ALTO: An Efficient Network Orchestrator for Compound AI Systems

Keshav Santhanam*
Stanford University

Deepti Raghavan*
Stanford University

Muhammad Shahir Rahman
Stanford University

Thejas Venkatesh
Stanford University

Neha Kunjal
Stanford University

Pratiksha Thaker
Carnegie Mellon University

Philip Levis
Stanford University

Matei Zaharia
University of California, Berkeley

Abstract

We present ALTO, a network orchestrator for efficiently serving compound AI systems such as pipelines of language models. ALTO leverages an optimization opportunity specific to generative language models, which is streaming intermediate outputs from the language model to downstream stages. We highlight two challenges that emerge while serving these applications at scale: handling how some stages can be stateful across partial outputs, and handling how language models can produce variable amounts of text. To address these challenges, we motivate the need for an *aggregation-aware routing* interface and *distributed prompt-aware scheduling*. ALTO's partial output streaming increases throughput by up to 3× for a fixed latency target of 4 seconds / request and reduces tail latency by 1.8× compared to a baseline serving approach, on a complex chat bot verification pipeline.

CCS Concepts: • **Computing methodologies** → **Distributed artificial intelligence**.

Keywords: Compound AI systems, Stream processing

ACM Reference Format:
Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. 2024. ALTO: An Efficient Network Orchestrator for Compound AI Systems. In *4th Workshop on Machine Learning and Systems (EuroMLSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3642970.3655844

*Both authors contributed equally to this research.

EuroMLSys '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0541-0/24/04
https://doi.org/10.1145/3642970.3655844

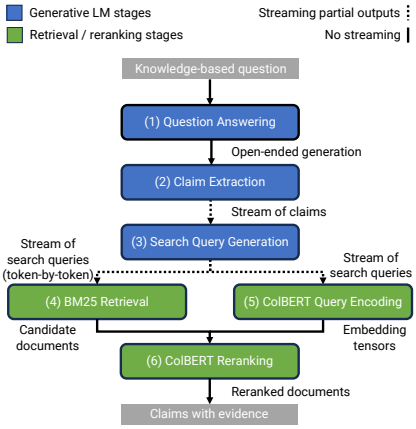


Figure 1. FacTool [8]-inspired pipeline for verifying chatbot claims. When the user asks a question (stage 1), claims are extracted from the response (stage 2) and search queries are generated for each claim (stage 3). The search queries retrieve relevant documents from a knowledge corpus using BM25 (stage 4) and reranked by ColBERT (stages 5 and 6).

1 Introduction

Generative language models (LMs) are often chained together and combined with other components into *compound AI systems* [44]. Compound AI system applications include retrieval-augmented generation (RAG) [11, 12, 17, 27], structured prompting [3, 38, 41], chatbot verification [7, 8, 10, 33], multi-hop question answering [14, 42], agents [20, 25, 29, 40], and SQL query generation [18, 34].

This paper explores how to serve compound AI systems efficiently at scale. One interesting property of generative language models (LMs) is that they incrementally produce *partial outputs*, emitting a single output token in each iteration. While language models incrementally produce tokens, stages in an AI pipeline may operate at a variety of granularities of text, ranging from individual tokens to larger quanta such as sentences or paragraphs (e.g., one stage may generate a list of claims that another stage can verify in parallel). Compound AI system pipelines with language models therefore process partial outputs at multiple levels of quantization.

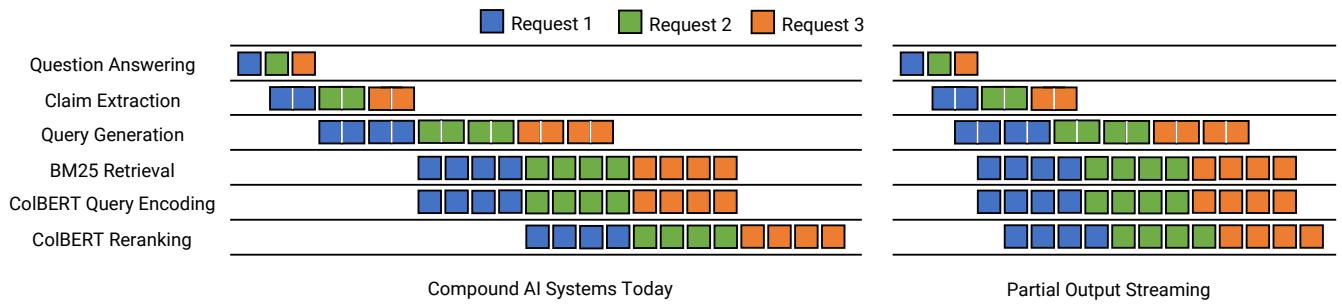


Figure 2. Visualizations of serving an example trace for the FacTool-inspired pipeline from Figure 1. The squares at each stage represent a single unit of output; in these traces two claims are extracted for each global request, and two search queries are generated for each claim. On the left we wait for all output to be produced before sending intermediate data to downstream stages, while on the right we stream partial outputs between stages as soon as it is available.

