Pythia: Exploiting Workflow Predictability for Efficient Agent-Native LLM Serving

Shan Yu[1*], Junyi Shu[1*†], Yuanjiang Ni[2], Kun Qian[2], Xue Li[3], Yang Wang[4], Jinyuan Zhang[1], Ziyi Xu[5], Shuo Yang[6], Lingjun Zhu[3], Ennan Zhai[2], Qingda Lu[2], Jiarong Xing[7], Youyou Lu[8], Xin Jin[9], Xuanzhe Liu[9], Harry Xu[1]

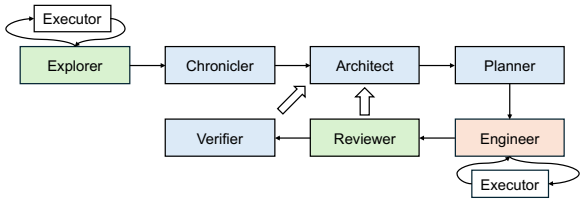
[1]UCLA [2]Alibaba Cloud Computing [3]Alibaba Group [4]Intel [5]SJTU
[6]UC Berkeley [7]Rice University [8]Tsinghua University [9]Peking University

Abstract

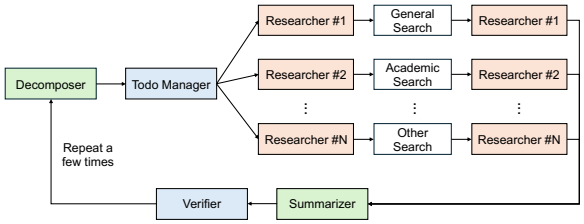
As LLM applications grow more complex, developers are increasingly adopting multi-agent architectures to decompose workflows into specialized, collaborative components, introducing structure that constrains agent behavior and exposes useful semantic predictability. Unlike traditional LLM serving, which operates under highly dynamic and uncertain conditions, this structured topology enables opportunities to reduce runtime uncertainty—yet existing systems fail to exploit it, treating agentic workloads as generic traffic and incurring significant inefficiencies. Our analysis of production traces from an agent-serving platform and an internal coding assistant reveals key bottlenecks, including low prefix cache hit rates, severe resource contention from long-context requests, and substantial queuing delays due to suboptimal scaling. To address these challenges, we propose Pythia, a multi-agent serving system that captures workflow semantics through a simple interface at the serving layer, unlocking new optimization opportunities and substantially improving throughput and job completion time over state-of-the-art baselines.

1 Introduction

The conventional wisdom in general LLM serving is that *everything is unpredictable*: neither the types nor the number of models activated at any given time can be anticipated, and the same holds for the types and volumes of requests directed to each model. This uncertainty stems from the fact that model providers typically expose only API endpoints for users to access their models, without control over when or how those models are invoked. Consequently, most prior work on LLM serving [1–4] assumes a highly dynamic execution environment, where nearly every configuration parameter of the serving system (*e.g.*, KV cache size, sharing mode, or autoscaling level) must be adjusted adaptively based on the workload observed at runtime. While such adaptability offers considerable flexibility, a fully reactive serving system often converges to suboptimal policies.



(a) Our internal coding assistant.



(b) An example of deep research agent.

Figure 1: Examples of multi-agent workflows.

Production trace analysis. To demonstrate how existing black-box approaches stifle efficiency, we analyzed large-scale production traces from our agent-serving service. We conducted in-depth profiling of an internal multi-agent coding assistant. Our analysis (§2) exposes three fundamental challenges in serving agentic workloads that contradict common assumptions. First, cache effectiveness is far lower than expected: specialized agents use distinct prompts and are invoked with long temporal gaps, causing previously cached prefixes to be evicted and leading to frequent zero-hit scenarios and long prefill latency. Second, the system suffers from severe resource inefficiency due to workflow-agnostic scheduling: heterogeneous requests are arbitrarily mixed, creating load imbalance across replicas and models, inducing memory contention, frequent preemptions, and costly recomputation. Third, multi-agent workflows exhibit sharp, structured bursts—often exceeding 50% spikes within a minute—that overwhelm serving capacity; because existing systems rely on reactive, smoothed autoscaling without awareness of workflow dependencies, they fail to anticipate these cascades, resulting in queuing and congestion across multiple LLMs.

*Shan Yu and Junyi Shu contributed equally.
†Junyi Shu is the corresponding author.

Key insight: Multi-agent workflows are predictable. Unlike general LLM serving which is fundamentally unpredictable, multi-agent workflows (as illustrated in Figure 1) present a remarkably different scenario. While the underlying models might be general-purpose, the fixed set of agent types and their specific roles within the workflow tightly constrain how these models are invoked.

In such workflows, high-level reasoning components—often powerful LLMs acting as architects, planners, or decomposers—process complex task requirements and break them down into a set of sub-tasks. Each sub-task is then handed off to a specialized worker agent (*e.g.*, an engineer or researcher) equipped with a particular (often smaller) model and a set of tools. Each agent typically performs a well-defined task and interacts with its model in a consistent manner. In many industrial deployments, a GPU cluster is dedicated to serving a small set of particular agentic applications (*e.g.*, coding assistants, deep-research systems).

As a result, requests from the same agent tend to follow similar usage patterns, introducing a substantial degree of *predictability in resource usage*.

Resource predictability in a multi-agent application arises from two key factors: *(1) predictable workflow* and *(2) predictable workload*. First, at serving time, the structure of the workflow—often represented as a graph in which nodes correspond to individual agents and edges denote data flow—is either fully determined (in static workflows) or follows repetitive patterns (in workflows that evolve dynamically through reasoning). This structural information reveals how requests propagate across agents and enables more informed optimization. For example, a worker agent that processes entirely new inputs each time derives minimal benefit from prefix caching. Likewise, when agents are deployed across multiple GPUs, scheduling can be designed to rapidly saturate the pipeline and maximize utilization, rather than relying on a naïve FCFS policy that simply prioritizes earlier-arriving requests.

