Type-Driven DAG Synthesis for Streaming Compound AI Systems

Nayani Modugula

Advisor: Deepti Raghavan
Reader: Nikos Vasilakis

Department of Computer Science
Brown University
Providence, RI

April 2026

Contents

Abstract

Compound AI Systems (CAIS), applications consisting of LLM and tool calls, are becoming more widespread and complex, often comprising many sequential stages. These applications can benefit from streaming intermediate LLM output so that downstream stages can start earlier, lowering end-to-end latency. ALTO is a framework that supports streaming in CAIS, but is difficult to use because developers must specify low-level orchestration details rather than focusing on application logic. The core challenge is that ALTO connects stages via a shared queue that carries flat, serializable items rather than the input type each stage expects – for example, individual stream items instead of a stream object. Before invoking a stage, the system must ensure it is invoked exactly once per stream and, when multiple inputs are present, that only items belonging to the same request are matched together. Users are currently expected to handle this by manually constructing a preprocessing DAG consisting of ALTO primitives, which is tedious and requires deep knowledge of the framework internals. This thesis addresses this programmability limitation so that developers only write application logic while retaining the performance benefits of streaming and pipelining. The key insight is that the required preprocessing DAG is fully determined by a stage's input types. Using this insight, we build a decorator that inspects input type annotations to automatically synthesize the DAG, eliminating manual specification. This reduces code by 50–55% compared to the previous ALTO version while preserving performance benefits.

Acknowledgements

First and foremost, I would like to thank my advisor, Deepti Raghavan, for always being there and believing in me. I could not have asked for a better advisor to guide me through this journey. Thank you for encouraging me to pursue independent research and expertly guiding me through the many twists and turns this project has presented. I also want to thank the ALTO team, particularly Keshav, Matei, and Phil, for creating an amazing project I was able to build on and for their feedback throughout the different iterations of the new API. Thank you, Eric and Edward, for being part of the broader ALTO project, it was a joy working with you both. Thank you to the Brown Systems group for being a great, supportive community; Morgan's humor and candid advice on research and life is something I'll carry well beyond this thesis. Finally, I want to thank my family and friends for their endless emotional support and warm hugs.

Chapter 1

Introduction

Compound AI Systems (CAIS) are becoming more prevalent, with many applications relying on external tools like search engines, document retrievers, and code execution alongside large language model calls. Oftentimes these applications are sequential, and there can be performance benefits by streaming partial output to start downstream stages earlier. However, implementing these optimizations requires careful orchestration to ensure the correct data is sent to each stage. This thesis builds on ALTO [6], a framework for streaming CAIS pipelines, and addresses a key limitation in its previous version.

1.1 Problem Overview

Building CAIS with streaming optimization requires users to reason about low-level execution details. In the previous ALTO version, stages were separate processes and connected using a shared queue across requests. This queue contained flat, serializable items instead of the structure expected by the downstream stage, for example, stream items instead of a stream object. Before a stage could be invoked, the queue had to be demultiplexed and the items had to be transformed to ensure that a stage is invoked per stream and in the case of multiple inputs, a stage is invoked with items belonging to the same request. In order to do this users had to manually construct a preprocessing directed acyclic graph (DAG) consisting of ALTO primitives. This conflation of application logic with orchestration concerns made it harder to write, reason about, and maintain these pipelines.

1.2 Motivation

Manually implementing execution details is tedious and represents an opportunity cost where developer effort spent on orchestration details could instead have been spent on application logic. As pipelines grow and consist of more stages, this burden compounds, as each stage requires its own preprocessing DAG. A small mistake in this orchestration code can lead to subtle bugs that are difficult to trace. A higher-level abstraction would allow developers to focus on the core application logic instead of orchestration and data preprocessing details.

1.3 Key Ideas and Contributions

The key insight is that the preprocessing DAG is fully determined by a stage's input types. This thesis presents a decorator that inspects the input type annotations of a stage to automatically synthesize the preprocessing DAG coordination logic, reducing the application code by approximately 50-55% and eliminating the need for manual DAG specification.

Chapter 2

Background

2.1 Compound AI Systems (CAIS)

Compound AI systems integrate multiple large language models (LLMs) or tools into a single workflow. These systems have become increasingly popular as they are well-suited for complex tasks such as a fact checker that retrieves documents and uses that information to verify claims [1], providing answers by grounding them based on Wikipedia [7], or refining SQL queries [2]. Such workflows often rely on subcomponents like retrieval-augmented generation (RAG), multi-step chains, and judge models, where developers break down complex tasks into sequential stages. Each stage typically waits for the previous one to complete, resulting in high end-to-end latency. This is particularly costly in CAIS because LLM stages generate output incrementally, yet high latency downstream stages sit idle even though they can begin processing partial results.

