcal{G}_e$)                 ▷ Pass 3
    $\mathcal{G}_e \leftarrow$ DecodingPipeling($\mathcal{G}_e$)                ▷ Pass 4
    return $\mathcal{G}_e$                            ▷ return optimized e-graph
```

dependencies while providing a more detailed view of the workflow's inner workings.

4.2 Optimization

As mentioned in §2.2, Teola focuses on maximizing parallelism in distributed execution rather than single-point optimization or acceleration (orthogonal and discussed in §9). Specifically, the optimizer identifies opportunities for primitive parallelism (parallelization) and pipeline parallelism (pipelining), employing a set of static, rule-based optimizations.

Exploitable opportunities. Firstly, the original dependencies inherited from the workflow template, which only depict a high-level sequence of components, may introduce redundancy in the fine-grained p-graph. To maximize parallelization, it is essential to analyze and prune unnecessary dependencies, thereby freeing independent primitives and creating parallel dataflow branches (**Pass 1**). Additionally, compute-intensive primitives can be broken down into multiple pipelining stages, where feasible, enabling them to be executed concurrently with subsequent primitives (**Pass 2**).

Furthermore, we have observed that the core of the workflow, the LLM, has exploitable special attributes. Specifically, two key attributes can be leveraged: (1) *causal prefilling*: This allows the LLM's prefilling to be split into dependent parts, enabling parallelization of partial prefilling with preceding primitives (**Pass 3**), and (2) *streaming decoding output*: The auto-regressive and partial output of specific LLM decoding can be pre-communicated as input to the downstream primitives, creating additional pipelining opportunities (**Pass 4**).

Optimization passes. Based on the above analysis, the following optimization passes are integrated and can be applied to the p-graph to optimize end-to-end workflow execution:

- ▹ *Pass 1: Dependency pruning.* To increase parallelization potential, we eliminate unnecessary dependencies and identify independent dataflow branches for concurrent execution by examining each task primitive's inputs with its current upstream primitives. Redundant edges are pruned, ensuring that remaining edges represent only data dependencies, which may detach certain task primitives from the original dependency structure. For example, primitives in query expansion and embedding modules are detached to form a new branch in Figure 3c.
- ▹ *Pass 2: Stage decomposition.* For `batchable` primitives that process data exceeding the engine's maximum efficient batch size (i.e., the size beyond which throughput does not increase), they are decomposed into multiple stages, each handling a sub-micro-batch and pipelining with downstream `batchable` primitives. While more aggressive division may increase pipelining degree, finding the optimal split size is time-consuming. Moreover, adjacent batchable primitives lead to an exponential search space, which is impractical for latency-sensitive scenario. To balance resource utilization and execution efficiency, we only explicitly segment a primitive into multiple stages when its input size reaches the maximum efficient batch size. An `Aggregate` primitive is added at the end of pipelines to explicitly synchronize and aggregate the results if necessary.
- ▹ *Pass 3: LLM prefilling split.* In LLM prefilling, a full prompt consists of components such as system/user instructions, questions, and context. Within a workflow, some prompt parts may be available in advance (e.g., user instructions or questions), while others may not be (e.g., retrieved context from database in RAG). Instead of waiting for all components, available components of a prompt can be opportunistically pre-computed as they become ready, while respecting the causal attribute of attention computation, thus enabling partial prefilling parallelization.
- ▹ *Pass 4: LLM decoding pipeling.* During the decoding process of LLM, tokens are generated incrementally. Once a coherent output (e.g., a new rewritten sentence in query expansion) is available, it can be promptly forwarded to downstream `batchable` primitives, avoiding delays associated with waiting for full decoding. To enable this optimization, the LLM call must be annotated as `splittable`, indicating that its outputs can be semantically divided into distinct parts. The corresponding parser monitors the progressive, structured output (e.g., JSON) of the decoding process, extracting and forwarding complete pieces of a partial decoding to successors as soon as they become available.

Optimization procedure. The optimizer iteratively traverses the p-graph, matching primitive nodes to the pattern of each optimization pass. When a match is found, the corresponding pass is applied, and the relevant primitives are modified accordingly, as outlined in Algorithm 1. This process continues until no further optimizations are possible.

