ASPLOS 2025

Towards End-to-End Optimization of LLM-based Applications with Ayo

Xin Tan
The Chinese University of Hong Kong
Hong Kong SAR, China
xtan22@cse.cuhk.edu.hk

Yimin Jiang
Unaffiliated
Beijing, China
jymthu@gmail.com

Yitao Yang
The Chinese University of Hong Kong
Hong Kong SAR, China
ytyang@cse.cuhk.edu.hk

Hong Xu
The Chinese University of Hong Kong
Hong Kong SAR, China
hongxu@cuhk.edu.hk

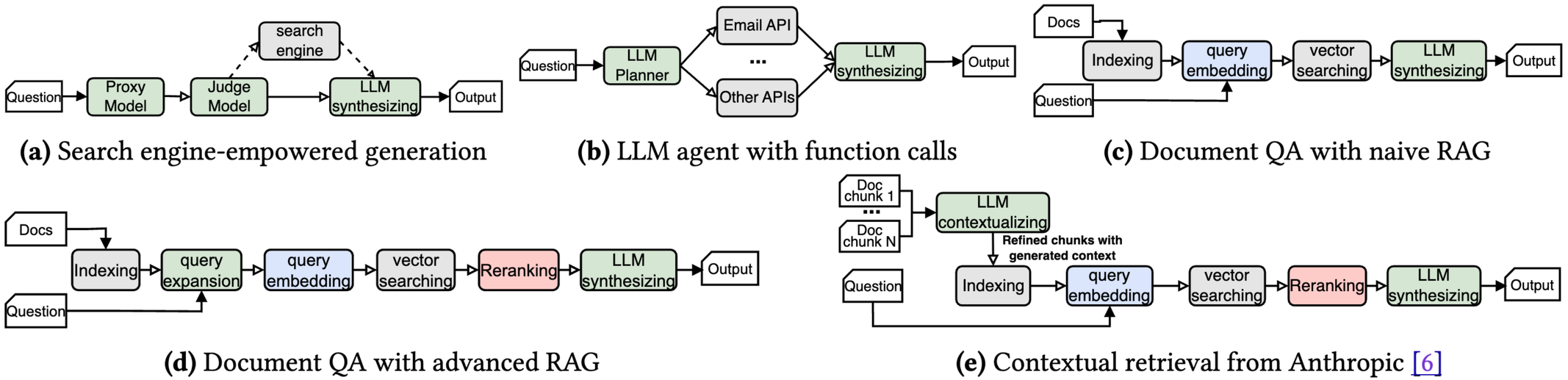
黃佳薇 / 2025.4.18

Background

LLM apps are more than just LLM.

- Training datasets not up-to-date
- knowledge gaps
- hallucination issues.
- lack abilities to interact directly with the environment

Integrate additional tools with LLMs



(a) Search engine-empowered generation **(b)** LLM agent with function calls **(c)** Document QA with naive RAG

(d) Document QA with advanced RAG **(e)** Contextual retrieval from Anthropic [6]

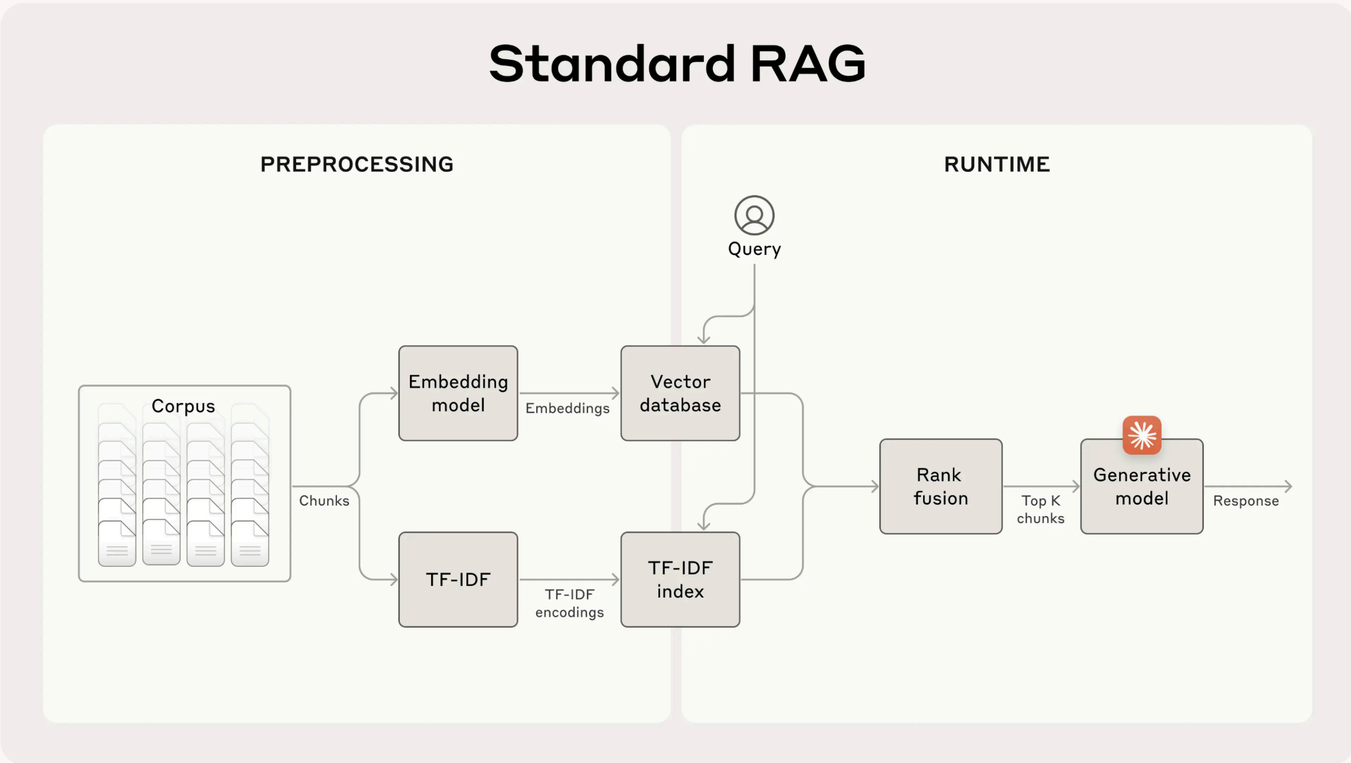
Figure 2. Common real-world LLM-based application workflows obtained from orchestration frameworks [1, 12, 17] and Anthropic [6].