This paper makes the key observation that streaming partial outputs between distributed stages can reduce serving latency and increase throughput. Streaming reduces latency by enabling downstream stages to begin processing intermediate tokens before an upstream LM has finished generating its output. Dynamically scheduling streams across functional units increases throughput because it prevents stages from falling idle when there is work to do.

Figure 1 shows an example of this benefit for a pipeline inspired by FacTool [8][1], a compound AI system that fact-checks a chatbot by retrieving relevant documents as corroborating evidence for factual claims. Here each claim extracted in stage 2 can stream to stage 3 as soon as it is available instead of waiting for all claims to be emitted. Each search query generated in stage 3 can similarly stream to stages 4 and 5. Figure 2 shows how this reduces latency by overlapping computation across stages within a single request.

Streaming partial outputs between pipeline stages introduces two challenges: correctness and efficient load balancing. Correctness challenges emerge because some pipeline stages are stateful and aggregate partial data across a stream. In Figure 1, for example, stage 4 is stateful because it needs to aggregate document relevance scores across each search query token streamed from stage 3. Stateful stages impose a hard requirement that all partial outputs corresponding to a particular in-flight request must follow a consistent path throughout pipeline stage instances. At the same time, other stages can have their partial outputs fan out across different paths for greater parallelism. Specifying partial output routing requirements is difficult because each stage can have a dynamic fan-out spanning different quanta of output (e.g. words, sentences) with complex aggregation logic.

[1]The differences between our pipeline and FacTool are that we omit the final claim appraisal stage and we use local BM25 and ColBERT deployments for document retrieval rather than Google search. We will include the claim appraisal stage in a future version of the work.

Load balancing is challenging due to the need to decide how to route parallel requests when the LM generates an unknown amount of fan-out. In particular, each prompt served in a pipeline can generate a varying number of output tokens and be queried at different frequencies depending on this fan-out. Therefore when serving many prompts concurrently, these streams must be load balanced across many instances.

In this paper, we propose **ALTO** (**A**utomatic **L**anguage **T**oken **O**rchestrator), a serving system for automatically distributing and parallelizing compound, streaming AI pipelines. We describe a prototype of ALTO's streaming architecture and show that streaming over multiple quanta of partial outputs provides performance benefits to pipelines over naïve architectures that do not support streaming. Our current ALTO implementation addresses the challenge of correctness with *aggregation-aware routing*, an interface to express where partial outputs must be routed for aggregation. We propose possible extensions to our current design that would enable more dynamic load-aware task placement while still meeting hard requirements imposed by aggregation, describing a design for *distributed prompt-aware scheduling* to load balance across a dynamic distribution of prompts without introducing long queueing delays.

Aggregation-aware routing. ALTO introduces a novel interface to enable fine-grained specification of routing at multiple levels of output granularity through *aggregation-aware routing*. §4 shows how this interface enables developers to specify both the quantum of partial output (e.g. token, sentence) to be aggregated as well as the aggregation destination. Using this interface ALTO is able to fully load balance across logically independent partial outputs while still enforcing any specified aggregation rules.

Distributed prompt-aware scheduling. The goal of *distributed prompt-aware scheduling* is to balance load across a heterogeneous set of prompts, each producing varying quanta of partial outputs at different frequencies. We quantitatively

motivate the need for distributed prompt-aware scheduling and discuss preliminary ideas toward an algorithm design.

We evaluate ALTO on the FacTool-inspired pipeline from Figure 1. Our results show that ALTO's streaming optimizations increase throughput by 3× for a given latency target of 4 seconds / request while also reducing tail latency by 1.8×.

