ALTO: An Efficient Network Orchestrator for Compound AI Systems

Keshav Santhanam*¹, Deepti Raghavan*¹, Muhammad Shahir Rahman¹, Thejas Venkatesh¹, Neha Kunjal¹, Pratiksha Thaker², Philip Levis¹, Matei Zaharia³

*Equal contribution

¹Stanford University  ²CMU  ³UC Berkeley
**Compound AI Systems** combine AI models with external tools to solve challenging tasks.

Retrieval-augmented generation (RAG)

**Compound AI Systems**

**Compound AI systems** combine AI models with external tools to solve challenging tasks.

Writing Wikipedia articles from scratch (STORM)

**Compound AI Systems**

**Compound AI systems** combine AI models with external tools to solve challenging tasks.

---

**Proving mathematical theorems (AlphaGeometry)**

Compound AI Systems

**Compound AI systems** combine AI models with external tools to solve challenging tasks.

Many frameworks and DSLs exist to build compound AI systems.
Compound AI Systems

Compound AI systems combine AI models with external tools to solve challenging tasks.

How do we serve compound AI systems efficiently at scale?
Streaming partial outputs

• Key optimization: Streaming partial outputs between pipeline stages
  • Partial outputs are segments of text such as words, sentences, or paragraphs
  • Generative LMs output text incrementally so we can emit these partial outputs as soon as they are generated

• FacTool¹ as a representative pipeline:
  1) Chat-bot answers question
  2) Extract factual claims
  3) Generate search queries for each claim
  4-6) Search for corroborating evidence

Streaming partial outputs

• We can pipeline across requests by treating each stage as a microservice

• Streaming partial outputs enables pipelining within a single request

• This reduces per-request latency and enables higher serving throughput

• Our prototype system ALTO enables partial output streaming
Empirical evaluation using FacTool

- We vary the input load (requests / second) and measure the resulting per-request latency.
- We compare streaming partial outputs (orange) vs fully materializing LM outputs (blue).
- Streaming partial outputs enables up to $3\times$ higher serving throughput* while reducing tail latency by $1.8\times$.

*For a fixed latency target of 4 seconds / request.
ALTO: Automatic Language Token Orchestrator

- **ALTO**: a serving system for automatically distributing and parallelizing compound, streaming AI pipelines

- Key challenges when streaming partial outputs:
  - **Correctness**: How do we segment partial outputs automatically and route them through the correct pipeline stage instances?
  - **Efficient load balancing**: Given a pipeline with heterogeneous prompts, how should we dynamically schedule prompts across instances?

- We discuss the need for **aggregation constraints** and **distributed prompt-aware scheduling** to address these challenges
Stateful pipeline stages

• Some pipeline stages are **stateful**, which means all partial outputs routed through this stage must follow a consistent path

[Diagram showing Stateful pipeline stages with BM25 Retrieval and Search Query Generation stages, and an example streaming query: “Who is Tom Cruise?”]
Stateful pipeline stages

• Some pipeline stages are **stateful**, which means all partial outputs routed through this stage must follow a consistent path.

BM25 stage is **stateful**, so every token in this query’s stream must be routed through the same instance.

We call this an **aggregation constraint**.
Stateful pipeline stages

- Some pipeline stages are **stateful**, which means all partial outputs routed through this stage must follow a consistent path.

BM25 stage is **stateful**, so every token in this query’s stream must be routed through the same instance.

We call this an **aggregation constraint**.
Stateful pipeline stages

- Some pipeline stages are **stateful**, which means all partial outputs routed through this stage must follow a consistent path.

BM25 stage is **stateful**, so every token in this query’s stream must be routed through the same instance.

We call this an **aggregation constraint**.

Streaming query “Who is Tom Cruise?” token-by-token.

Timestamp: 3
Stateful pipeline stages

- Some pipeline stages are **stateful**, which means all partial outputs routed through this stage must follow a consistent path.

```
Timestamp: 4
```

```
Search Query Generation
```

```
BM25 Retrieval
```

```
BM25 Retrieval
```

```
BM25 Retrieval
```

```
Streaming query “Who is Tom Cruise?”
token-by-token
```

```
Cruise?
```

BM25 stage is **stateful**, so every token in this query’s stream must be routed through the same instance.

We call this an **aggregation constraint**.
Aggregation constraints

Load balancing without an aggregation constraint

Load balancing with an aggregation constraint

Enforcing aggregation constraints can limit load balancing efficiency
Interface for aggregation-aware routing

• Currently in ALTO we provide this interface to specify aggregation constraints:

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

• We can enforce a consistent path by taking hash(constraints) % N (# of downstream stage instances)

• This requires explicitly encoding hierarchical ids in the object definitions, but we are working on a design for automating this
Prompt heterogeneity

- Prompts can vary significantly in their output type, frequency, and output volume
- We can observe this empirically within the FacTool pipeline

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

• Many LM serving engines (e.g. vLLM, SGLang Runtime, Hydragen) benefit from serving the same prompts repeatedly on a single GPU by re-using prompt prefixes in the KV cache

• In a distributed setting we want to balance (1) dynamically load balancing prompts across stage instances with (2) maximizing prefix locality – these goals may be in conflict!

• This requires **distributed prompt-aware scheduling**

• ALTO does not currently implement distributed prompt-aware scheduling, but see our paper for some initial design ideas

---


ALTO Implementation

• ALTO is implemented using an asynchronous queue interface over UNIX domain sockets

• Applications (written in Python) send Protobuf messages to a centralized runtime (written in Rust) which routes messages to downstream stage instances

• We are also working on a Ray-based implementation
Conclusion

• We can optimize compound AI system serving by streaming partial outputs between pipeline stages.

• Our prototype system ALTO demonstrates this on the FacTool pipeline by improving throughput by up to 3x while reducing tail latency by 1.8x.

• Streaming partial outputs introduces the new challenges of correctness and efficient load balancing, which require aggregation constraints and distributed prompt-aware scheduling to solve.

keshav2@stanford.edu  https://cs.stanford.edu/~keshav2