Background

A primer on RAG：scaling to larger knowledge bases

Retrieval-Augmented Generation

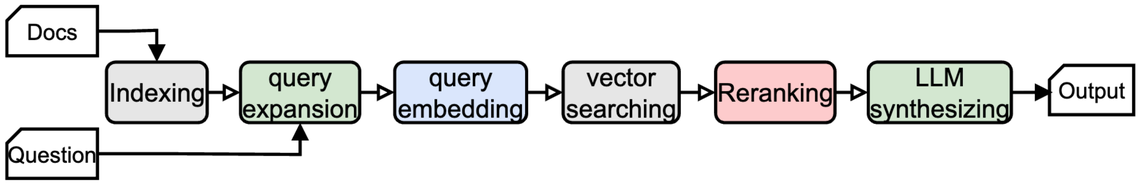
- Break down the knowledge base into smaller chunks of text
- Use an embedding model to convert these chunks into vector embeddings
- Store these embeddings in a vector database that allows for searching by semantic similarity.



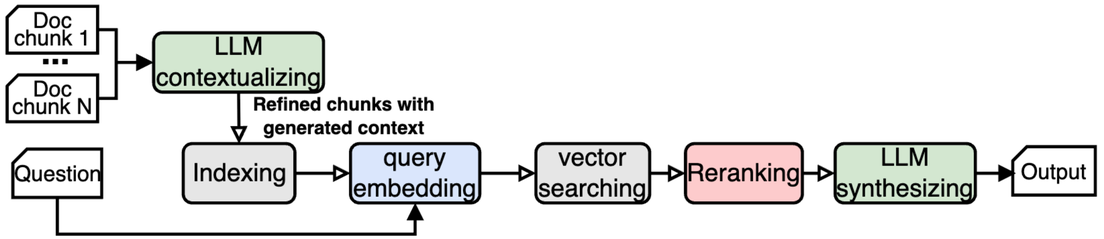
A Standard Retrieval-Augmented Generation (RAG) system that uses both embeddings and Best Match 25 (BM25) to retrieve information. TF-IDF (term frequency-inverse document frequency) measures word importance and forms the basis for BM25.

Introduction & Motivation

Observation：The non-LLM modules account for a significant portion of the end-to-end latency



(d) Document QA with advanced RAG



(e) Contextual retrieval from Anthropic [6]