2.2 Opportunities for Streaming in CAIS

CAIS consist of sequential stages and have high end-to-end latency, which can be reduced by streaming intermediate outputs so that downstream stages can start earlier. For example, in Factool [1], an application that verifies the factual claims in an LLM's response to a question, there are seven stages, as shown in Figure 2.1. The blue stages involve invoking an LLM and the green stages are document retrieval and reranking stages. Given an input question, the application first generates a response and then extracts a list of claims from it. Each claim is then processed through a subgraph that generates search queries, retrieves relevant documents, and verifies the claim. The final output is a list of verified claims.

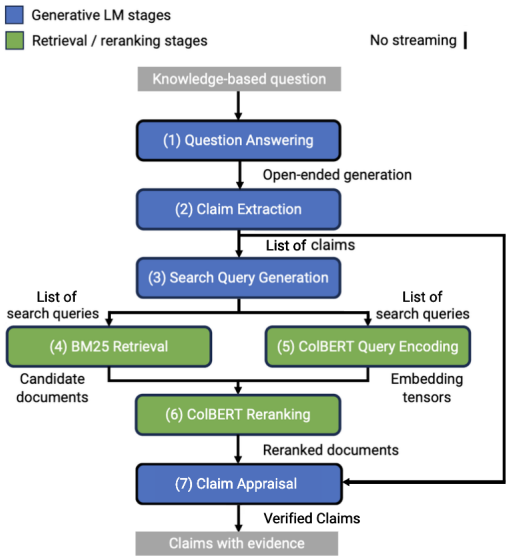

Figure 2.1: Factool before Streaming/Pipelining

This sequential structure creates opportunities for pipelining. As LLMs output a stream, by streaming intermediate outputs, downstream stages can begin processing earlier. Concretely in Factool, a claim can immediately be processed in the claim subgraph after it is generated instead of waiting for the entire list of claims to be generated by the LLM. Similarly, a search query can immediately be processed in the search subgraph after it is generated instead of waiting for the entire list of search queries. Figure 2.2 illustrates this pipelined version of Factool.

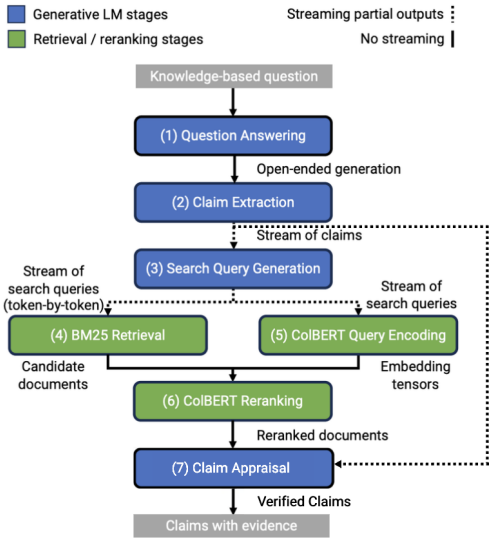

Figure 2.2: Factool after Streaming/Pipelining

2.3 ALTO: Previous Implementation

In the previous version of ALTO, each stage in the CAIS is its own process, connected together using queues and Unix sockets. Many requests share the same queue, so the system uses unique identifiers called ancestry tags on each item to distinguish them. These queues carry flat, serialized items instead of the structured Python types (e.g. `Stream[...]`, `Stream[Stream[...]]`, `Joined[...]`) that the stage expects, for example, stream items instead of a stream object. This matters because a stage must be invoked once per stream, and not once per stream item. Furthermore, in the case of multiple inputs, a stage must be invoked with inputs belonging to the same request.

Demultiplexing becomes particularly complex when stages expect input types such as nested streams or tuples that combine items from multiple upstream modules, as the system must reorganize flat items into the correct hierarchical structure and synchronize multiple inputs. To handle this, ALTO requires each stage to have a preprocessing DAG consisting of ALTO primitives that demultiplexes the shared queue and transforms the flattened items into the input type expected by the stage. This preprocessing DAG must be manually constructed by the developer for each stage, a process that is both tedious and orthogonal to the application logic itself.