In summary, this paper makes the following contributions:

- An empirical analysis of how streaming partial outputs can significantly accelerate compound AI systems.
- An analysis of the novel correctness and load balancing challenges introduced when streaming partial outputs, which introduces the concepts of aggregation-aware routing and distributed prompt-aware scheduling.
- The ALTO system which implements a network orchestration layer to efficiently forward data across pipeline stage instances while respecting aggregation constraints.

2 Streaming can improve performance of serving pipelines

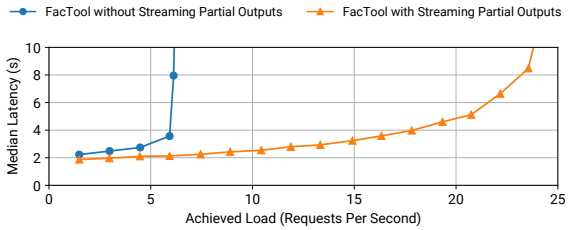
In this section we demonstrate how streaming partial outputs between pipeline stages can significantly improve compound AI system serving performance in terms of both throughput and latency. In particular, we evaluate streaming performance for the FacTool-inspired pipeline presented in Figure 1 using a prototype version of ALTO. We spawn multiple instances of each stage as specified in the following table:

Stage	# Instances	GPU
Question Answering	2	✓
Claim Extraction	2	✓
Search Query Generation	3	✓
BM25	4	✗
ColBERT Query Encoder	1	✓
ColBERT Reranker	4	✗

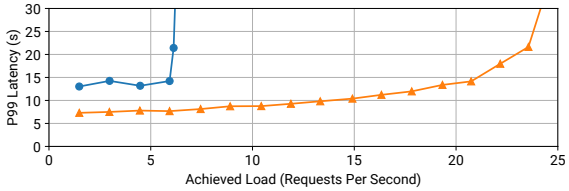
We load balance in-flight data across the instances using round-robin scheduling; we use a simple hashing-based approach to choose the next instance for forwarding in-flight data (`hash(request_id) % n`, where n is the number of downstream stage instances).

We measure end-to-end throughput (achieved load) and latency (median and P99) as we inject load into the system according to a Poisson distribution with varying λ. The achieved load here is the total number of requests sent within a time interval (in this case 12 minutes) / the wall-clock time it took to complete all requests.

We use SQuAD [26] queries for the input data. We use vLLM [16] version 0.26 for generative LM serving. For retrieval we use a retrieve-and-rerank pipeline [24] which uses a custom BM25 [28] implementation as the first stage retriever and then ColBERT [31, 32] as the reranker; the BM25 implementation is specifically designed for streaming as it exposes an interface to compute document relevance scores token-by-token and then sum across all query tokens. We evaluate on a single NVIDIA HGX node with 8 80 GB A100-SXM GPUs and 256 AMD EPYC 7763 CPUs.



(a) Median Latency vs Achieved Load.



(b) P99 Latency vs Achieved Load.

Figure 3. Serving performance for the FacTool-inspired pipeline from Figure 1. We compare performance between a baseline serving approach which waits for all LM generations to complete and an approach which instead streams partial outputs between pipeline stages. We only include points where the achieved load was ≥ 80% of the offered load. Note that both the baseline approach and the optimized approach are implemented using ALTO.

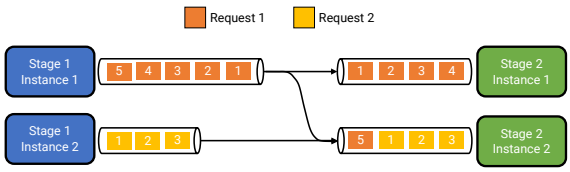
Figure 3 presents the results. We observe that streaming partial outputs enables up to 3× higher load for a given latency target of 4 seconds per request. Furthermore, streaming enables 1.8× lower P99 latency at low load. These results show that a streaming architecture is not only natural for compound AI systems but also can provide dramatic performance improvements. However, realizing these improvements for general distributed systems is not as straightforward as running an AI system on an existing streaming architecture. In the next section, we detail challenges specific to the setting of streaming in compound AI systems.

3 Challenges

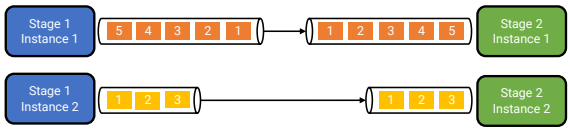
Running multiple instances of each pipeline stage results in new challenges when streaming partial outputs between each stage. In particular, this requires special considerations for correctness and load balancing that are unique to LM applications.