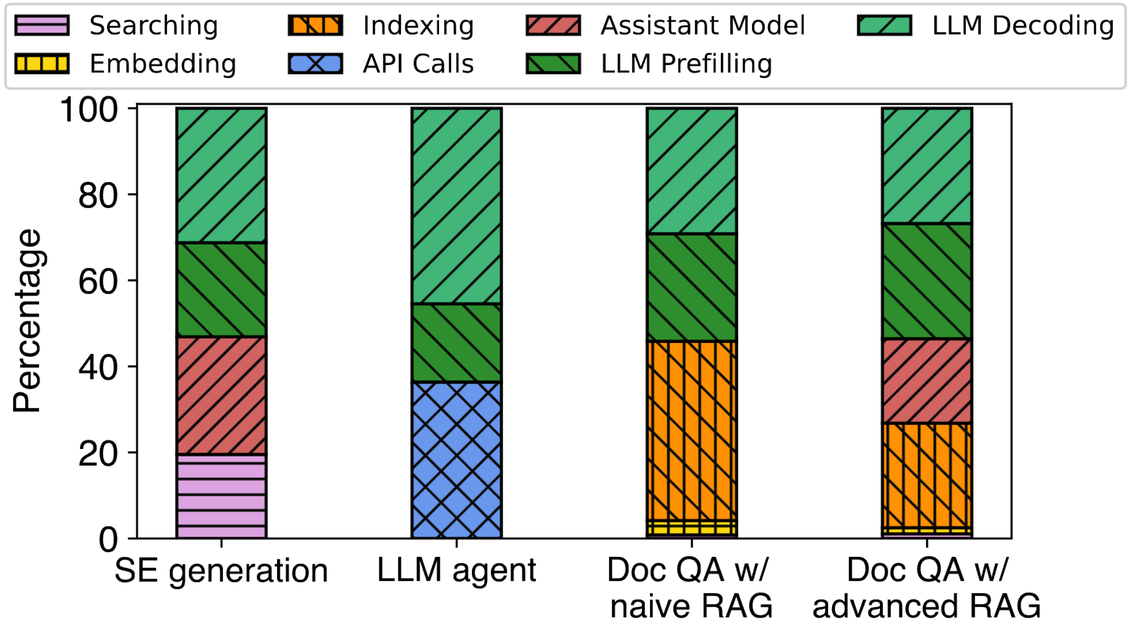


Figure 1. Latency breakdown of each task module for various applications in Figure 2 using LlamaIndex [1]. The LLM synthesizing module time is divided into prefilling and decoding.

Introduction & Motivation

Current approach has limitations in effective end-to-end performance optimization

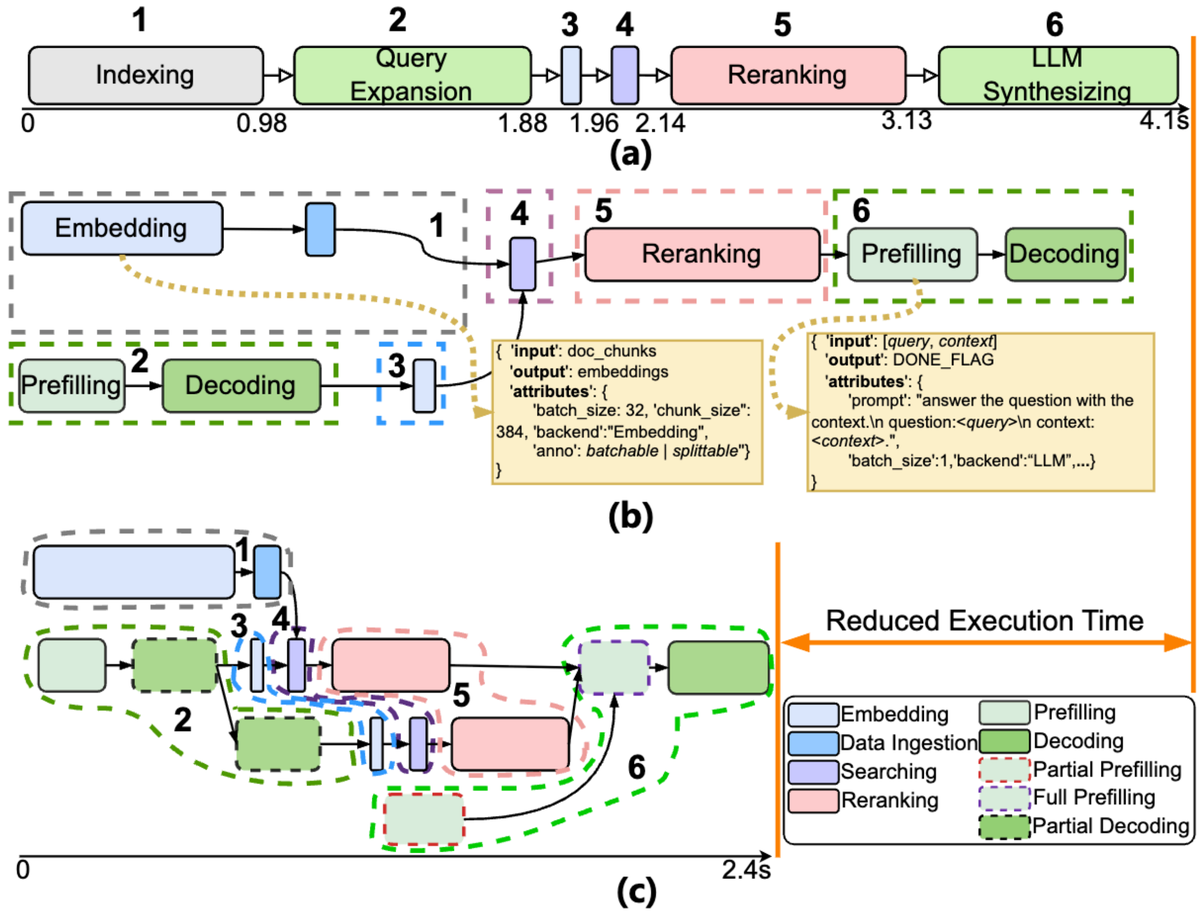
Existing solution：

organize the workflow as a simple module-based chain pipeline (such as vllm)
Figure (a)