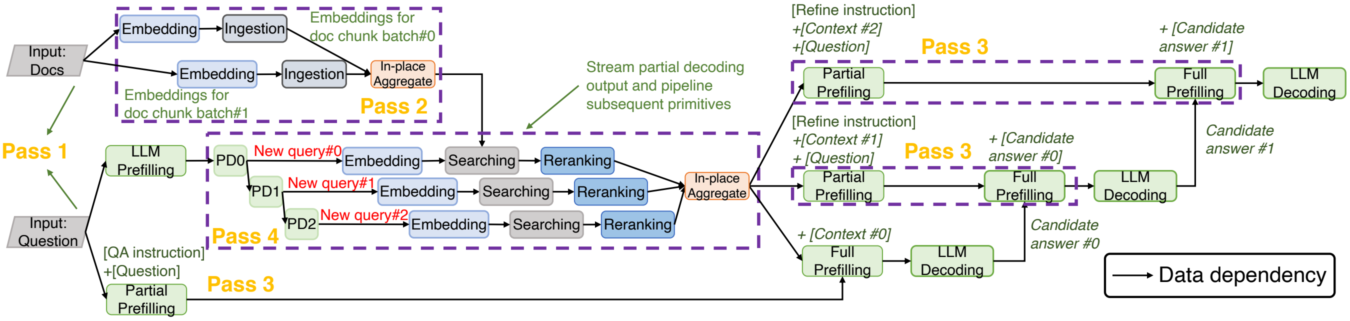


Figure 6. An illustrative optimized e-graph of a query for the advanced RAG-based document QA with a *refine* synthesis mode. (PD: partial decoding; annotated computed prompt part for Partial/Full Prefilling; primitive metadata omitted; block length not indicative of execution time.)

To reduce overhead, a cache can be employed to store and reuse the results of optimized subgraphs.

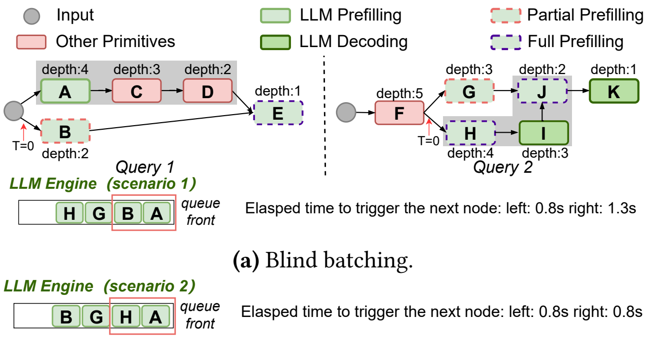
An example. Figure 6 presents an optimized e-graph for the app example shown in Figure 2d. In this optimized e-graph, query expansion module generates three new queries to enhance searching process, and the top three retrieved chunks are fed into the LLM synthesizing module. The LLM synthesizing module operates in *refine* mode, first generating an initial answer using the top chunk with a QA-style prompt template. It then refines the candidate answer twice using the remaining two chunks with a *refine*-style prompt template. The different passes applied by the optimizer are annotated in the figure for clarity.

5 Runtime Scheduling

Teola utilizes a two-tier scheduling mechanism at runtime. The upper-tier graph scheduler dispatches primitive nodes of each query's optimized e-graph. The lower tier consists of engine schedulers that manage engine instances and fuse primitive requests from queries for efficient execution. Separating graph scheduling and operation execution enhances scalability and extensibility for Teola.

5.1 Graph Scheduler

The graph scheduler closely tracks the status of each query's e-graph and issues primitive nodes as their dependencies are met. It evaluates node in-degrees and dispatches nodes to the appropriate engine scheduler when in-degrees reach zero. Note that the graph scheduler dispatches the node itself rather than its associated requests, ensuring that the lower scheduler can identify requests originating from a primitive, instead of treating them independently like in existing frameworks (see §2.3). Upon completion of a primitive's execution, the scheduling thread is notified via RPC calls, and the output is transferred. The thread then decrements the in-degrees of downstream primitives, preparing them for execution.



(a) Blind batching.

(b) Topology-aware batching.

Figure 7. An illustrative comparison of two batching schemes for an LLM instance with a maximum token size of 1024. Assume each Prefilling or Partial/Full Prefilling input contains 512 tokens, with a latency of 0.5s for 512 tokens and 0.8s for a batch of two 512-token input.

Additionally, a dedicated per-query object store manages intermediate outputs. This store acts as both an input repository for pending primitives and offers a degree of fault tolerance, safeguarding against operation failures.

5.2 Engine Scheduler

Execution engine instances are managed by dedicated engine schedulers, enabling independent execution of primitive nodes mapped to different engine types. The main challenge is efficiently fusing primitives that request the same engine. With an optimized e-graph, a query may dispatch multiple primitive nodes simultaneously to an engine scheduler or have several pending primitive nodes in the queue, especially when components share the same engine, such as the proxy and judge modules in Figure 2a or the query expansion and LLM synthesizing modules in Figure 2d using the same LLM.