Second, since each worker (non-orchestrator) agent is responsible for a specific task with a bounded level of complexity—where complex problems are typically decomposed into a collection of simpler subtasks—the output length and execution time of a request at each agent node typically follow relatively simple distributions, making them amenable to accurate estimation via dry runs or online profiling. Once characterized, these estimates can be leveraged during workflow execution to enable precise ahead-of-time cache warm-up and proactive model autoscaling prior to invoking an agent.

Pythia. Building on this insight, we designed Pythia, an agent-native serving system that systematically exploits workflow predictability to unlock optimizations previously considered intractable. Traditional serving engines treat agentic reasoning as an entirely opaque and dynamic process, limiting them to reactive, suboptimal resource allocation. In contrast, Pythia realizes proactive, global coordination by approximating a request's remaining execution time—inferred from its current position in the workflow graph and the expected output footprint of its specific agent role. By translating these structural insights into a priority mechanism that schedules requests based on anticipated completion times, Pythia enables fine-grained optimizations that fundamentally elude workflow-agnostic systems, significantly improving both end-to-end performance and cluster utilization.

Challenges. Fully leveraging these aspects of predictability for effectively serving multi-agent applications requires overcoming the following two challenges.

The first challenge is *how to inform the serving system about what aspects of the workflow are predictable*. For example, the workflow graph is a critical piece of information that needs to be conveyed to the serving system. However, as agent development becomes increasingly accessible to a broad audience (*e.g.*, as illustrated by OpenClaw [5]), many agent developers may not be expert programmers and therefore may not fully understand or be able to specify the underlying graph structure. This difficulty is further compounded by the fact that real-world multi-agent applications often generate dynamic graphs, where agent nodes are created by the orchestrator according to the complexity of the initial task.

To solve the problem, Pythia introduces our core contribution: an *example-driven synthesis* approach [6], which automatically synthesizes a graph representation from the profiling data. This approach builds on two important observations. First, most agentic applications have straightforward structures and a few runs with representative inputs can already generate enough profiling data that can help us construct the workflow graph. Second, although a dynamic graph cannot be statically described, it often consists of repetitive patterns (*e.g.*, consecutive Engineer-Executor interactions as in Figure 1a). By analyzing these patterns, the dynamic execution flow can be entirely expressed and predicted using *regular expressions*.

To this end, Pythia only requires agentic frameworks [5, 7–9] to pass lightweight information about application

and agent types via the `extra_body` payload already supported by OpenAI-compatible endpoints [10] without making any code change in user applications. At the gateway, Pythia employs a profiler to derive workflow and workload insights—including a regular expression capturing the workflow graph, as well as per-agent output length—and injects these annotations back into the payload for downstream consumption by the serving system.

The second challenge is *how to take advantage of such predictive information in the serving system.* Traditional LLM serving engines rely on a reactive paradigm that manages memory and schedules requests based on the current system state. Naively transitioning to a proactive model introduces the risk of severe performance penalties; aggressive, look-ahead decisions can easily trigger destructive resource

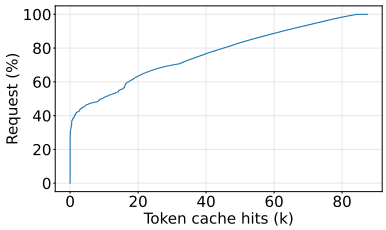


Figure 2: CDF depicting the percentages of requests with various numbers of token hits from our agent-serving platform.

thrashing and premature cache evictions if not carefully managed. To mitigate these risks and safely realize these optimizations, Pythia introduces three coordinated mechanisms—each aligned with the key opportunities (§2) identified in our trace analysis—to proactively optimize execution. At the node level, a *speculative cache manager* enhances reactive LRU with a Belady-inspired policy, leveraging workflow insights to prefetch shared contexts, evict transient tokens, and asynchronously warm caches to mask prefill latency. At the cluster level, a *lookahead request scheduler* uses predicted output lengths and workflow structure to balance heterogeneous workloads and prioritize execution, thereby reducing memory contention and head-of-line (HoL) blocking. Complementing these components, a *phase-adaptive autoscaler* anticipates structural workload shifts, such as fan-outs and phase transitions, to proactively scale models up or down, mitigating burst-induced queuing and improving overall resource efficiency.

Results. We have implemented Pythia on top of SGLang and evaluated it across representative multi-agent applications. Our results demonstrate up to 2.9× average JCT reduction with 1.96× throughput improvement. Pythia is currently in the phase of extensive testing and pilot deployment at a larger cluster scale internally for broader adoption.

2 Background and Motivation

2.1 Background

LLM serving. Today's LLM serving systems are architected around request-level and token-level execution [1–3, 11, 12]. Performance optimizations heavily target isolated micro-benchmarks, focusing on reducing Time-To-First-Token (TTFT) and Time-Between-Tokens (TBT) to accelerate prefill and decode phases. Crucially, the serving layer is usually assumed to have no prior knowledge of the output of ongoing requests and the incoming workload. Treating each request as an opaque, isolated entity forces the serving layer to rely on conservative, reactive policies. For example, request scheduling is typically relegated to FCFS queues, while prefix cache management relies strictly on reactive, recency-based eviction strategies.

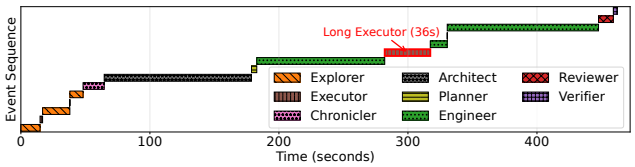


Figure 3: Timeline of the coding agent workflow: each bar represents an agent and "executor" represents command line execution.