3.1 Correctness

Compound AI systems can include stateful pipeline stages which aggregate across partial outputs within a stream. As



(a) Load balancing without an aggregation constraint.



(b) Load balancing with an aggregation constraint.

Figure 4. Aggregation complicates load balancing. Stage 1 instances stream partial outputs to stage 2 instances; each numbered box represents a single partial output. Output streams that do not require aggregation (Figure 4a) can be easily load balanced across instances. An aggregation-aware routing policy (Figure 4b) that requires all outputs from each request to visit the same instance, however, forces different instances to have different loads; instance 2 falls idle after processing its three outputs while instance 1 continues processing the 4th and 5th outputs.

we discuss in §1, stage 4 in Figure 1 is stateful because it needs to sum across per-token relevance scores when streaming search query tokens from stage 3. In addition to sums, stateful stages may include aggregation operators such as top-k, counters, and filters.

Aggregation-aware routing is necessary to ensure correct aggregation for stateful stages while load balancing partial outputs across multiple stage instances. With aggregation-aware routing, every partial output in a stream of partial outputs is routed through the same destination stage instance. The experiments discussed in §2 use a simple hashing mechanism to implement aggregation-aware routing. This approach is suboptimal, however, because it unnecessarily forces every stage to respect a global aggregation-aware routing policy even when the stage performs no aggregation. As Figure 4 shows, this can compromise load balancing efficiency by limiting routing flexibility. The optimal approach would instead locally apply aggregation-aware routing exclusively to stateful stages.

Restricting aggregation-aware routing to stateful stages requires designing a new interface for specifying the stateful stages and their respective aggregation rules to the underlying routing engine. This is challenging because the interface must generalize across the space of possible output quanta while capturing complex aggregation logic. Consider the aggregation-aware routing rule for partial outputs streamed from stage 3 to stage 4 in Figure 1. A complete specification of this rule must indicate that the partial output quantum to aggregate is a token, the tokens must be aggregated at stage 4, and the tokens should be aggregated across a given search query.

Streaming APIs such as those defined in Kafka [36], Spark Streaming [2, 43], Naiad [22], or Ray [37] can be used to define exactly-once semantics for ensuring fault tolerance as well as aggregation operations (e.g. joins) over multiple (potentially stateful) streams. These interfaces, however, do not easily let developers automatically specify hierarchical nesting of streams and track the ancestry of partial outputs through this hierarchy as data fans out over downstream stages. Therefore existing streaming systems will have difficulty enforcing aggregation-aware routing at different granularities throughout the pipeline.

3.2 Efficient Load Balancing across Prompts

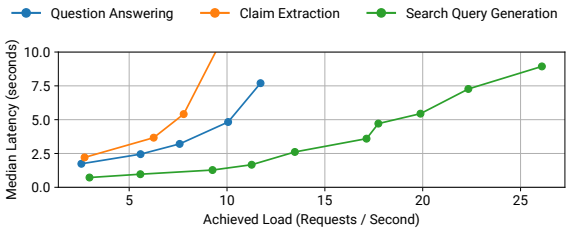
Streaming partial outputs between pipeline stages can generate dynamic fan-out of partial outputs spanning multiple quanta. This can complicate load balancing for LM stages. Table 1 measures this fan-out for the LM stages within the FacTool-inspired pipeline from Figure 1. We observe that each prompt generates partial outputs which vary significantly across their size and processing times. Figure 5 further illustrates the diversity across prompts. In this experiment we plot the latency achieved by each prompt type in the FacTool-inspired pipeline as we increase the number of requests; here each prompt saturates a single GPU at a different rate.

Stage	Overall Count	Per-output Quantum	Average # Outputs	Average Length / Output (words)	Average Time / Output (ms)
Question Answering	10795	Response (paragraphs)	-	62.5 ± 57.2	1292.3 ± 1175.0
Claim Extraction	10795	Claim	3.3 ±1.8	9.8 ± 3.5	403.6 ± 1175.0
Search Query Generation	35516	Search query	2.5 ± 1.3	5.5 ± 3.1	326.6 ± 252.4
		Search query token	5.5 ± 3.1	-	59.8 ± 79.3

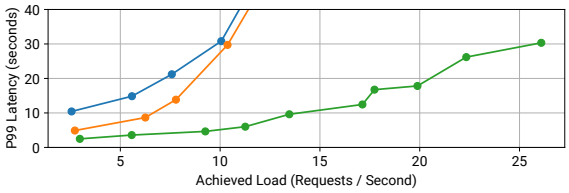
Table 1. Statistics measured from a run of the FacTool-inspired pipeline from Figure 1. Inputs are issued at a rate of 15 requests / second for 12 minutes. Each prompt generates a different output unit and number of outputs.