Strawman solution and limitation: blind batching. A naive approach to handling diverse primitive nodes is to treat them uniformly. A engine scheduler dynamically batches associated primitive requests from the pending queue using

Algorithm 2 Topology-aware batching

```
Event 1: After getting the optimized e-graph $\mathcal{G}$ for a query:
# Determine node's depth following a reversed topological sort.
$\mathcal{G}' \leftarrow$ RevTopoSort($\mathcal{G}$); InitDepth($\mathcal{G}'$)
for $v \in \mathcal{G}'$ do
    for $p \in v.parents$ do
        $p.depth \leftarrow \max(p.depth, v.depth + 1)$
Event 2: On the scheduling period of an Engine Scheduler:
# Form the batch based on primitives' depth and relationship
$max\_bs \leftarrow$ GetConfigBatchsize(); $batch \leftarrow [\,]$
$\mathcal{B} \leftarrow$ group nodes from the same query into buckets from the queue,
    sorted by the earliest arrival time of each bucket's node.
for $b \in \mathcal{B}$ do
    $slots \leftarrow max\_bs - batch.size()$
    if $slots = 0$ then
        break
    $candidates \leftarrow$ pop associated requests (up to $slots$) from each
        node with highest depth in $b$.
    remove the nodes whose all associated requests are scheduled.
    $batch.append(candidates)$
```

a FIFO policy, reaching a predefined maximum batch size or upon timeout, similar to existing systems [17, 23]. The batch is then dispatched to an engine instance. However, this simplistic method overlooks that not all primitive nodes from the same query equally contribute to graph progression.

As illustrated in Figure 7, for query 1, primitive A and B requesting the LLM engine enter the queue along with primitive G and H from query 2. Blind batching would batch A and B, leaving G and H to wait. However, executing primitive B at this point yields little benefit since B's child E cannot be issued later due to E's other untriggered parent D. In contrast, batching A and H advances both queries' graph execution, with B's delay not bottlenecking query 1.

Our solution: topology-aware batching. The example highlights the limitations of blind batching, which ignores the unique contributions of each primitive to the query's graph progression. Primitive nodes in a graph vary in topological depth; delaying lower-depth nodes can reserve resources for more contributive ones, enhancing overall execution. Besides, it is essential to consider the correlation and dependency between requests, unlike the approach taken by existing orchestration (as discussed in §2.3). Combining these insights, we propose topology-aware batching, a heuristic solution that *leverages the depth of primitive nodes and their relationships to intelligently guide batch formation.*

Concretely, the approach offers two primary benefits. First, for an individual query, depth information naturally captures the dependency among different primitives, enabling straightforward adjustments to the scheduling preferences with the inherent request correlation in each primitive (see §2.3). For example, primitives at the same depth can be executed at the maximum efficient batch size to optimize throughput and advance the graph. Second, while depth information may not pinpoint the exact critical path due to unpredictable latency in real execution, it guides primitive prioritization for a query (see Figure 7), facilitating efficient resource utilization across multiple queries.

Algorithm 2 shows the procedure for topology-aware batching. After obtaining a query's e-graph, primitive nodes are reverse topologically sorted and their depths recorded, with the output node having the smallest depth (Event 1). When scheduling, primitives from the same query in the engine scheduler's queue are grouped into buckets. Within each bucket, primitives are sorted by depth, prioritizing those with higher depths. The buckets are then sorted based on the earliest arrival time within each bucket. Given a slot number, which represents the pre-determined maximum batch size (or maximum token size for LLM) that ensures optimal throughput efficiency, the scheduler iterates through each bucket. For each bucket, it examines the highest-priority primitive nodes and, if free slots are available, moves the associated requests from a primitive into the candidates (Event 2).

6 Implementation

We implement the prototype of Teola with ~5,300 lines of code in Python. Specifically, we leverage several existing libraries: (1) Ray [52] for distributed scheduling and execution; (2) LlamaIndex [1] for pre-processing tasks, such as text chunking and HTML/PDF parsing; (3) postgresql [15] as the default database; (4) pgvector [14] as the vector search engine; (5) Google custom search [7] as the search engine, supporting both single and batched requests; and (6) vLLM [39] as the LLM serving engine, which we additionally modify to support `Partial Prefilling` and `Full Prefilling` in Table 1.

For the frontend, we provide user interfaces via FastAPI [5] for submitting queries and user configurations. For the backend, the graph scheduler maintains a thread pool to allocate a dedicated thread for each new query, in order to construct, optimize and dispatch the e-graph. Beyond the discussion in §5, each engine scheduler also manages load balancing across different instances based on various load metrics – primarily the number of executed requests for general engines and the occupied KV cache slots for LLMs.