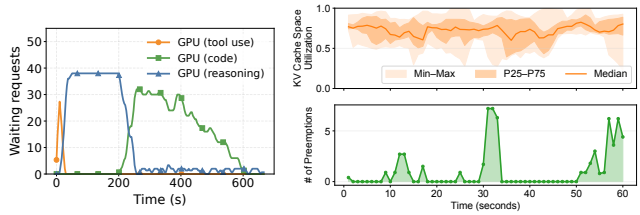
Multi-agent workflow. The interaction paradigm for LLMs is rapidly evolving from isolated, human-in-the-loop chat sessions to autonomous, goal-driven agents [5, 13, 14]. Driven by this shift, the share of inference requests generated by agents executing multi-step workflows has surged over the past six months. At our agent-serving platform, over **80%** of current requests are originated from agentic AI applications today. In a modern multi-agent system, high-level objectives are decomposed into actionable plans, with specific subtasks delegated to specialized worker LLMs and tools. These worker agents—driven by tailored system prompts and often backed by domain-specific models—handle distinct roles such as planning, coding, and reviewing. Traditional serving systems treat agent requests as normal LLM requests, handling them in a way that is agnostic to agent semantics.

Hierarchical prefix cache. Prefix caching stores computed key-value (KV) tensors to reduce redundant prefills. Modern systems balance latency and capacity using a three-tier hierarchy [15, 16]: L1 resides in scarce GPU HBM for immediate execution access; L2 acts as a spacious staging buffer in CPU host DRAM; and L3 utilizes shared storage to persist cross-engine contexts and minimize global cache misses.

2.2 Challenges in Serving LLMs for a Multi-Agent Workflow

As agentic AI increasingly dominates production workloads, MaaS providers including us, have introduced dedicated products with specialized billing models to support these applications [17–20]. Yet, a fundamental architectural mismatch persists: these services still rely on legacy, application-agnostic inference APIs. By flattening complex workflows into a series of opaque, independent requests, the serving layer is left entirely without knowledge of structural dependencies or distinct agent roles. To demonstrate how this black-box interface stifles efficiency, we analyzed large-scale production traces from our agent-serving service and conducted in-depth profiling of an internal multi-agent coding assistant deployed atop the service. This analysis reveals three major challenges where the lack of workflow visibility directly degrades system performance.

Challenge #1: Long prefill times due to unexpectedly low cache hit rates. While existing work often assumes that agentic AI applications naturally achieve high cache hit rates due to repeated prompt structures [21–23], our production traces



(a) Traffic imbalance between nodes in a workflow.

(b) Memory imbalance between nodes serving the same model.

Figure 4: Load imbalance and preemption when serving a multi-agent coding assistant.

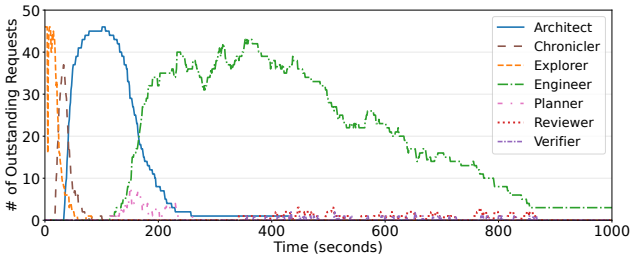


Figure 5: Outstanding requests of the multi-agent coding assistant for a batch of cron jobs.

reveal a starkly contradictory reality. Figure 2 reports the distribution of the requests with different numbers of token cache hits. As shown, more than 40% of requests have zero or very limited cache hit rates. This occurs because specialized agents rely on distinct system prompts, preventing prefix sharing across agent boundaries. Furthermore, the sequential execution of these workflows actively degrades cache retention. To illustrate this, Figure 3 details the execution timeline of the coding assistant. We make two observations: (1) only a few consecutive LLM requests can benefit from prefix cache; and (2) there can be long-running tool calls (*e.g.*, code compilation) which can take tens of seconds between these requests. By the time an agent is re-invoked, its previously cached prefixes have mostly been flushed by the intervening traffic.

Challenge #2: Wasted resources due to load imbalance and uncoordinated resource competition. Figure 4 shows a one-minute serving trace of coding agents on an 8-GPU cluster, revealing a system heavily constrained by memory pressure and frequent request preemptions. This inefficiency arises from mixing heterogeneous requests—ranging from lightweight Planner responses to long, thousand-token Architect outputs—without any awareness of the underlying workflow. We identify three key sources of this imbalance. First, naïve routing leads to significant load skew across data-parallel replicas of the same model. Second, resource allocation across models is uneven, leaving some saturated while others remain underutilized. Third, the lack of workflow awareness causes co-located agents to compete arbitrarily for shared GPU memory instead of being coordinated. As a result, overloaded replicas resort to preempting requests; without knowledge of expected output lengths, the system often evicts newer requests indiscriminately. These preempted requests must be recomputed from scratch, resulting in substantial computational waste and pronounced head-of-line blocking.

Challenge #3: Burst-induced queuing and congestion across LLMs. Multi-agent workflows inherently exhibit extreme structural volatility; in our agent-serving platform, we observe that scheduled automated tasks (*e.g.*, batched cron jobs) can trigger request spikes of up to **50.3%** for certain models within a single minute. As illustrated in Figure 5, these massive surges rapidly overwhelm the multiple LLMs backing the system. Crucially, the interdependent nature of multi-agent workflows fundamentally exacerbates this issue. A burst at an entry node (*e.g.*, an Explorer agent) does not remain isolated; instead, it triggers a cascading wave of requests that rapidly propagates through downstream agents (*e.g.*, Chronicler, Architect, Engineer). However, because existing serving systems rely on smoothed, long-term historical metrics and workflow-agnostic scaling, they cannot anticipate this inter-model propagation. Standard reactive autoscaling requires tens of seconds to add capacity—a delay during which the burst has often already shifted to the next model in the pipeline, leaving the system perpetually lagging behind the wavefront and causing queue depths to surge.