Serving LM instances efficiently with streaming requires load balancing the dynamic fan-out across downstream stages. The experiment detailed in §2 fixes a particular prompt to each LM instance, but statically assigning prompts precludes adapting to workload-dependent dynamic fan-out in the pipeline. If the static assignment does not match the relative frequencies of each prompt, instances serving certain prompts will be under-utilized while other instances serving other prompts will be over-utilized.

An ideal load balancing strategy should dynamically forward requests for all prompts across all LM instances, without fixing a static assignment. Remaining completely agnostic to the prompt content, however, may also compromise serving throughput. LM serving engines [13, 16, 45] are capable of re-using KV cache state for frequently occurring prompts that share a common prefix, thereby improving



(a) Median Latency vs Achieved Load.



(b) P99 Latency vs Achieved Load.

Figure 5. Microbenchmark measuring the diversity in performance achieved by each prompt in the FacTool-inspired pipeline from Figure 1. We only include points where the achieved load was ≥ 80% of the offered load.

overall request throughput. Maximally leveraging this optimization requires issuing many requests with the same prefix to the same LM instance (i.e. maximizing prompt locality). While LM serving engines apply local scheduling algorithms to maximize in-batch prompt locality [45], scheduling has to happen at the network level to encourage locality across distributed LM instances.

Unfortunately, enforcing prompt locality directly conflicts with the goal of retaining load balancing flexibility across different prompts. Resolving this tension is a key challenge for efficient load balancing, and requires *distributed prompt-aware scheduling*; we discuss some preliminary ideas in §6.2.

4 ALTO System Design and Interface

This section describes a potential system design for ALTO that does partial output streaming while addressing the challenges described in §3. Note that the version of ALTO used to generate the results presented in §2 has the same general architecture, but does not include the scheduler design for aggregation constraints.

ALTO consists of two pieces: (1) an inference interface for individual pipeline stages, with queues sitting between stages, and (2) a central runtime that forwards data between these stages.

ALTO has three design goals:

- *Ensure partial state is aggregated correctly for any stateful pipeline stage.* When certain pipeline stages must aggregate fanned-out work, ALTO must ensure partial outputs that need aggregation are sent to the same instance of the stage.
- *Load balance as much work as possible evenly across replicas.* While respecting aggregation constraints, ALTO should load balance work that arrives at the central scheduler as evenly as possible. ALTO should maximize parallelism within queries where possible when aggregation constraints permit.
- *Lightweight interface to specify aggregation constraints and prompt information.* The interface to specify aggregation constraints and prompt information should be lightweight on top of the queueing interface.

To fulfill these goals, ALTO is modeled off of microservices but deviates from existing microservice programming models in two ways. Instead of RPCs, ALTO provides an API to specify *aggregation constraints* and *prompt information*. The scheduling algorithm uses this information to ensure correctness and improve load balancing. The rest of this section describes the basic queueing interface, explains how developers specify aggregation constraint and prompt information, and defines a scheduling algorithm that uses this information to load balance across stages.

Developer interface. Application developers specify a pipeline by a sequence of stages. Stages process data, executing LMs and aggregation or other computation steps, and communicate data to downstream stages through queues. Application developers use Protocol Buffers [35] to specify the data format of each queue. Each stage reads data off of its input queues, processes the data, and pushes output data onto one or more queues to the next stages.

4.1 Aggregation Constraints Interface

At a high level, ALTO lets the central scheduler know about aggregation constraints and prompt affinities via a header on each data item in a queue. Application developers specify these headers to define when aggregation is required at at what granularity. For example, for the FacTool pipeline a query can be tagged with a `query_id`, but when the pipeline generates a claim for the query in a later stage, the application can augment the header with a `claim_id` to ensure that aggregation will take place correctly at both the claim and query level.

The following code snippet shows how the queue interface includes this argument:

```
write(
    queue="bm25", obj=Token(...), id=obj_id,
    constraints=[obj_id, claim_id, query_id]
)
```

The first part of the header is an array of integers and allows an application to express custom aggregation constraints: as long as the application attaches the same array of integers to any data that needs to be aggregated (e.g., the global `obj_id`, `claim_id`, and `question_id` for any individual token sent to BM25 in the FacTool pipeline), the central scheduler will

send all this data to the same instance of the destination stage.