Mitigating communication overhead. To reduce communication overhead in a central scheduler, we use a dependent pre-scheduling mechanism for adjacent primitives with large data interactions or the same execution engine. This allows simultaneous issuance of two dependent primitives, namely A and B, with B waiting for the output of A. Along with sending A's result to the scheduler, an RPC call also sends the output of A directly to the execution engine of B. This avoids relaying results through the scheduler before issuing B, and hence can reduce the communication overhead.

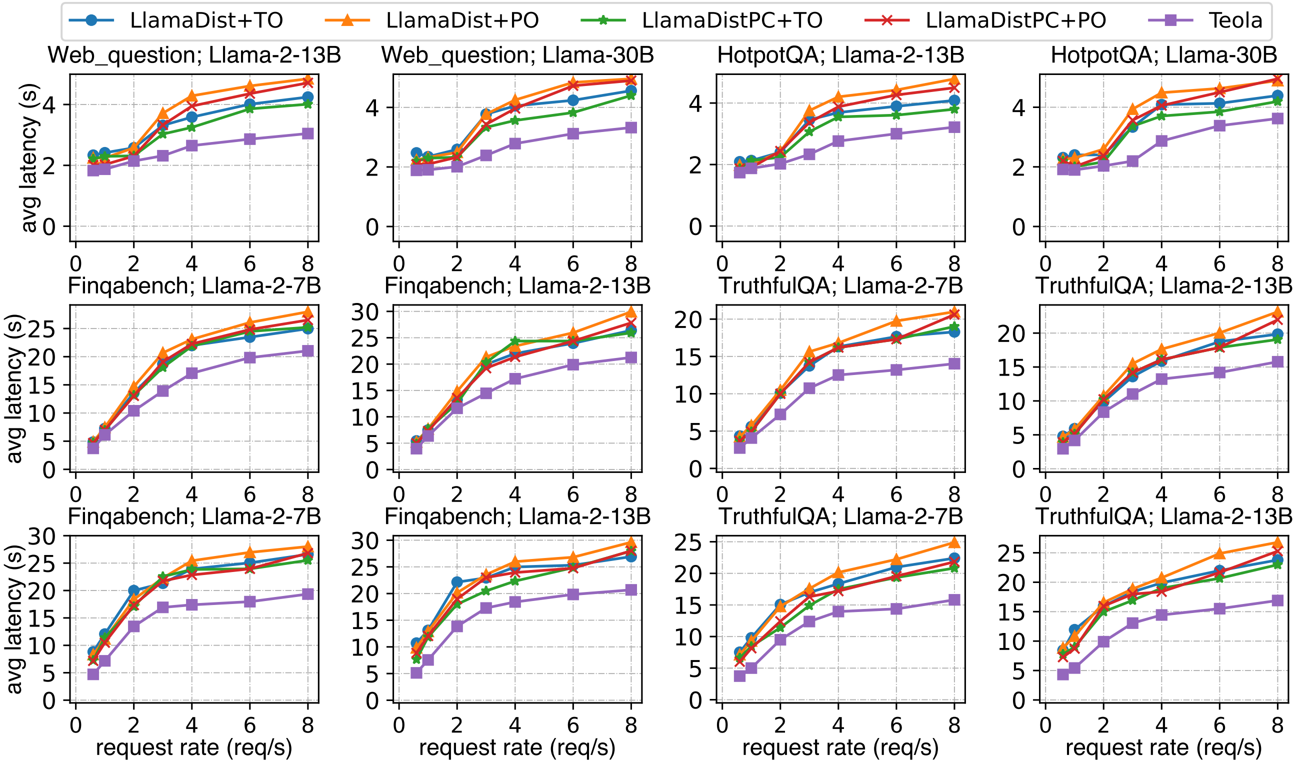


Figure 8. End-to-end performance of search engine-empowered generation (top), document QA with naive RAG (middle), and document QA with advanced RAG (bottom). Subtitles for each subfigure indicate the dataset and core LLM.

7 Evaluation

Testbed setup. We allocate model-based engines (e.g., LLMs) and model-free engines with GPUs and CPU-only resources, respectively. Each engine instance used for embeddings or other non-LLM models are each hosted on a single NVIDIA 3090 24GB GPU. For LLMs, each instance of llama-2-7B and llama-2-13B [60] is deployed on 1 and 2 NVIDIA 3090 GPUs, respectively. Each instance of llama-30B [59] is deployed on 2 NVIDIA A800 80GB GPUs. The network bandwidth between any physical servers is 100 Gbps.

Baseline. To our knowledge, few studies have specifically focused on optimizing LLM-based workflows in distributed settings. Therefore, we compare Teola with the following frameworks based on LlamaIndex [1]:

- *LlamaDist*: a distributed version of LlamaIndex that we implemented with Ray, defining a chain of task modules to construct an application pipeline. Each task module invokes requests to different distributed backend engines. This implementation integrates Ray with LlamaIndex's orchestration approach, utilizing the same engines as Teola but differing in the granularity of orchestration.