2.3 Clairvoyance in Agent-Native Serving

To address the challenges outlined above, we must first understand why today's inference services fail to mitigate them. Current LLM serving engines are designed around an implicit assumption: requests are generated by independent human users. Human chat traffic is characterized by uncorrelated intents, unpredictable arrival times, and arbitrary prompt contents. Serving systems are forced to operate as strict online algorithms—managing caches, scheduling requests, and scaling capacity reactively without any knowledge of the future.

However, agent traffic is generated by programs. Multi-agent workflows follow certain control flows, reuse established prompt templates, and exhibit access patterns dictated by their underlying source code. While the exact execution paths may be dynamically generated, the space of likely executions is far more constrained than human interactions. This programmatic nature implies a fundamental shift: if the serving system can capture the workflow's knowledge *a priori*, it can transition from reactive, online guesswork to proactive, near-offline optimization. Specifically, we identify three critical dimensions of such knowledge that can systematically mitigate the observed bottlenecks.

Opportunity #1: Workflow graph. Capturing the structural dependency graph of a multi-agent application unlocks multiple critical system optimizations. First, it identifies recurring agent roles, allowing the serving layer to strategically retain

Table 1: Output length statistics by agent role in the coding assistant.

Agent Role	Avg. Output Length	CV
Explorer	1924	0.45
Chronicler	912	0.26
Architect	1194	0.22
Planner	60	0.15
Engineer	3152	0.45
Reviewer	2620	0.18
Verifier	65	0.16

their reusable prefixes rather than naïvely evicting them under temporary memory pressure. Second, by understanding a request's approximate position within a workflow, the scheduler can estimate remaining execution times and dependencies, enabling critical-path-aware request scheduling. Finally, this structural awareness allows the system to proactively mitigate burst-induced queuing by anticipating imminent phase changes and fan-out patterns.

Acquiring this structural clairvoyance is highly feasible in production environments. While frameworks like AutoGen and LangGraph theoretically support fully dynamic, unconstrained agent group chats, real-world deployments rarely operate this way. Because unconstrained LLMs are susceptible to hallucination and task drift, enterprise applications typically impose stricter routing constraints and operational guardrails. For instance, our internal coding assistant enforces execution through hardcoded transition rules, using an LLM-based group chat selector only as a fallback for edge cases. As a result, production workflows tend to be stable, following highly probable execution paths with well-constrained iterative loops, leading to strongly concentrated transition probabilities between agent roles.

Opportunity #2: Output length of ongoing requests. Predicting the output length of active requests provides two critical operational advantages. First, it enables resource-aware scheduling to mitigate severe memory contention and load imbalance. By balancing the system based on predicted resource consumption rather than a naïve request count, the serving layer avoids the localized memory pressure that triggers catastrophic HoL blocking. Second, estimating the decode duration of an ongoing request provides a fine-grained timeline for when the request will arrive at the next agent in the workflow. This temporal awareness allows the serving system to fetch and warm the KV cache for the subsequent agents *just in time*.

The basis for this predictability lies in the strict division of labor across the workflow. Because individual agents are scoped to specific, narrow tasks, their generative behaviors naturally converge into stable, recognizable profiles. As detailed in Table 1, Chronicler, Planner, Reviewer, and Verifier reliably produce relatively consistent lengths with a low coefficient of variation (CV). Furthermore, while Explorer and Engineer's output lengths have higher variance due to their interactive nature, we observe a strong correlation between consecutive calls. Therefore, by observing an agent's semantic role and current context, the system can roughly bound a request's compute/memory footprint.

Opportunity #3: The composition and lineage of future prompts. Predicting the structure of upcoming prompts allows the serving system to proactively hide expensive prefill latency. Because specialized agents rely on distinct system prompts, they cannot natively share prefix caches; when a workflow transitions from one agent to another, the new request typically triggers a full, costly prefill computation. However, if the system predicts the next prompt, it can proactively compute and prepare the KV cache for the subsequent agent early, well before the actual request is issued. This proactive "pre-prefill" effectively eliminates the prefill bottleneck from the critical path.

The mechanics of multi-agent frameworks naturally expose the exact contents of these upcoming requests. A subsequent prompt is never a black box; it is reliably constructed by *concatenating known static content (*e.g.*, the agent's specific system instructions) with the explicit inputs and generative outputs of the dependent agents that executed earlier in the workflow*. Since the serving layer already tracks the workflow graph and holds the outputs from preceding steps, it possesses every ingredient required to accurately assemble and precompute the next prompt ahead of time.

Takeaway. The programmatic nature of multi-agent workloads introduces strong structural, temporal, and semantic predictability. However, this valuable *a priori* information is entirely lost across today's stateless application–system interface. To address the resulting cascade of issues—cache thrashing, load imbalance, and burst-driven queuing, the serving layer must be endowed with a form of clairvoyance.

3 Pythia Overview

Figure 6 depicts the system overview for Pythia.

Predictive information generation. The entry point to Pythia is an OpenAI-compatible API (§4.1) extended to support lightweight metadata annotations. These simple annotations serve to identify distinct workflows and specific agent roles directly within popular frameworks (*e.g.*, LangGraph or AutoGen). This minimal API extension is completely transparent to end users and requires no changes to the application logic.

In addition to collecting the structural information of the multi-agent workflow, the profiler (§4.1) acts as the analytical core of Pythia by operating asynchronously outside the critical path. By continuously ingesting annotated request/response logs, it extracts and maintains the overarching control flow graphs and program characteristics of different agents. Upon a new request, it uses the provided metadata to query these historical profiles, further annotating the requests with three critical execution estimates to downstream modules.