4.2 Scheduling Policy

The ALTO runtime currently uses a simple scheduling policy to respect aggregation constraints while still maximizing parallelism opportunities. When data arrives for a given stage, the scheduler checks to see whether an aggregation constraint exists in front of the data. If it does, it hashes the aggregation constraint and mods it with the number of instances for the destination stage; this ensures that data with the same aggregation constraints are forwarded to the same instance. If not, ALTO chooses the next instance in a round robin fashion. This simple algorithm could be augmented with techniques such as consistent hashing to ensure even load balancing in the case of instances coming up and down. Note that the ALTO scheduler does not yet support distributed prompt-aware scheduling, but we discuss preliminary ideas in §6.2.

5 Implementation

Figure 6 presents a system diagram of ALTO. ALTO includes a centralized runtime which routes data through a series of asynchronous queues. Applications running on top of ALTO communicate with the centralized runtime by receiving data from input queues and sending intermediate data through output queues.

Queues. ALTO uses queues to asynchronously forward data between pipeline stages. Each queue is a wrapper over two reliable UNIX domain sockets, though this can be expanded to a multi-node setting by wrapping a network socket instead. One socket is from the source stage to the central runtime (output queue), while the second socket is from the central runtime to the next stage (input queue). Each queue has an associated user-defined Protobuf describing the data type (e.g., Token or Claim). Deserialization only happens in the application; in-flight data is not deserialized as it is forwarded through the runtime.

Centralized runtime. The ALTO runtime accepts as input two configuration files: the first specifies the individual pipeline stages as well as their resource requirements and input and output queue names, while the second specifies how many instances to spawn for each pipeline stage. The runtime automatically adds a global request ID to the data headers corresponding to each request. The runtime is also responsible for enforcing aggregation constraints and scheduling in-flight data as discussed in §4. The runtime is implemented in ~7000 lines of Rust.

Applications. In our prototype implementation, ALTO stages are written in Python. The ALTO runtime passes the input

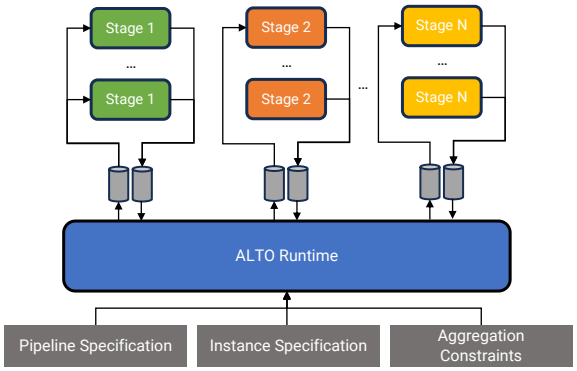


Figure 6. ALTO system diagram. ALTO includes an asynchronous queueing interface and centralized runtime to automatically route data between pipeline stages.

and output queue names to each stage when it is started, and then the developer then calls an ALTO library function to initialize the application-side sockets. The application-side queues use `asyncio` for asynchronous execution. ALTO supports arbitrary LM serving engines as long as they can interface with `asyncio` and asynchronously stream incremental outputs. We have currently implemented integrations with vLLM [16] and SGLang Runtime [45].

6 Discussion

Here we discuss opportunities to improve the current aggregation interface and design a distributed prompt-aware scheduler.

6.1 Improving Constraint Interface

While the interface described in §4 can correctly express aggregation constraints, it requires the developer to manually specify tags to define the aggregation logic for stateful stages. Instead, ALTO should automatically infer aggregation logic from the pipeline structure itself, via extra annotations provided by the programmer. The annotations would provide a way to tell ALTO which stages are stateful and aggregate data, and which stages cause fan-out (produce multiple partial outputs for a single inputs). Using these hints, ALTO can automatically infer tags. For example, in the FacTool pipeline from Figure 1, the developer could annotate that stage 4 is stateful, and how many levels of quanta this stage aggregates. The runtime could then infer the aggregation constraint that all search query tokens should go to a consistent BM25 instance, and generate the tags needed by the scheduler.

Another opportunity would be to implement a general set of aggregation operators to express a wide variety of aggregation patterns. Currently each application running on top of ALTO must implement bespoke aggregation logic, but instead the common design patterns could be abstracted

away into a library which is tied to the aggregation constraint interface. Potential operators for this library would include `sum`, `top-k`, `count`, and `filter`.