- *LlamaDistPC (parallel & cache-reuse)*: an advanced LlamaDist variant that examines the predefined pipeline and manually parallelizes independent modules for concurrent execution. It also incorporates prefix caching for LLM to avoid recomputation for partial instructions in the prompt, as proposed in some previous works [27, 45, 71].

For the request scheduling of deployed engines, we compare two approaches:

- *Per-Invocation oriented (PO)*: We slightly modify the invocation from the orchestration side and make requests in an invocation as a bundle (essentially adding extra correlation information that is exploited in Teola to enhance baselines). The engines schedule each bundle at a time, prioritizing latency preferences for each invocation.
- *Throughput oriented (TO)*: We pre-tune a maximum batch/-token size for each engine (i.e., increasing the batch size for DNNs or token size for LLMs by powers of 2 until no further throughput gain is observed) and employ dynamic batching strategy [17, 23, 39]. This maximizes the overall throughput but ignores any relationships among requests.

Applications, models and workloads. Our experiments cover three applications:

- *Search engine-empowered generation* (Figure 2a): A search engine assists a core LLM in generating answers. A smaller LLM (llama-2-7B) acts as a proxy and judge, formulating a heuristic answer and determining if a search is needed. Any search results (top 4 entities) are fed into the core LLM to synthesize the final answer. Workload requests are generated using a Poisson distribution-based synthesis of web_question [22] and HotpotQA [67] datasets.

- *Document QA with naive RAG* (Figure 2c): Users input documents or webpages along with the question. The app segments the documents into chunks (default chunk size: 256, chunk overlap:30), embeds them with the bge-large-en-v1.5 model [66], and stores them in a vector database (postgresql & pgvector). It retrieves the most relevant chunks (default: top 3) to generate a response with a *tree*-based mode. The workload (i.e. question and documents/webpage) is synthesized from Finqabench [6] and TruthfulQA [44] datasets using a Poisson distribution.
- *Document QA with advanced RAG* (Figure 2d): Extending the second app, a query expansion is used (the core LLM) to rewrite and expand the original query into multiple new queries (default: 3), improving retrieval accuracy. A reranker (bge-reranker-large [66]) evaluates the similarity between retrieved chunks, with each query searching for 16 chunks and determining the top 3 overall. These top chunks are fed into the core LLM for generation in a *refine* mode as mentioned in §4.2.

Unless otherwise specified, the above default configurations are applied. All models are deployed in half precision, and different core LLMs (7B, 13B and 30B) are experimented.

7.1 End-to-end Performance

We evaluate the performance with different schemes for various apps, all under the same resource allocation. Each non-LLM engine is provisioned with a single instance, while each LLM is provisioned with two instances.

Search engine-empowered generation. Figure 8 (top row) shows Teola outperforming the other four schemes by up to 1.79x. Teola's efficiency is attributed to parallelizable partial prefilling for instructions and questions for both the judge and core LLM, and effective batching coordination for different engines. In contrast, LlamaDist executes modules sequentially and struggles with request scheduling for multiple queries. PO's focus on per-invocation latency results in longer queue times under high request rates, while TO generally performs better in these scenarios. LlamaDistPC fails to benefit from parallelization due to the lack of explicit parallelization across modules. Its prefix caching for partial instructions (typically around 60 tokens) provides limited benefit, as prefix caching is most advantageous when prefixes are significantly longer [2, 71].

Document QA with naive RAG. Figure 8 (middle row) demonstrates that Teola outperforms the other four schemes by up to 1.62x at low request rates and 1.46x at high rates. LlamaDist executes modules sequentially without specific optimizations while LlamaDistPC enables limited parallelization (indexing and query embedding modules) and partial instruction KV cache reuse, performing slightly better than LlamaDist. Regarding scheduling, PO is better than TO at low request rates due to its focus on per-invocation latency, but performance suffers at high rates. Furthermore, the app introduces intricate request relationships. Both the indexing and query embedding modules utilize the embedding model. Meanwhile, the LLM synthesizing module makes three initial requests followed by a subsequent request to construct the *tree* synthesis. If overlooked, these can cause batching inefficiencies, leading to reduced goodput, similar to TO. Conversely, Teola leverages the e-graph, incorporating pipelining to split compute-heavy tasks like large embeddings for document chunks, while also exploring more parallelization opportunities, such as four partial prefilling. Additionally, Teola's topology-aware batching captures dependencies and correlations among requests linked to different primitives, facilitating effective batching for each engine.