Chapter 3

Programmability Issues with Previous ALTO Version

In an ideal framework, developers should only need to write application-specific logic. However, the previous version of ALTO required developers to also write orchestration code. This section examines the programmability issues in the previous ALTO version and why they occur in the first place. Each stage in a compound AI system is its own process and is connected using a shared queue. The queue consists of flat, serializable items instead of the input type expected by the downstream stage. For example, going back to Factool, the CAIS discussed in Chapter 2, we can examine the claim appraisal stage which takes in two inputs: a stream of reranked documents and a claim. The queue between reranker and claim appraisal consists of reranked documents corresponding to different claims and requests, and the queue between claim extraction and claim appraisal stages, consists of claims belonging to different requests. Correct execution requires that claim appraisal is called using a stream of reranked documents corresponding to a specific claim and request and the same claim for the same request. This is ensured by having a preprocessing DAG consisting of ALTO primitives. This must be manually constructed by the developer, forcing them to reason about execution mechanics rather than application logic, making ALTO difficult to use in practice.

The preprocessing DAG requires two key primitives: `StreamCollector` and `Join` nodes. A `StreamCollector` node collects all items belonging to the same ancestry and groups them into a stream, transforming a flattened queue of items into the input type expected by the stage. A `Join` node synchronizes items from multiple upstream stages by maintaining per-ancestry buffer and only emitting a `Joined[A, B]` when both sides have matching ancestry tags, ensuring a stage is invoked with a coherent tuple rather than mismatched fragments.

The claim appraisal stage in Factool illustrates this complexity. The stage must be invoked with the claim and all reranked documents generated by the search subgraph across different search queries from that specific claim. A `StreamCollector` is needed to collect all reranked documents belonging to the same request and claim, and a `Join` node is needed to synchronize these collected documents with their corresponding claim. Figure 3.1 shows the resulting preprocessing DAG that must be manually specified before any application logic can execute.

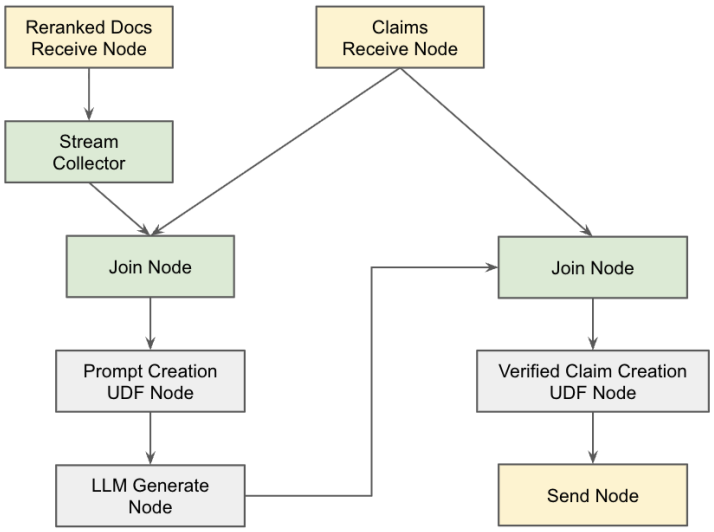

Figure 3.1: Preprocessing DAG for Claim Appraisal stage in Factool

This complexity compounds significantly in real applications as a DAG must be built for each stage. Furthermore, each stage is written as a different file which is then passed to a graph API, which uses this information to set up the queues between stages. This is a very tedious process and orchestration code takes up almost the same number of lines as the application logic itself.

Chapter 4

Design

4.1 Goal

The new framework should improve upon the programmability limitations of the previous ALTO version while supporting arbitrary CAIS applications and preserving ALTO's performance benefits through streaming and pipelining. We define three concrete goals:

1. **Programmability**: Developers should not need to define how stages are connected through a framework-specific graph interface. They also should not have to manually construct a preprocessing DAG that demultiplexes a shared queue and transforms flattened items into the structured input type expected by the stage. Ideally, users should only write code relevant to their application logic, without reasoning about orchestration or execution details. We measure programmability by the number of lines of code required to implement a CAIS application.
2. **Expressivity**: The framework must support dynamic control flow such as conditionals and loops, enabling developers to express arbitrary CAIS applications rather than being confined to static applications.
3. **Performance**: The framework should introduce minimal overhead and preserve the benefits of streaming and pipelining in CAIS.