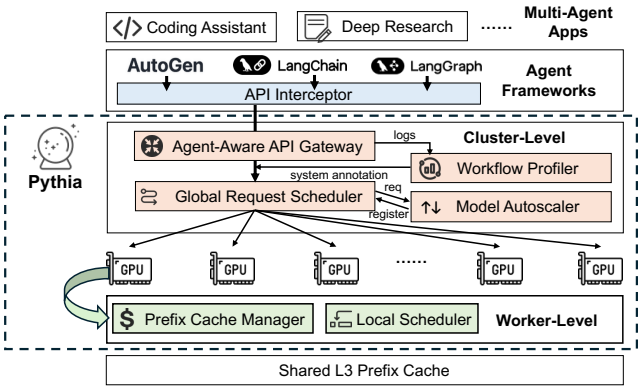


Figure 6: Pythia overview.

Predictive information consumption. Pythia uses the aforementioned predictive information at three distinct locations of the serving pipeline: per-node (agent) prefix cache management, global request scheduling, as well as per-node model scaling, which, respectively, correspond to the three major challenges faced by existing techniques (§2).

Operating at the node level, the cache manager (§4.2.1) mitigates prefix thrashing by shifting from reactive LRU eviction to a proactive Belady-like strategy. It leverages the extracted workflow information to accurately prefetch shared contexts and immediately drop transient tokens that are no longer needed. Furthermore, by anticipating the precise prompt composition of upcoming requests, it asynchronously warms the cache for the next probable agent, effectively hiding prefill latency before the subsequent request even arrives.

Operating at the cluster level, the request scheduler (§4.2.2) leverages the extracted program characteristics to execute resource-aware load balancing and precise request prioritization. By utilizing predicted output lengths, it evenly distributes heterogeneous workloads across data-parallel replicas. Furthermore, this profile-aware approach establishes better execution priorities, effectively mitigating localized memory contention and HoL blocking that drive request preemptions.

The per-node model autoscaler (§4.2.3) manages cluster-wide capacity by anticipating load changes rather than reacting to them. By evaluating the state of current active requests against the extracted workflow patterns, it forecasts imminent structural shifts, such as fan-outs and phase changes. This clairvoyance allows the system to proactively scale specialized models up before a burst hits, and scale them down when they are no longer needed, effectively alleviating burst-induced queuing.

We discuss how Pythia supports many distinct workflows, respond to pattern shifts, and handle adversarial behaviors in §5.

4 Pythia Design

4.1 Producing Predictive Information

To predict resource usage across serving stages, Pythia first extracts the semantic context of incoming requests. Our goal is to do so without imposing heavy integration overhead on application developers or violating the separation between application logic and serving infrastructure. To this end, we introduce a minimal, transparent API extension, coupled with an asynchronous background profiler that models workflow executions using classic program analysis techniques.

API abstraction. Standard inference APIs isolate requests, stripping away the overarching application context. To bridge this gap, Pythia leverages the `extra_body` parameter natively supported by OpenAI-compatible endpoints. When an agentic framework dispatches a request, it injects a lightweight `app_metadata` payload identifying the execution context. As shown in Listing 1, the framework only needs to supply three explicit identifiers:

1. `workflow_type_id` uniquely identifies an application (*e.g.*, `coding_assistant`), allowing the system to map the request to a specific structural profile.
2. `workflow_id` uniquely identifies a session, allowing the serving layer to track the ongoing lineage and contextual history of a specific execution instance.
3. `agent_id` uniquely identifies the current node or role in the workflow (*e.g.*, `engineer`), explicitly signaling which sub-task is currently executing.

We restrict the API extension to these three identifiers rather than requiring applications to pass explicitly defined dependency graphs or scheduling hints. Crucially, demanding a full workflow graph and agent characteristics upfront is impractical because they are often dynamically determined at runtime. Conversely, requiring application developers to manually tag requests with system-level hints (*e.g.*, "cache this prefix") violates the separation of concerns, burdening them with infrastructure resource management. By passing only the identity of workflows and agents, Pythia can easily correlate the request with historical patterns, leaving the burden of prediction and resource allocation entirely to the underlying serving system.

This minimal abstraction enables easy deployments. For framework developers building orchestration layers like LangGraph or AutoGen, integrating with Pythia requires no fundamental architectural changes. They simply need to configure a global API interceptor that automatically attaches the current session and agent variables to the `extra_body` payload before dispatch. Hence, developers building atop these frameworks reap the benefits of advanced system-level optimizations completely transparently, without needing to alter their application logic or write hardware-aware code.

Execution trace modeling via regular expressions. Once these annotated requests flow into the system, the workflow

Listing 1: API payload: the framework injects lightweight identity metadata via `app_metadata`; our Workflow Profiler intercepts this and injects actionable predictions via `sys_annotations` inside the same `extra_body` block before routing.

```
{
  "model": "meta-llama/Llama-3.1-8B-Instruct",
  "messages": [{"role": "user", "content": "..."}],
  "extra_body": {
    "app_metadata": {
      "workflow_type_id": "coding_assistant",
      "workflow_id": "<session_id>",
      "agent_id": "engineer"
    },
    // Injected internally by the Workflow Profiler
    "sys_annotations": {
      "predicted_output_len": [1000, 1300],
      "predicted_path_regex": "planner -> (explorer)^{||3,4} -> (
           engineer)^{3,6} -> reviewer -> (engineer^{2-4} ->
           reviewer)? -> verifier -> terminal",
      "prompt_composition": {"engineer": "You are a helpful
           engineer. Base code: ${req_12:request:[0,250]} Previous
            output: ${req_12:response:[0,1024]}"}
    }
  }
}
```

profiler asynchronously ingests the logs to reconstruct the application's global behavior. Drawing inspiration from dynamic program analysis [24] and program synthesis [6], we treat the sequence of agent invocations as an execution trace over an alphabet of agent roles. The profiler mines historical traces to construct a *Probabilistic Finite Automaton* (PFA).