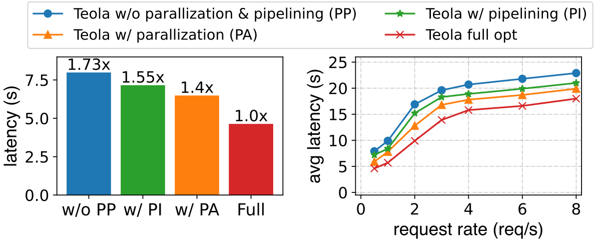


Figure 9. Ablation study on graph optimization in document QA with advanced RAG on truthfulQA dataset using llama-30B as core LLM. Left: single-query latency averaged over 10 runs. Right: average latency under varying request loads.

Document QA with advanced RAG. This app is the most complex in our settings, yet it provides ample opportunities to demonstrate the effectiveness of Teola. It leverages aggressive optimization techniques such as parallelization at different levels (e.g., independent dataflow branches and partial prefilling for different LLM calls) and pipelining (e.g., breaking large embeddings into smaller ones and splitting the decoding process in query expansion into three partial decodes), as shown in Figure 6. In contrast, LlamaDist runs sequentially with a simple run-to-completion paradigm, missing opportunities to reduce end-to-end latency. LlamaDistPC improves parallelization across the indexing and query expansion modules and reuses partial KV cache but still fails to explore the full optimization potential like Teola. Additionally, similar to naive RAG, both LlamaDist and LlamaDistPC struggle to efficiently coordinate requests whether in PO or TO, whereas Teola performs well. Overall, Teola outperforms others by up to 2.09x at low request rates and 1.68x at high request rates, as shown in Figure 8 (bottom row).

7.2 Ablation Study

We show the effectiveness of Teola's main components from graph optimization and runtime scheduling perspectives.

Graph optimization. As shown in Figure 9, we compare the performance of Teola under different scenarios, i.e., with or without parallelization (Pass 1 & 3) and pipelining (Pass 2

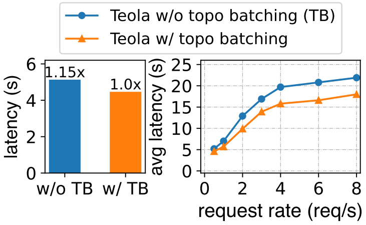


Figure 10. Ablation study on runtime scheduling. The setting is same as that in Figure 9.

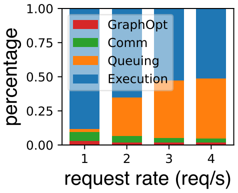


Figure 11. Latency breakdown of Teola's execution critical path.

& 4) optimization as mentioned in §4. The left figure shows the average end-to-end latency of a single query and explicitly demonstrates that both parallelization and pipelining effectively capture optimization opportunities and reduce latency. This holds true as well in a request trace with different request rates.

Runtime scheduling. Figure 10 illustrates the performance impact of enabling and disabling topology-aware batching as discussed in §5.2. The left figure demonstrates its effectiveness in capturing the varying depths of different primitives for the same engine and scheduling them with the informed correlation and dependency information to better meet the application-level performance. This results in an average 1.15x speedup for single query execution. In multi-query scenarios, topology-aware batching remains beneficial. Beyond single-query efficiency, it effectively fuses contributive primitives across queries, thereby facilitating overall execution and reducing average latency by up to 19.2%.

7.3 Overhead Analysis

To demonstrate that the overhead incurred by Teola, we provide a breakdown of latency by profiling the different parts in the real critical path of execution. This analysis covers document QA with advanced RAG on the TruthfulQA dataset at varying request rates. It includes latency measurements for graph optimization, communication, queuing, and primitive execution. The results clearly show that the graph optimization overhead is minimal (1.3% ~3%) with optimization cache reuse, and the communication overhead is low (3.1% ~6.2%) compared to the total latency. As the request rate increases, more latency is attributed to the queuing time for certain operations. These indicate that Teola incurs negligible overhead.

8 Limitations and Future Work.

Dynamic graph. While Teola's ahead-of-time graph optimization offers benefits, adapting to dynamic patterns like generation with reflection [49], iterative retrieval in RAG [48], and agent-determined workflows [38, 56] is challenging due to their unpredictable patterns and the difficulty of capturing the entire primitive-level graph prior to execution.

Coupling with the backends. To enable finer-grained orchestration, we had to modify several engine-side mechanisms, such as supporting decomposed primitive operations and certain batching strategies. These modifications required extra engineering efforts compared to existing frameworks like LlamaIndex [1] and Langchain [9] that decouple orchestration and execution and work with pluggable engines. However, these additional efforts enhance performance. Besides, at the interface level, Teola hides optimization details from users so as to be user-friendly.