4.2 Key Observation

The input type of a stage determines the preprocessing DAG that demultiplexes and transforms queue items to expected input type. In the previous version of ALTO, developers had to

manually construct this DAG using two primitives: `StreamCollector` and `Join`. The DAG itself can be created based on the input type information, for example, if a stage consumes a stream, a `StreamCollector` is needed, whereas if it consumes inputs from two upstream stages, a `Join` is needed. This means the system can inspect a stage's type annotations and automatically construct the DAG, without requiring the user to specify it manually.

4.3 Approaches

Meeting the design goals simultaneously is non-trivial as there are tradeoffs between programmability and performance. This is reflected by the three approaches explored where each represents a different point in this tradeoff space.

4.3.1 Static Analysis

This approach uses Abstract Syntax (AST) to infer input type of a stage and how stages are connected. This achieves the highest degree of programmability among the three approaches because the developer only writes code relevant to application logic.

The current implementation uses AST to infer stage dependencies and create a static graph structure. However, with conditionals and loops, the graph becomes dynamic where the stage at which data is sent to is only known at runtime. Supporting this is possible, but is non-trivial and out of scope for this project.

4.3.2 Per-Stream Private Queues

This approach eliminates the need for a preprocessing DAG by creating a private Ray queue [5] per stream per request. Each stage is represented as a Ray actor, a stateful worker process, and stages call each other directly, removing the need for a framework specific Graph API. In the previous version of ALTO, `StreamCollector` and `Join` nodes were needed to demultiplex the shared queue, transform queue items into the input type expected, and synchronize inputs from multiple upstream stages. Here, each stream has its own dedicated Ray queue and the downstream actor receives the queue handle directly, eliminating the need for a `StreamCollector` Node. `Join` nodes are not needed because Ray actors communicate using the Ray's distributed shared-memory object store. Object references are passed directly to the downstream actor, and Ray's scheduler ensures the actor is only invoked once all referenced objects are available. Stages are expressed as plain Python classes annotated with a decorator, which automatically handles actor setup, load balancing, and queue management.

While this approach meets the programmability requirements, it has significant performance limitations. Per-stream queue creation in Ray introduces substantial overhead, leading to higher end-to-end latency. This overhead arises because each Ray queue is itself a Ray actor and initializing it involves spawning a new process, setting up the Python environment, and registering it with the Ray control plane. This limitation makes it unsuitable as the final solution, but it is a useful baseline to compare against in Chapter 6.

4.4 Final Design: Type-Directed DAG Synthesis

A decorator can be used to automate the construction of the preprocessing DAG by inspecting the type annotation of the stage's processing function. Instead of manually instantiating and connecting the DAG node, users simply annotate the input type signature. Listings 4.1 and 4.2 illustrate this for the Claim Appraisal stage in Factool. In the previous version of ALTO, the user had to explicitly construct the preprocessing DAG by instantiating a `StreamCollector` for the stream of reranked documents and connecting it to a `Join` node. In the new version, this manual DAG construction is eliminated entirely. The processing method has an input signature of Joined[LMTextPipe, Stream[ScoredDocumentList]], from which the decorator infers that a `Join` node is needed to synchronize the two inputs and that the document stream requires a `StreamCollector`.

```
deserialize_node = DeserializeNode(
    TreeNodeKey("deserialize"),
    input_queue_group,
    state_node,
)
recv_nodes = deserialize_node.init_recv_nodes()
assert len(recv_nodes) == 2, len(recv_nodes)
docs_recv_node = recv_nodes[0]
docs_recv_node.debug = False
claims_recv_node = recv_nodes[1]
stream_collector = StreamCollector(TreeNodeKey("
   docs_stream_collector"))
docs_recv_node.add_child(stream_collector)
stream_collector.debug = False
input_join_node = Join(
    TreeNodeKey("verified_claim_input_join"), stream_collector,
       claims_recv_node
)
```

```
input_join_node.debug = False
input_join_node.add_child(prompt_udf_node)
```

Listing 4.1: Partial Preprocessing DAG in Previous ALTO version.

```
@alto.module()
class ClaimAppraisal():
def __init__(self, args):
  self.lm = alto.LM(args, prompt)
async def aforward(self, joined_input: Joined[LMTextPipe, Stream[
    RerankedDocs]])-> VerifiedClaim:
     claim_textpipe = joined_input.left
     docs_stream = joined_input.right
     claim = await claim_textpipe.as_full()

     ... application stage logic
```

Listing 4.2: ClaimAppraisal Stage in New ALTO version.