Ayo solution：

orchestrate with a primitive-level dataflow graph.
The task primitive serves as the basic unit.
Figure (b)

allows us to explore various parallelization and pipelining opportunities across primitives that are not visible in existing module-based orchestration.
Figure (c)



Introduction & Motivation

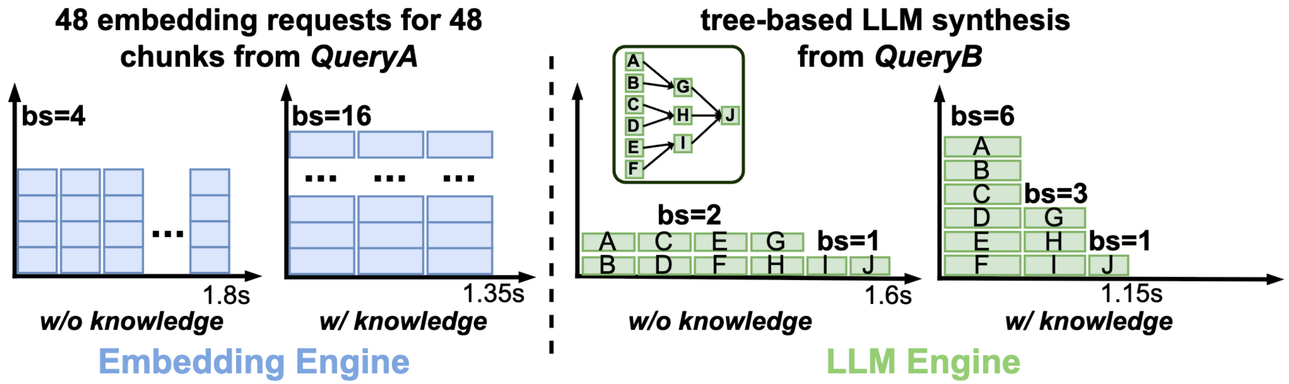
request-level optimization of backend execution engines doesn't align with the application-level performance perceived by users

Existing solution：

Without any application-level information, we can only optimize for the per-request latency

Ayo solution：

fine-grained orchestration 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.



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

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

Design Overview

Ayo：fine-grained end-to-end orchestration framework for LLM-based applications

Ayo important component:

Graph Optimizer

- parses each query into a primitive-level dataflow graph
- targeted optimization passes are applied to the primitive graph
- generate an efficient execution graph for runtime execution

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

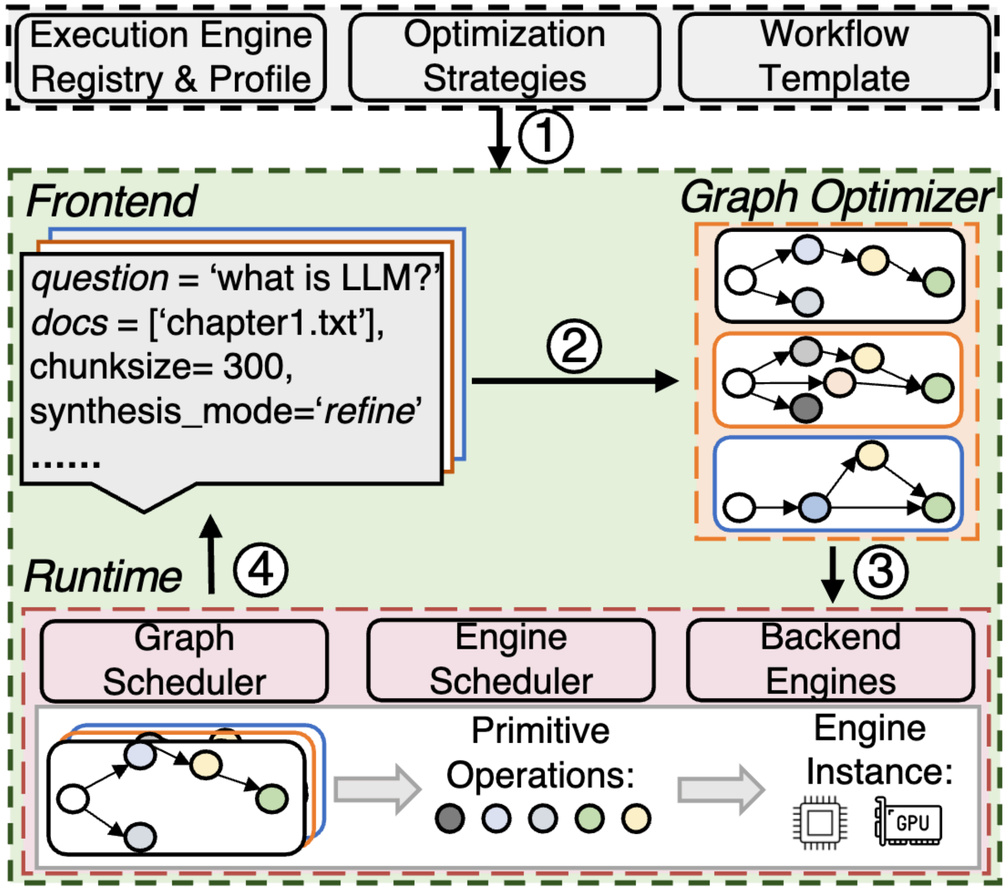


Figure 5. System Overview of Ayo.