The profiler does not merely predict the single next agent; it synthesizes the PFA into a *regular expression* (regex) that projects the entire anticipated execution graph. Consider a complex coding workflow: a `planner` delegates to several parallel `explorers`, followed by an `engineer` that iteratively refines code, a `reviewer`, and an optional rework loop. Rather than representing this as an infinite, unpredictable cycle—such as `(engineer | reviewer)`*—the profiler bounds the execution to its highly probable structural limits. As shown in the `predicted_path_regex` field in Listing 1, the system captures the parallel fan-out (`(explorer)`$^{||3,4}$), the bounded sequential self-reflection (`(engineer)`^{3,6}), and the optional single-retry backward edge (`(...)?`).

Furthermore, the profiler models the upcoming prompt structure as a deterministic template. Each placeholder within the `prompt_composition` acts as a strict memory pointer, consisting of three exact parameters: (1) the historical `request_id` of a dependent agent (*e.g.*, `req_1`), (2) a flag indicating whether to extract from that agent's input `request` or generative `response`, and (3) the specific token index `range` to retrieve. Listing 1 demonstrates how this template captures an agent stitching together the components.

Optimizing for the dominant percentile. A key insight in applying program analysis to agentic workflows is that modeling every possible execution edge case is counterproductive. Due to the inherent stochasticity of LLMs, workflows may occasionally exhibit anomalous transitions, enter chaotic error-recovery loops, or trigger runaway token generation. Attempting to fully account for such outliers leads to infinitely recursive representations and significantly over-provisioned memory reservations.

Instead, Pythia deliberately optimizes for the dominant percentile of executions in both control flow and resource usage. It applies trace filtering to prune low-probability transitions (*e.g.*, those occurring in less than 5% of cases), yielding a path expression that tightly captures the common execution graph. Likewise, rather than predicting exact output lengths or reserving for the worst-case context window, the profiler derives high-confidence intervals (*e.g.*, the 99th percentile of historical token usage) for each agent. This filtered, probabilistic view aligns with real-world deployments: it reflects the structured routing and guardrails that keep LLMs on stable paths, while bounding memory usage without overreacting to rare, long-tail behaviors.

Actionable request annotation. At inference time, the profiler operates at the gateway. When a request arrives, it extracts the `app_metadata` and queries the filtered models to retrieve the expected output length, resolve the regular expression describing the upcoming workflow graph, and bound the template for the next prompt. It then injects these insights into the `sys_annotations` field within the `extra_body` payload. This single interception step transforms an otherwise opaque request into a fully informed data structure, equipping downstream components with precise execution plans and memory hints for proactive caching, global scheduling, and cluster autoscaling.

4.2 Consuming Predictive Information

4.2.1 Speculative Cache Management. Equipped with the workflow graphs and prompt templates generated by the profiler, Pythia uses a speculative cache manager to mitigate the prefix thrashing and cache misses inherent to multi-agent workloads. By exploiting the future execution graphs embedded in a request's `sys_annotations`, our cache manager shifts memory management from reactive guesswork to a proactive, Belady-like strategy. As shown in Algorithm 1, this is achieved through a combination of early cache eviction and forward cache staging.

Early cache eviction (Lines 5-10). At the completion of each request (`OnRequestComplete`), the cache manager actively prunes the KV cache to prevent memory bloat and protect critical contexts. By parsing `predicted_path_regex`, it generates a deterministic set of future agent nodes. It then evaluates the lineage of every block in the cache against this set. Blocks belonging to transient steps that will not be revisited are classified as dead tokens and freed immediately, bypassing the delayed reclamation of standard LRU policies. Conversely, blocks whose lineage appears in the future path are explicitly marked for retention. Rather than simply trapping these retained blocks on the local worker, the manager

Algorithm 1 Speculative Cache Management.

Require: Current request R_{curr}, Hierarchical Cache C, GPU Compute Stream S_{bg}

```
1:  regex ← R_curr.predicted_path_regex
2:  future_nodes ← ParseRegex(regex)
3:  prompt_dict ← R_curr.prompt_composition
4:
5:  procedure OnRequestComplete
6:      for each block b ∈ C do
7:          if b.lineage ∉ future_nodes then
8:              Free(b)                    ▷ Dead tokens, drop immediately
9:          else if b.lineage ∈ future_nodes then
10:             RetainInHierarchy(b)
11:
12: procedure OnPrefetchRequested
13:     for each (agent, template) ∈ prompt_dict do
14:         //prompt_dict only has one entry
15:         prompt_next ← Assemble(template, History())
16:         if ExistsInCache(C, prompt_next) then
17:             PromoteToHost(prompt_next)
18:         else if IsIdle(GPU) then
19:             Enqueue(S_bg, ForwardPass(prompt_next))
```

aggressively writes them out to a shared L3 storage layer accessible by all inference engines. By proactively persisting future-relevant contexts in a global cache space, Pythia decouples cache residency from worker node affinity. This simplifies the cluster's global routing strategy, as subsequent requests in a multi-agent workflow can be scheduled on any available worker without suffering a hard cache miss.

Forward cache staging (Lines 12-19). To mitigate the prefill latency associated with transitioning between agents, the manager proactively prepares the exact context for the anticipated next step (`OnPrefetchRequested`). Guided by the `prompt_composition` template provided in the request annotations, the manager assembles the upcoming prompt by injecting the relevant inputs and outputs from the execution `History()`. Once the target prompt is assembled, the manager ensures its corresponding prefix cache is staged as close to the compute units as possible. If the cache for this prompt already exists in deeper, shared L3 storage, the manager promotes it to the L2 host DRAM (`PromoteToHost`).

Crucially, the system intentionally holds this prefetched context in the L2 host memory rather than immediately pushing it to the highly constrained GPU HBM. This design choice absorbs two practical realities: first, the profiler's prediction of the next agent is speculative and could be incorrect; second, the currently executing request may continue generating tokens for an extended period, and aggressively moving speculative data would prematurely pollute the limited GPU memory. This forward staging acts similarly to logistics positioning: it keeps the necessary memory blocks just one hop away in a less constrained environment, ready for an immediate, on-demand PCIe transfer when the actual request arrives.