Exploitation of critical path. Critical-path information in the e-graph can be further leveraged. For resource allocation, we can adjust resources for operations on critical and non-critical paths to maximize utilization based on workload patterns. For request scheduling, prioritizing critical nodes for specific queries can enhance the current topological batching, but this requires accurate online predictions of critical paths and coordination complexities.

Multi-app co-orchestration. The current design focuses on a single app but has the potential to be extended for co-orchestrating multiple apps sharing common engines within a cluster, such as RAG and LLM dialogue apps. By analyzing their distinct dataflow graphs and requirements, we can achieve broader system-wide optimizations. Optimizing the cross applications' performance is left as the future work.

9 Related Work

LLM inference optimization. LLM inference has garnered significant attention, with numerous studies focusing on various optimization directions, including kernel acceleration [24, 28, 65], request scheduling [19, 20, 57, 68], model parallelism [41, 51, 72], semantic cache [21, 73], KV cache management [39, 42, 62], KV cache reusing [27, 36, 39, 46, 71] and advanced decoding algorithms [47, 50, 53]. Recent works [31, 54, 72] disaggregate the deployment of the prefilling and decoding phases to increase goodput. This philosophy aligns well with Teola's decomposition approach and could be seamlessly integrated. While most works provide point solutions in the LLM domain, Teola takes a holistic view, optimizing the entire application workflow and facilitating cooperation among diverse components. Thus, the optimizations in LLM inference would complement Teola's efforts.

Frameworks for LLM-based applications. Apart from frameworks like [1, 2, 9], several studies [37, 38, 43, 71] focus on optimizing complex LLM tasks involving multiple LLM calls. They explore opportunities such as parallelism and sharing by developing specific programming interfaces or compilers. Moreover, several AI agent frameworks [10, 29, 30, 37, 43, 56, 63] enable LLMs to autonomously control workflows, make decisions, select tools, and interact with other LLMs, reducing the need for human intervention but introducing specific challenges. Teola is more similar to [1, 2, 9], maintaining an app of various components with human-defined flow using a primitive-level graph while focusing on end-to-end execution efficiency. Parrot [43] also captures the application-level affinity of multiple LLM requests using prompt structure to facilitate joint scheduling. Orthogonally, Teola focuses on full execution graph optimizations for applications involing both LLM and non-LLM parts.

ML analytics systems. Existing ML analytics systems, such as VideoStorm [69], Jellybean [64], Llama [55], and Vulcan [70], focus on optimizing video analytics pipelines by configuring and placing components across heterogeneous resources. While they are similar to LLM-based workflows, the latter involves more complex request patterns and flexible configurations. Besides, video systems often maintain uniform pipelines for all queries and overlook coordination between frontend orchestration and backend scheduling. Teola addresses these limitations by constructing finer-grained dataflow graph for orchestration, enhancing execution and scheduling efficiency, and leveraging LLM-specific attributes.

10 Conclusion

We present Teola, a fine-grained orchestration framework for LLM-based applications. The core idea is orchestration using primitive-level dataflow graphs. This explicitly exposes the attributes of primitive operations and their interactions, enabling natural exploration of workflow-level optimizations for parallel execution. By leveraging the primitive relationships from the graph, Teola employs a topology-aware batching heuristic to intelligently fuse requests from primitives for execution. Testbed experiments demonstrate that Teola can outperform existing schemes across different applications.

References

[1] LlamaIndex. https://github.com/jerryjliu/llama_index, 2022.

[2] Promptflow. https://https://github.com/microsoft/promptflow, 2023.

[3] Bing Copilot. https://www.bing.com/chat, 2024.

[4] Character.ai/. https://character.ai/, 2024.

[5] Fastapi. https://fastapi.tiangolo.com/, 2024.

[6] Finqabench Dataset. https://huggingface.co/datasets/lighthouzai/finqabench, 2024.

[7] Google custom search. https://programmablesearchengine.google.com/, 2024.

[8] Gpt-rag. https://github.com/Azure/GPT-RAG, 2024.

[9] Langchain. https://github.com/langchain-ai/langchain, 2024.

[10] LangGraph. https://python.langchain.com/docs/langgraph/, 2024.

[11] Openai function calling. https://platform.openai.com/docs/guides/function-calling, 2024.

[12] Pairag. https://github.com/aigc-apps/PAI-RAG, 2024.

[13] Perplexity ai. https://www.perplexity.ai/, 2024.

[14] Pgvector. https://github.com/pgvector/pgvector, 2024.

[15] Postgresql. https://www.postgresql.org/, 2024.

[16] Privatellm. https://privatellm.app/en, 2024.

[17] Triton inference server. https://github.com/triton-inference-server, 2024.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *Proc. USENIX OSDI*, 2016.