Graph Optimizer

generates a fine-grained, per-query representation (p-graph) and optimize to produce an efficient e-graph for execution

Primitives

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

Algorithm 1 Graph transformation and optimization

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

Graph Optimizer

4 passes of optimization

Pass 1: Dependency pruning

analyze and prune unnecessary dependencies, thereby freeing independent primitives and creating parallel dataflow branches

Pass 2: Stage decomposition

For batchable primitives processing data beyond the engine's maximum efficient batch size, they are divided into stages

Pass 3: LLM prefilling split

causal prefilling: split the LLM's prefilling into dependent parts, enabling parallelization

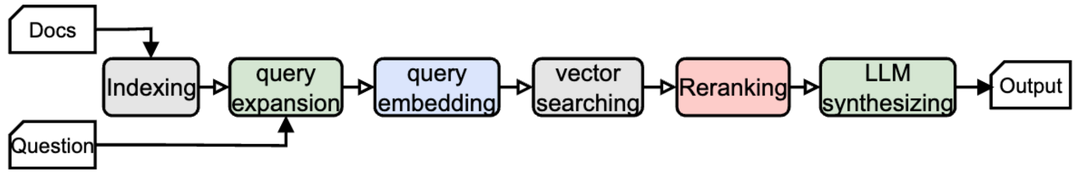
Pass 4: LLM decoding pipeling

streaming decoding output: The auto-regressive and partial output of specific LLM decoding can be pre-communicated as input to the downstream primi-tives, creating additional pipelining opportunities

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*

Graph Optimizer



(d) Document QA with advanced RAG

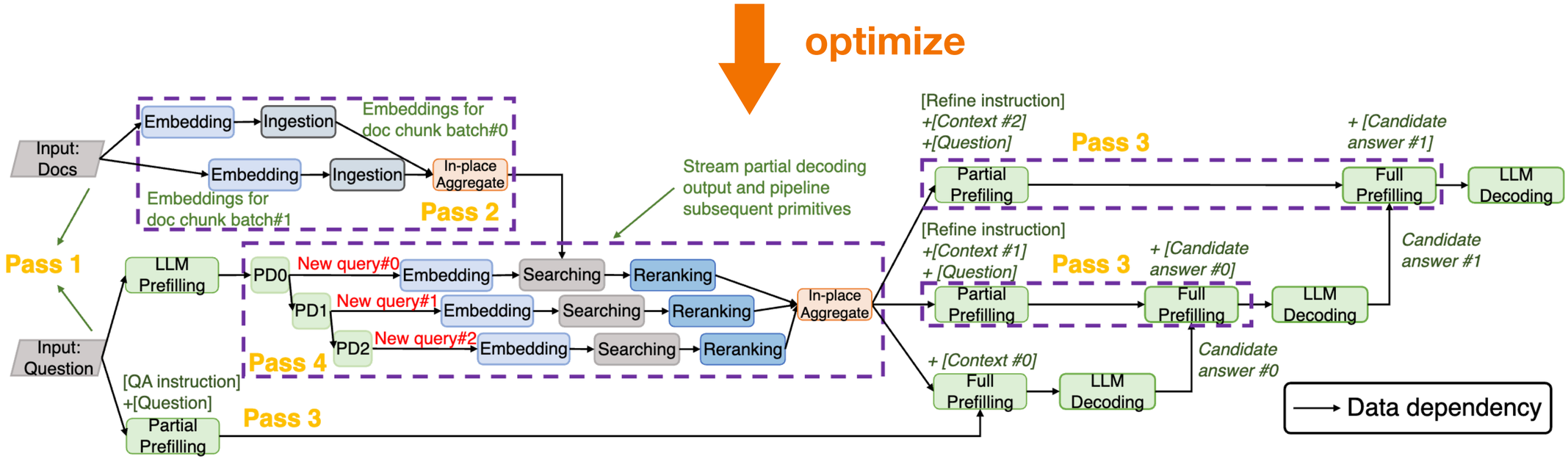


Figure 6. An illustrative optimized e-graph of a query for the advanced RAG-based document QA with a *refine* synthesis mode. (PD: partial decoding; annotated computed prompt part for Partial/Full Prefilling; primitive metadata omitted; block length not indicative of execution time.)