6.2 Distributed Prompt-Aware Scheduling

Our current implementation of the ALTO scheduler takes into account aggregation constraints based on developer-specified headers, but does not use information about prompts when scheduling. There are two opportunities here: profiling the relative resource consumption across prompts and understanding the optimal resource allocations between prompts, and prompt-aware routing. As discussed in §3, recent work on language model serving [13, 16, 45] has demonstrated that LM throughput can improve when requests sharing the same prompt prefix are routed through the same LM. ALTO can take advantage of these opportunities with a distributed prompt-aware scheduling algorithm.

The design goals for a distributed prompt-aware scheduling algorithm are twofold:

1. Support flexible load balancing to handle dynamic fan-out (understand the relative resource consumption between prompts and automatically assign GPU time to prompts based on this).
2. Maximize prompt locality when possible (to take advantage of LM engines' prompt sharing optimizations).

A first-pass attempt at designing a distributed prompt-aware scheduling algorithm to achieve these goals would involve two mechanisms: a mechanism to measure statistics about each prompt's relative resource consumption and a mechanism to route a prompt request to a particular LM instance. For the first mechanism, the scheduler could keep track of statistics related to how much output data each prompt request tends to create, and how long each request type takes to serve, and queuing delay at each LM instance. Measuring these statistics accurately is challenging as model serving engines contain internal scheduling mechanisms to handle batching, which sometimes de-prioritize requests relative to others (e.g., Radix attention). For the second mechanism, we speculate the scheduler could run an optimization problem using these statistics, which also encourages "sticky" routing rules, where it keeps sending requests for the same prompt to the same LM instance. We defer fully exploring these ideas to future work.

7 Related Work

Compound AI system front-ends Many frameworks and domain-specific programming languages offer interfaces for expressing compound AI systems using high level abstractions [6, 14, 19, 45]. ALTO can efficiently serve the pipelines expressed in these higher level abstractions given some intermediate translation layer.

LM serving systems LM serving systems optimize LM inference throughput by efficiently managing the memory used by the LM computation across requests [13, 16, 45]. These engines and many commercial LM API endpoints also expose interfaces for streaming tokens. Unlike ALTO, these systems do not handle distributed deployments nor the associated correctness and load balancing challenges which emerge when streaming partial outputs. Furthermore, these systems exclusively optimize LMs rather than combinations of LMs with other tools. ALTO can use these systems as high-throughput LM executors as we discuss in §5.

Parallelizing compound AI systems Previous works have tried to automatically parallelize graphs of LM calls [15, 23, 30]. While ALTO can also execute logically parallel stages concurrently, these approaches do not leverage partial output streaming or handle any of the correctness and load balancing concerns discussed in §3.

Stream processing Streaming query engines have been widely explored in the database and systems communities [1, 4, 5, 22, 36, 37, 43]. These systems execute long-running queries that continuously output results while pipelining computations and reliably maintaining long-lived state. Like these systems, ALTO streams partial results between nodes, but it aims to minimize end-to-end latency for relatively short AI pipeline computations. The varying resource consumption of queries (e.g., longer or shorter LM outputs) also creates a need for dynamic pipeline-aware scheduling at a fine granularity in ALTO to keep worker nodes efficiently utilized.

Microservice serving systems Many distributed system frameworks are capable of deploying pipelines of logically independent computation stages as microservices and using queues to communicate data between these stages [21, 39]. Some are even optimized for machine learning workloads in particular [9]. In contrast to ALTO, these systems do not leverage the autoregressive generation property of LMs to facilitate partial output streaming.

8 Conclusion

This paper presents ALTO, a system that orchestrates compound AI system pipelines built around generative language models. ALTO is based on the observation that generative language models produce output incrementally, such that they can be streamed through a distributed pipeline. However, as some pipeline stages can parallelize outputs while others must aggregate them, correctly routing tokens requires careful orchestration and load balancing. ALTO provides an interface to specify such routing requirements. Experimental results show that ALTO's pipelining can both reduce latency and increase throughput of a representative compound AI application.

Acknowledgments

This research was supported in part by affiliate members and other supporters of the Stanford Platform Lab and the Stanford DAWN Project including VMware, Meta and Google, as well as the NSF under Career Grant CNS-1651570, Graduate Research Fellowship Grant DGE-1656518, and Grant No. 1931750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 666. https://doi.org/10.1145/872757.872855

[2] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*. 601–613.