4.5 Discussion

The three approaches explored represent different points in the tradeoff space between programmability and performance. Static analysis improves programmability over manual DAG construction but fails to work with dynamic CAIS, limiting its use. The Per-Stream Private Queues approach supports dynamic CAIS with good programmability, but the overhead of per-request queue setup hurts performance. The final approach uses type signatures to automatically construct the preprocessing DAG, introducing no additional overhead and preserving the benefits of streaming and pipelining CAIS. The only additional requirement beyond standard application code is a decorator per stage and annotations for input type signature, which is part of good programming practice. However, stages are connected together using a framework-specific `@graph_module` interface with an explicit `define_graph` method. While this poses a programmability limitation, it is consistent with other frameworks like LangGraph [3]. The three approaches are evaluated against the design goals in the table below.

Approach	**Programmability**		**Expressivity**	**Performance**
	No Preprocessing DAG	**No Graph API**		
Static Analysis	✓	✓	×	✓
Per-Stream Private Queues	✓	✓	✓	×
Type-Directed DAG Synthesis	✓	×	✓	✓

Table 4.1: Comparison of approaches against design goals.

Chapter 5

Implementation

The type-directed DAG construction is implemented in approximately 550 lines of Python, excluding the decorator code that invokes it. The core contribution is a system that inspects a stage's input type annotations at startup and automatically constructs the preprocessing DAG that was previously built manually. This construction happens once per stage replica at initialization time.

5.1 Type Annotation Inspection

The decorator for the stage extracts the input type annotation from the stage's forward or aforward method, which are used to call a stage during execution. The supported input types are: `Stream[A]`, `Joined[A,B]`, `LMTextPipe` (wrapper around `Stream[str]` with additional manipulation utilities, e.g. consume the stream word by word), any Serializable subclasses, and arbitrary nestings of these types like `Stream[Stream[A]]` or `Joined[A, Stream[B]]`. The system recursively parses the type annotation before instantiating the nodes in the DAG.

5.2 DAG Construction

DAG construction happens in two phases: shadow tree construction and real tree construction.

The shadow tree is built first as a lightweight representation of the DAG structure. There are four possible cases for a type: `Joined`, `Stream`, `LMTextPipe`, or base type (str or Serializable subclasses), and each maps to a corresponding shadow node(s) as shown in Table 5.1. `Stream[A]` becomes a `Deserialize` node → `StreamCollector` chain and then recursively processes the inner type A. `Joined[A,B]` creates a single `Join` node attached as a child to both

branches, where each branch is recursively processed independently. `LMTextPipe` becomes `Deserialize` node → `StreamCollector` chain and the base case becomes `Deserialize` node. The shadow tree is built separately from the real tree because a `Join` node cannot be constructed until both of its parent nodes exist. By first building the shadow tree, the system can determine the full structure of the DAG without needing real node objects, avoiding half-initialized nodes or backtracking parents later.

Once the shadow tree is complete, the real runtime nodes are instantiated by traversing down the shadow tree. `Deserialize` and `StreamCollector` nodes are constructed during traversal, and `Join` node construction is deferred until both parent branches have been instantiated. Finally, depending on whether the stage outputs a stream or not, a `ProcessNode` or `StreamingProcessNode` is attached respectively, followed by a `SendNode` to emit results to the shared queue.

Input Type	**Shadow Tree DAG Nodes Generated**	**Description**
`Stream[A]`	Deserialize → StreamCollector	Reassembles flattened stream items by ancestry tag
`LMTextPipe`	Deserialize → StreamCollector	Reassembles flattened string stream items by ancestry tag
`Joined[A, B]`	Deserialize → Join	Synchronizes two inputs by ancestry tag
`Serializable`	Deserialize	No stream reassembly needed

Table 5.1: Input types and their corresponding Shadow Tree DAG nodes.

5.3 Limitations

The current implementation supports up to two inputs for a `Join` node, reflecting a binary `Joined[A, B]` type. Stages that require more than two synchronized inputs would need to express this as nested joins, for example `Joined[A, Joined[B, C]]`. This is a known constraint of the system that was also present in the previous version of ALTO.

Chapter 6

Evaluation

The new ALTO framework uses a decorator that inspects input type signatures to automatically construct preprocessing DAGs. This is evaluated to answer three questions:

1. How much additional effort is required to express a CAIS application in the new ALTO framework? (§6.1)
2. Can the new ALTO framework express arbitrary CAIS applications? (§6.2)
3. Does the new ALTO framework preserve the performance benefits of streaming and pipelining? (§6.3)

6.1 Programmability

We measure programmability by the number of lines of code required to implement a CAIS application, comparing the new ALTO framework against the previous version. Table 6.1 shows that the new framework reduces application code by approximately 50-55% across all applications evaluated. This reduction comes primarily from the elimination of manually specified preprocessing DAGs.