Runtime Scheduling

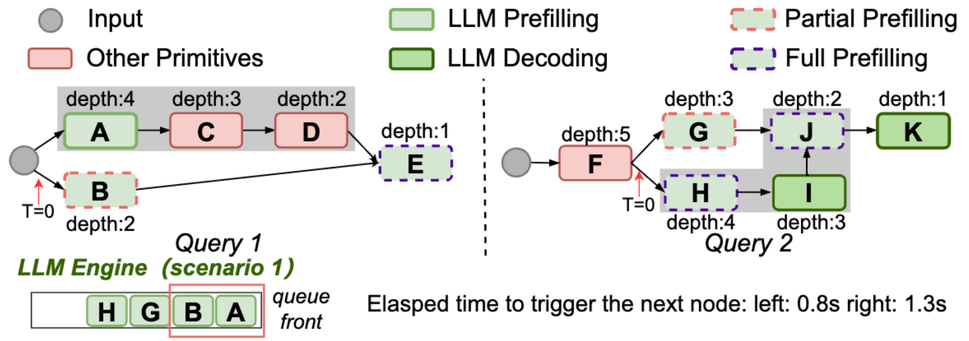
two-tier scheduling mechanism： graph scheduler and engine scheduler

Graph Scheduler

dispatches primitive nodes of each query's optimized e-graph：
closely tracks the status of each query's e-graph and issues primitive nodes as their dependencies are met.

Engine Scheduler

enabling independent execution of primitive nodes mapped to different engine types.
The main challenge is **efficiently fusing primitives that request the same engine.**



(a) Blind batching.

(b) Topology-aware batching.

Evaluation

End-to-end performance outperforms other frameworks in four typical LLM-based applications.

Baseline frameworks：

- LlamaDist
- LlamaDistPC (parallel & cache-reuse)
- AutoGen

Approaches of deployed engines

- Per-Invocation oriented (PO)
- Throughput oriented (TO)

LLM-based applications：

- search engine-empowered generation
- document QA with naive RAG
- document QA with advanced RAG
- contextual retrieval