[3] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. 2023. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687* (2023).

[4] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (12 2015).

[5] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. 668. https://doi.org/10.1145/872757.872857

[6] Harrison Chase. 2022. *LangChain*. https://github.com/langchain-ai/langchain

[7] Jifan Chen, Grace Kim, Aniruddh Sriram, Greg Durrett, and Eunsol Choi. 2023. Complex Claim Verification with Evidence Retrieved in the Wild. *arXiv preprint arXiv:2305.11859* (2023).

[8] I Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei Liu, et al. 2023. FacTool: Factuality Detection in Generative AI–A Tool Augmented Framework for Multi-Task and Multi-Domain Scenarios. *arXiv preprint arXiv:2307.13528* (2023).

[9] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 477–491.

[10] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-Verification Reduces Hallucination in Large Language Models. *arXiv preprint arXiv:2309.11495* (2023).

[11] Jie Huang, Wei Ping, Peng Xu, Mohammad Shoeybi, Kevin Chen-Chuan Chang, and Bryan Catanzaro. 2023. Raven: In-context learning with retrieval augmented encoder-decoder language models. *arXiv preprint arXiv:2308.07922* (2023).

[12] Gautier Izacard and Edouard Grave. 2021. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. In *EACL 2021-16th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 874–880.

[13] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. 2024. Hydragen: High-Throughput LLM Inference with Shared Prefixes. arXiv:2402.05099 [cs.LG]

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

[15] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2023. An LLM Compiler for Parallel Function Calling. *arXiv preprint arXiv:2312.04511* (2023).

[16] 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*. 611–626.

[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[18] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111* (2023).

[19] Jerry Liu. 2022. *LlamaIndex*. https://doi.org/10.5281/zenodo.1234

[20] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Le Xue, Shelby Heinecke, Rithesh Murthy, Yihao Feng, Zeyuan Chen, Juan Carlos Niebles, Devansh Arpit, et al. 2023. Bolaa: Benchmarking and orchestrating llm-augmented autonomous agents. *arXiv preprint arXiv:2308.05960* (2023).

[21] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[22] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[23] Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. 2023. Skeleton-of-thought: Large language models can do parallel decoding. *Proceedings ENLSP-III* (2023).

[24] Rodrigo Nogueira, Wei Yang, Kyunghyun Cho, and Jimmy Lin. 2019. Multi-stage document ranking with BERT. *arXiv preprint arXiv:1910.14424* (2019).

[25] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).

[26] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[27] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083* (2023).

[28] Stephen Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. 1995. Okapi at TREC-3. In *Overview of the Third Text REtrieval Conference (TREC-3)* (overview of the third text retrieval conference (trec–3) ed.). Gaithersburg, MD: NIST, 109–126. https://www.microsoft.com/en-us/research/publication/okapi-at-trec-3/

[29] Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Xingyu Zeng, and Rui Zhao. 2023. Tptu: Task planning and tool usage of large language model-based ai agents. *arXiv preprint arXiv:2308.03427* (2023).

[30] Swarnadeep Saha, Omer Levy, Asli Celikyilmaz, Mohit Bansal, Jason Weston, and Xian Li. 2023. Branch-solve-merge improves large language model evaluation and generation. *arXiv preprint arXiv:2310.15123* (2023).

[31] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. 2022. PLAID: an efficient engine for late interaction retrieval. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 1747–1756.

[32] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 3715–3734.

[33] Sina Semnani, Violet Yao, Heidi Zhang, and Monica Lam. 2023. WikiChat: Stopping the Hallucination of Large Language Model Chatbots by Few-Shot Grounding on Wikipedia. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 2387–2413.

[34] Ruoxi Sun, Sercan O Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. SQL-PaLM: Improved Large Language ModelAdaptation for Text-to-SQL. *arXiv preprint arXiv:2306.00739* (2023).

[35] Kenton Varda. 2008. Protocol buffers: Google's data interchange format. https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.

[36] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, et al. 2021. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 international conference on management of data*. 2602–2613.

[37] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 338–352.

[38] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*.

[39] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review* 35, 5 (2001), 230–243.

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

[41] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.

[43] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[44] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and et al. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/

[45] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently Programming Large Language Models using SGLang. *arXiv preprint arXiv:2312.07104* (2023).