Algorithm 2 Statistical Capacity-Based Routing.

Require: Request R, Candidate node N, Target OOM Threshold ϵ

```
1:
2:  procedure LRoute(R, N, ε)
3:      N_capable ← ∅, target ← null, max_headroom ← 0
4:      for each n ∈ N do
5:          P_oom ← Prob(Σ_{r∈n.active∪{R}} r.len > n.capacity)
6:          if P_oom ≤ ε then
7:              N_capable ← N_capable ∪ {n}
8:              headroom ← n.capacity – ExpectedSize(n)
9:              if headroom > max_headroom then
10:                 max_headroom ← headroom
11:                 target ← n
12:     return target                  ▷ Cache affinity as a tie-breaker
```

However, if the assembled prompt does not exist anywhere in the cache hierarchy, the manager attempts to generate it from scratch. The cache manager opportunistically capitalizes on the GPU's compute units; if it detects idle GPU cycles, it enqueues an asynchronous forward pass to pre-compute the prefix cache in the background. Through this combination of host-level staging and background pre-prefilling, the system maximizes the probability that the prefix is warmed before the framework ever issues the subsequent inference request.

4.2.2 Lookahead Request Scheduling.

With multi-agent request metadata extracted by the profiler and contexts managed by the cache manager, Pythia optimizes execution across the cluster through a two-tiered scheduling architecture: a statistical capacity-based router at the cluster level, and a graph-driven local scheduler at the worker level.

Statistical capacity routing. At the cluster ingress, the global router assigns incoming requests to specific worker nodes as shown in Algorithm 2. Because LLM output generation is inherently stochastic, relying on static maximum token lengths leads to severe memory underutilization, while relying on simple averages causes request preemptions due to out-of-memory (OOM). Pythia resolves this by leveraging the high-confidence intervals extracted by the profiler.

For each request r_i, the profiler provides an expected upper-bound length u_i derived from a high-confidence interval (*e.g.*, the 99th percentile, yielding an error probability $\alpha_i = 0.01$). This guarantees that the probability of the actual length L_i exceeding this bound is $P(L_i > u_i) \leq \alpha_i$. To avoid assuming any underlying statistical distribution, Pythia relies on a distribution-free strict capacity reservation. A candidate node with total KV capacity C is considered structurally capable if the sum of the confidence bounds for all active requests A and the new request R fits in memory:

$$\sum_{r_i \in A \cup \{R\}} u_i \leq C$$

When this reservation holds, an OOM event can only occur if the actual token generation violates these individual bounds. By applying the union bound (Boole's inequality), the router

Algorithm 3 Graph-Driven Global Priority Assignment.

Require: Cluster Replica Queues R_{queues}, Request r

procedure DOWNSTREAMIDLERISK(R_{queues}, r)
  $risk\_score \leftarrow 0$
  $future\_agents \leftarrow$ GetFutureAgents(r.regex)
  for each $a \in future\_agents$ **do**
    if $R_{queues}[a.model] <$ IDLE_THRESHOLD **then**
      $E[D_a] \leftarrow$ ExpectedDist(r.agent, a)
      $risk\_score \leftarrow risk\_score + \left(\frac{1}{E[D_m]}\right)$
  return $risk\_score$

procedure SETPRIORITY(r, R_{queues})
  $E[D_{remain}] \leftarrow$ ExpectedRemainingDistance(r.regex)
  $S_{completion} \leftarrow \frac{1}{E[D_{remain}]}$
  $S_{unblock} \leftarrow$ DownstreamIdleRisk(R_{queues}, r)
  r.base_priority $\leftarrow (\omega_1 \times S_{completion}) + (\omega_2 \times S_{unblock})$

calculates the worst-case joint probability of an OOM event, P_{oom}, which is strictly bounded by the sum of the individual error probabilities:

$$P_{oom} \leq \sum_{r_i \in A \cup \{R\}} P(L_i > u_i) \leq \sum_{r_i \in A \cup \{R\}} \alpha_i$$

The router enforces that this distribution-free failure probability remains below a strict, user-defined safety threshold ($P_{oom} \leq \epsilon$). Among the mathematically safe nodes, it routes the request to the one with the maximum expected headroom ($C - \sum u_i$). While not strictly enforced in the primary capacity bounds, the router uses forward cache staging (§4.2.1) as a deterministic tie-breaker. If multiple nodes offer safe capacity profiles, the router preferentially selects the node that already holds the required L2 host cache, gracefully marrying strict statistical load balancing with instantaneous cache hits.

Graph-driven global priority assignment. In a multi-agent environment where multiple agents share the same underlying model replicas, standard FCFS scheduling frequently causes cluster-wide starvation. It ignores workflow dependencies, allowing newly spawned, long-running agents to block final-stage agents, which severely delays final-stage completion. To resolve this, Pythia implements a global arbitration strategy (`SetPriority` in Algorithm 3). Before a request is routed to a worker, the global scheduler assigns it a static `base_priority` score.

This score explicitly balances two cluster-wide execution goals. First, it accelerates job completion by prioritizing requests closest to the end of their workflow. Because workflow paths are defined by probabilistic regular expressions rather than deterministic graphs, the scheduler calculates the mathematical expectation of the remaining distance to the terminal node ($E[D_{remain}]$). The priority score scales inversely with this expectation, ensuring that agents statistically closer to finishing (*e.g.*, a final verifier) inherently preempt early-stage agents to quickly flush completed jobs from the system.

Second, the scheduler actively monitors the real-time queue depths of all model replicas across the cluster (`DownstreamIdleRisk`). If a model replica serving future agents is at risk of becoming idle, the priority of any upstream request that will eventually unblock it is boosted. Crucially, this boost is weighted inversely by the expected distance between the current agent and the starved model ($E[D_m]$). A request that is only one step away from feeding an idle GPU receives a higher priority spike, whereas a request ten steps away receives a negligible boost. This ensures that no specialized inference worker sits idle while waiting for a stalled upstream dependency.