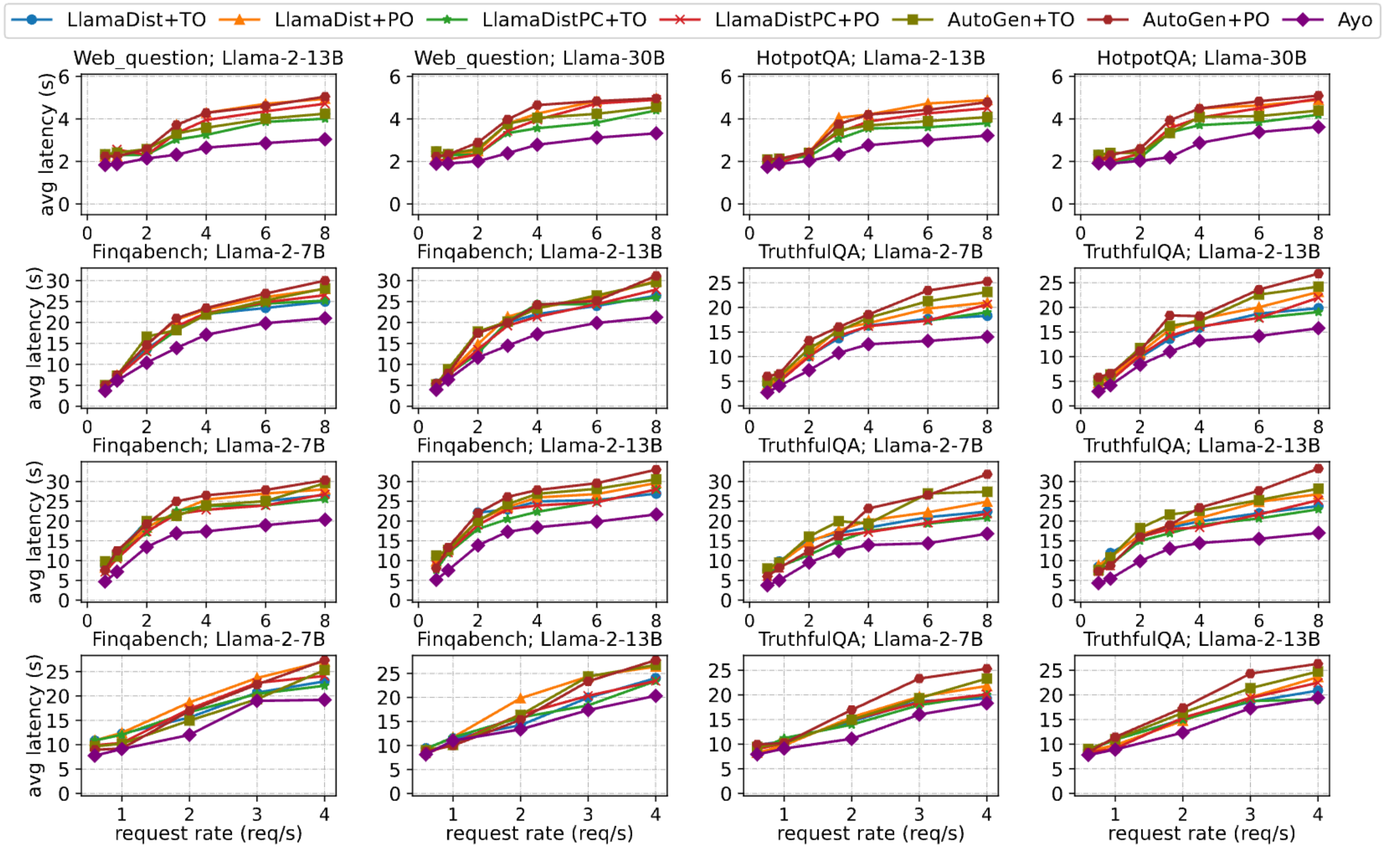


Figure 8. End-to-end performance of search engine-empowered generation (1st row), document QA with naive RAG (2nd row), document QA with advanced RAG (3rd row) and contextual retrieval (4th row). Subcaption of each subfigure indicates the dataset and core LLM.

Evaluation

Excellent Performance on co-located Applications

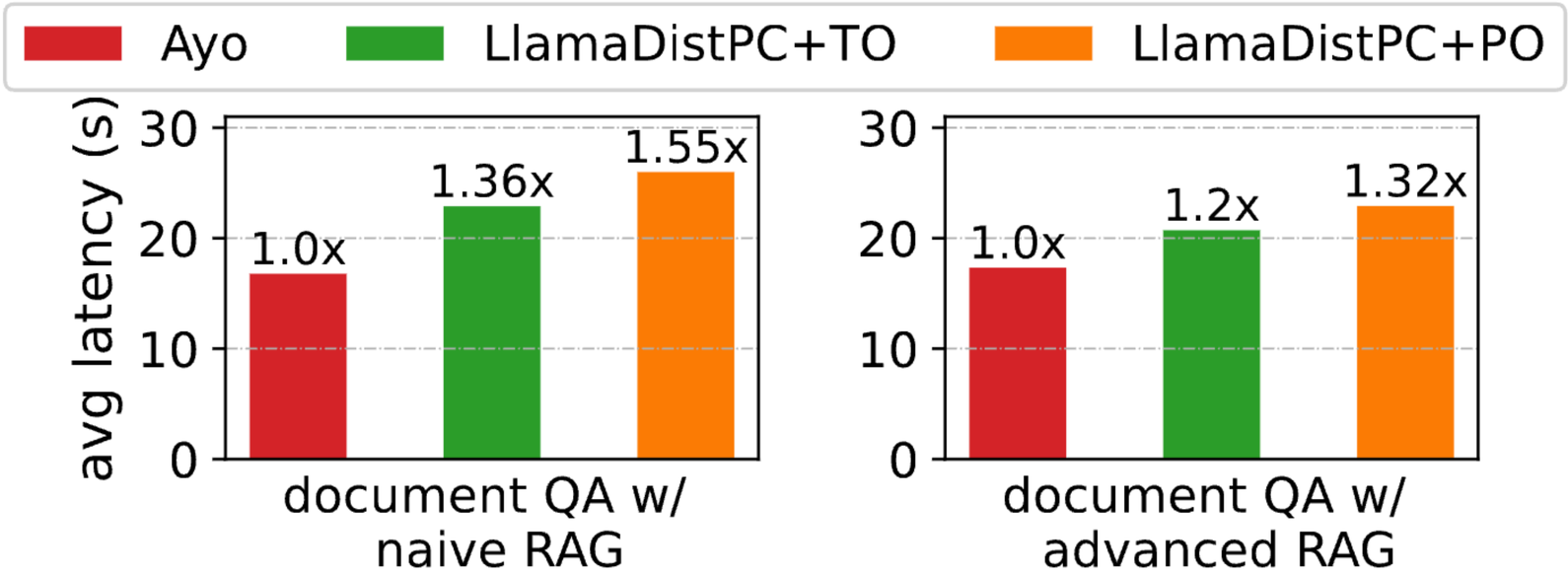


Figure 9. The average latency for two apps under a co-location scenario using llama-2-13B as the core LLM on truthfulQA dataset.

Evaluation

Ablation Study proves the effectiveness of Ayo's main components from graph optimization and runtime scheduling perspectives.