[19] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

[20] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[21] Fu Bang. Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings. In *Proc. the 3rd Workshop for Natural Language Processing Open Source Software*, 2023.

[22] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proc. EMNLP*, October 2013.

[23] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *Proc. USENIX NSDI*, 2017.

[24] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Proc. NeurIPS*, 2022.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proc. ACL*, 2018.

[26] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

[27] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934*, 2023.

[28] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Hanyu Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on gpus. In *Proc. Machine Learning and Systems*, 2023.

[29] Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zongze Xu, and Chenglin Wu. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*, 2024.

[30] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[31] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[32] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232*, 2023.

[33] Zhongzhen Huang, Kui Xue, Yongqi Fan, Linjie Mu, Ruoyu Liu, Tong Ruan, Shaoting Zhang, and Xiaofan Zhang. Tool calling: Enhancing medication consultation via retrieval-augmented large language models. *arXiv preprint arXiv:2404.17897*, 2024.

[34] Rolf Jagerman, Honglei Zhuang, Zhen Qin, Xuanhui Wang, and Michael Bendersky. Query expansion by prompting large language models. *arXiv preprint arXiv:2305.03653*, 2023.

[35] Soyeong Jeong, Jinheon Baek, Sukmin Cho, Sung Ju Hwang, and Jong C Park. Adaptive-rag: Learning to adapt retrieval-augmented large language models through question complexity. *arXiv preprint arXiv:2403.14403*, 2024.

[36] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*, 2024.

[37] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[38] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.

[39] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. ACM SOSP*, 2023.

[40] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2020.

[41] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. USENIX OSDI*, 2023.

[42] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[43] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. In *Proc. USENIX OSDI*, 2024.

[44] Stephanie Lin, Jacob Hilton, and Owain Evans. Truthfulqa: Measuring how models mimic human falsehoods. *arXiv preprint arXiv:2109.07958*,

2021.

[45] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024.

[46] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024.

[47] Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Ion Stoica, Zhijie Deng, Alvin Cheung, and Hao Zhang. Online speculative decoding. *arXiv preprint arXiv:2310.07177*, 2023.

[48] Yanming Liu, Xinyue Peng, Xuhong Zhang, Weihao Liu, Jianwei Yin, Jiannan Cao, and Tianyu Du. Ra-isf: Learning to answer and understand from retrieval augmentation via iterative self-feedback. *arXiv preprint arXiv:2403.06840*, 2024.

[49] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Proc. NeurIPS*, 2024.

[50] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proc. ACM ASPLOS*, 2024.

[51] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proc. ACM ASPLOS*, 2024.

[52] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *Proc. USENIX OSDI*, 2018.

[53] Jie Ou, Yueming Chen, and Wenhong Tian. Lossless acceleration of large language model via adaptive n-gram parallel decoding. *arXiv preprint arXiv:2404.08698*, 2024.

[54] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[55] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for autotuning video analytics pipelines. In *Proc. ACM SoCC*, 2021.

[56] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. In *Proc. NeurIPS*, 2023.

[57] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. *arXiv preprint arXiv:2401.00588*, 2023.

[58] Jiejun Tan, Zhicheng Dou, Yutao Zhu, Peidong Guo, Kun Fang, and Ji-Rong Wen. Small models, big insights: Leveraging slim proxy models to decide when and what to retrieve for llms. *arXiv preprint arXiv:2402.12052*, 2024.

[59] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Proc. NeurIPS*, 2017.

[62] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *arXiv preprint arXiv:2404.09526*, 2024.

[63] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[64] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. *arXiv preprint arXiv:2205.04713*, 2022.

[65] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *Proc. ICML*, 2023.

[66] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighof. C-pack: Packaged resources to advance general chinese embedding. *arXiv preprint arXiv:2309.07597*, 2023.

[67] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proc. EMNLP*, 2018.

[68] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *Proc. USENIX OSDI*, 2022.

[69] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and {Delay-Tolerance}. In *Proc. USENIX NSDI*, 2017.

[70] Yiwen Zhang, Xumiao Zhang, Ganesh Ananthanarayanan, Anand Iyer, Yuanchao Shu, Victor Bahl, Z Morley Mao, and Mosharaf Chowdhury. Vulcan: Automatic query planning for live ml analytics. In *Proc. USENIX NSDI*, 2024.

[71] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

[72] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

[73] Banghua Zhu, Ying Sheng, Lianmin Zheng, Clark Barrett, Michael I Jordan, and Jiantao Jiao. On optimal caching and model multiplexing for large model inference. *arXiv preprint arXiv:2306.02003*, 2023.