Application	**Previous ALTO Version**	**New ALTO Version**
Factool	1,700 LoC	800 LoC
Tree-of-Thought	1,600 LoC	720 LoC

Table 6.1: Lines of code comparison between previous and new ALTO framework.

Beyond lines of code, we also found the development experience to be substantially smoother. The developer doesn't have to manually construct a preprocessing DAG for each stage,

define StateNode routing rules to direct items between stages, and specify LM subgraphs for language model stages. Ideally, a user study would be conducted to validate this more rigorously.

6.2 Expressivity

We evaluate whether the new framework can express a range of CAIS applications with different pipeline structures. Table 6.2 summarizes the applications we implemented.

Application	**Type**
Factool	Sequential pipeline with nested parallelism
Tree-of-Thought	Iterative loop with nested parallelism
DeepScholar	Multihop loop
AdvancedRAG	Parallel subgraphs with pipelined execution

Table 6.2: Applications implemented using new ALTO Framework.

We found that the new framework can express all of the applications above. The decorator successfully infers the preprocessing DAG for stages with Stream, Joined, and nested stream input types across all applications. Furthermore, unlike the static analysis approach, the framework supports dynamic control flow including conditionals and while loops.

6.3 Comparison to Existing Abstractions

We evaluate whether the new framework preserves the performance benefits of streaming and pipelining compared to the RPC-style private queue approach from §4.3.2.

The two approaches were run on 4 NVIDIA L40S Ada 48 GB GPUs, attached to 2 64-core AMD EPYC 9554 3.1 GHz CPUs, with 768 GB of RAM. Due to the RPC-style baseline being unable to sustain throughput beyond 0.25 req/s, we compare both approaches at this operating point. Table 6.3 reports end-to-end latency for Factool at 0.25 req/s for 720s.

Approach	**Median (s)**	**P95 (s)**	**P99 (s)**
New ALTO (Type-Directed DAG)	33.4	70.9	78.7
Per-Stream Private Queues	45.4	85.7	93.4

Table 6.3: End-to-end latency comparison at 0.25 req/s on Factool.

The new framework achieves lower latency across all percentiles, with a 26% reduction in median latency. To investigate the source of this gap, we profiled the RPC-style approach and found that the first write to a Ray Queue took a median of 0.715s, compared to 0.002s for subsequent writes. This suggests that Ray Queue initialization contributes to the latency overhead, though the extent to which this accounts for the full 12s gap in median latency requires further investigation. In the future, we plan to profile both approaches more comprehensively to fully characterize the sources of the latency difference.

Chapter 7

Conclusion and Future Work

The previous version of ALTO, a framework for streaming and pipelining compound AI systems, had limited programmability as for each stage developers had to manually define a preprocessing DAG to demultiplex, transform flat queue items to expected input type and synchronize multiple inputs. This thesis found that the preprocessing DAG is dependent on input type of a stage and built a decorator that uses type annotations to auto-synthesize the DAG. In terms of programmability evaluation, the lines of code to build a CAIS in the new ALTO version decreased by 50-55%. This was done while still retaining expressivity of CAIS and performance benefits from streaming and pipelining. In the future, we hope to eliminate the graph API requirement by taking inspiration from TensorFlow's tracing mechanism [4], where a sentinel object that records all routing decisions is passed through the application to automatically synthesize the pipeline topology.

Bibliography

[1] I Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei Liu, et al. Factool: Factuality detection in generative ai–a tool augmented framework for multi-task and multi-domain scenarios. *arXiv preprint arXiv:2307.13528*, 2023.

[2] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*, 2023.

[3] LangChain AI. LangGraph. https://github.com/langchain-ai/langgraph, 2024.

[4] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. Ag: Imperative-style coding with graph-based performance. *Proceedings of Machine Learning and Systems*, 1:389–405, 2019.

[5] 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 *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[6] Deepti Raghavan, Keshav Santhanam, Muhammad Shahir Rahman, Nayani Modugula, Luis Gaspar Schroeder, Maximilien Cura, Houjun Liu, Pratiksha Thaker, Philip Levis, and Matei Zaharia. Alto: Orchestrating distributed compound ai systems with nested ancestry. *arXiv preprint arXiv:2403.04311*, 2024.

[7] Sina Semnani, Violet Yao, Heidi Zhang, and Monica Lam. 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*, pages 2387–2413, 2023.