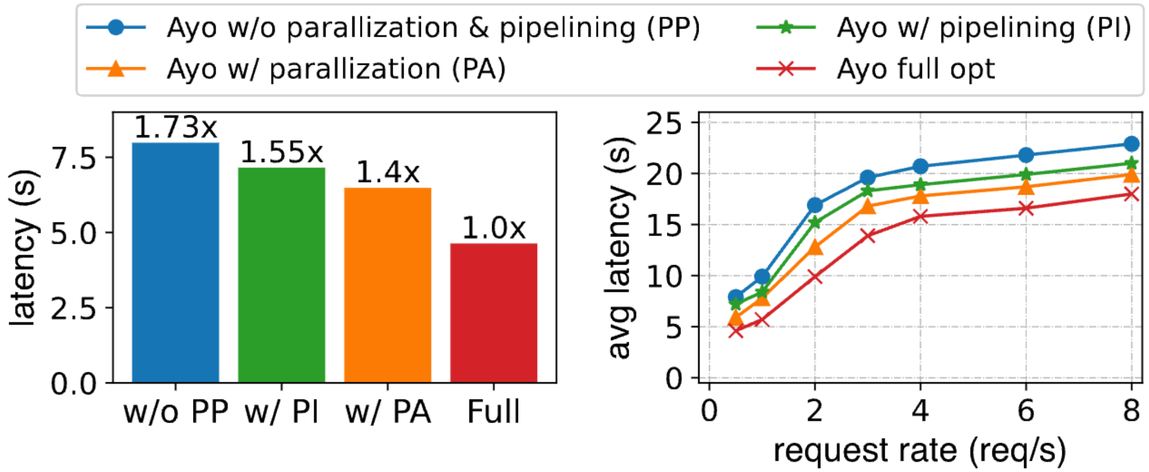


Figure 10. Ablation study on graph optimization in document QA with advanced RAG on truthfulQA dataset using llama-30B as core LLM. Left: single-query latency averaged over 10 runs. Right: average latency under varying request loads.

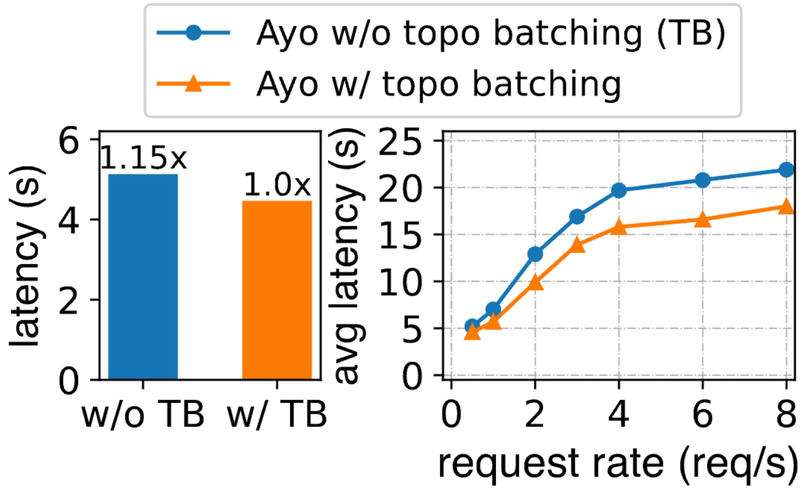


Figure 11. Ablation study on runtime scheduling. The setting is same as that in Figure 10.

Evaluation

Overhead Analysis：The overhead of e-graph optimization、communication across primitives and decomposing LLM prefilling is minimal

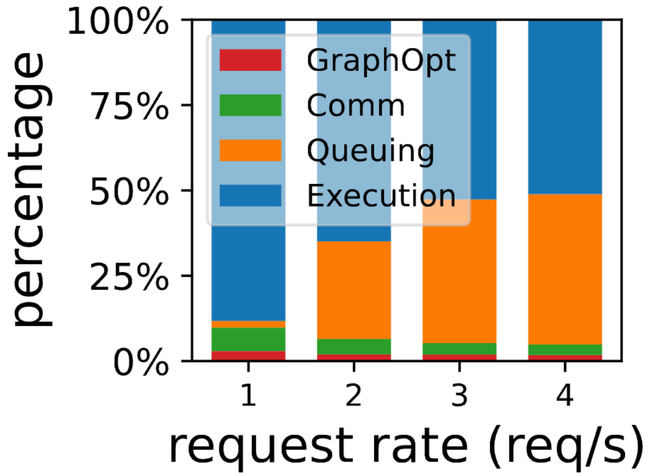


Figure 12. Latency breakdown of Ayo's execution critical path.

Partial Prefilling	Full Prefilling	Total	Single Prefilling
76.03 (200)	215.89 (800)	291.92 (1000)	260.36 (1000)
217.67 (850)	222.66 (850)	440.33 (1700)	414.09 (1700)
582.95 (2500)	159.65 (500)	742.60 (3000)	720.15 (3000)

Table 3. Execution efficiency comparison between decomposed prefilling (partial and full prefilling) of Ayo (left) vs. single complete prefilling (right) across various input sizes using Llama-2-7B. Values represent execution time in milliseconds, with input size shown in parentheses.

Conclusion

Core idea is orchestration using primitive-level dataflow graphs

- enabling workflow-level optimizations for parallel execution
- employs a topology-aware batching heuristic to intelligently fuse requests from primitives for execution.

Limitations and Future Work：

- Dynamic workflows
- Coupling with the backends
- Exploitation of critical path

Thank you!