Periodical local batch creation. Once routed to a worker, execution is governed by iteration-level dispatching. At the start of every scheduling window, the local worker dynamically recalculates the effective priority of all requests by adding an aging factor—scaled by accumulated wait time—to the global `base_priority`. The worker then sorts the pool and dispatches the highest-priority subset that fits within memory bounds. This ensures early-stage agents are mathematically protected from permanent starvation while preserving the global graph's intended execution order.

Priority-aware preemption recovery. When tail-latency generations occasionally force a worker into memory exhaustion, Pythia overrides standard FCFS eviction policies. Instead of blindly appending evicted requests to the back of the queue, the local scheduler specifically targets the active request with the lowest dynamically calculated priority (*e.g.*, an early-stage agent) as the preemption victim. This request is paused and re-inserted into the local queue. Because the queue is strictly re-sorted before every iteration, critical-path executions remain completely unperturbed by localized memory spikes, leading to high effective resource utilization and minimized workflow-level waiting time.

4.2.3 Phase-Adaptive Autoscaling.

While lookahead request scheduling (§4.2.2) mitigates load imbalance, it cannot create new capacity. Agentic applications frequently exhibit extreme burstiness driven by synchronized triggers (*e.g.*, cron jobs) or massive structural fan-outs. During these dramatic shifts, scheduling alone cannot prevent queue saturation; the cluster must physically scale its model replicas.

Current serverless LLM solutions rely on reactive autoscaling, which fails during the aggressive, instantaneous phase shifts typical of multi-agent execution. Because loading massive model weights into GPU memory takes tens of seconds, reactive provisioning is not merely too slow—it is actively detrimental. In a constrained cluster, spinning up a new model often forces the blind eviction of existing ones. By the time the new replica is finally ready, the transient burst has often subsided, leaving the system with an idle model and a destroyed cache of other essential models, triggering a destructive cascade of cold starts.

Algorithm 4 Phase-Adaptive Autoscaling.

Require: Active Requests A, Current Replicas R_{curr}, Look-Ahead Horizon H

procedure EstimateImminentDemand(A, H)
  $D \leftarrow$ InitializeZeroMap()
  for each $r \in A$ **do**
    $imminent\_agents \leftarrow$ ProjectGraph(r.regex, H)
    for each $a \in imminent\_agents$ **do**
      $D[a.\text{model}]$+=EstimatedLoad(a)
  return D

procedure AutoscaleCluster
  $D \leftarrow$ EstimateImminentDemand(A, H)
  $R' \leftarrow$ InitializeMap()
  for each $m \in$ RegisteredModels() **do**
    $R'[m] \leftarrow$ EstimateReplicas($D[m]$)
  for each $m \in$ RegisteredModels() **do**
    if $R'[m] < R[m]$ **then**
      ScaleDownOnIdle($m, R[m] - R'[m]$)
  for each $m \in$ RegisteredModels() **do**
    if $R'[m] > R[m]$ **then**
      ScaleUp($m, R'[m] - R[m]$)

Pythia overcomes this by leveraging the regular path expressions extracted by the workflow profiler to introduce a predictive control plane. Instead of reacting to queue buildup, it forecasts imminent bursts and proactively provisions replicas before the traffic shift occurs, entirely avoiding the reactive resource-thrashing cycle.

Proactive scale-up for structural shifts. Rather than waiting for queues to spike, Pythia continuously evaluates the state of all active requests against their workflow graph to forecast imminent demand (`EstimateImminentDemand` in Algorithm 4). By projecting the execution graph forward by a defined step horizon (H), the autoscaler calculates which models will be extensively utilized in the immediate future.

For example, if the cluster currently has fifty active requests executing in a `planner` phase, and the underlying regex dictates a transition of `planner` $\rightarrow$ `(explorer)`$^{||10}$, the autoscaler mathematically knows that 500 requests for the `explorer` model are arriving in the near future. Before the `planner` agents even finish generating their outputs, Pythia proactively triggers the background provisioning of additional `explorer` model replicas. By the time the framework actually dispatches the massive fan-out burst, the new GPU nodes are already warmed and ready, effectively alleviating burst-induced queuing and hiding cold-start latencies.

Predictive scale-down and resource reclamation. In a resource-constrained cluster, scaling up one specialized model often requires evicting another. Standard systems rely on static keep-alive timeouts (*e.g.*, waiting 5 minutes before spinning down an idle model), which hoards precious GPU memory during critical phase shifts. Pythia accelerates this reclamation process through predictive scale-down.

Because the autoscaler possesses the full structural map of the active workloads, it can identify which models will *not* be called in the near future. This includes models powering agents that are strictly in the past (*e.g.*, the `planner` once all workflows have transitioned to the `explorer` phase) and models powering agents that are topologically too far away to warrant immediate warming (*e.g.*, a `final_verifier` that is many steps away from the current execution frontier). To maximize the opportunity for a safe scale-down, the autoscaler coordinates directly with the global scheduler. Once a model is identified as obsolete for the immediate look-ahead horizon, the autoscaler signals the scheduler to cease load-balancing new requests to those specific replicas. This active routing exclusion allows the replicas to drain rapidly. As soon as a targeted replica's immediate queue is empty, Pythia forcefully deprovisions it without waiting for an arbitrary timeout. This aggressive, graph-driven reclamation ensures that cluster resources are continuously freed and reallocated to support the models required for the immediate execution phase.

5 Discussion

Support for many distinct agentic workflows. Currently, Pythia is primarily designed for environments running a small, targeted set of agentic applications. When scaling the system to concurrently serve a massive variety of distinct workflows, Pythia successfully maintains its core advantage of minimizing the global average JCT. However, in such a highly heterogeneous environment, relying on our current, relatively simple scheduling heuristics may inadvertently sacrifice the performance of certain individual workflows to maximize overall throughput. To effectively balance cluster-wide resource efficiency with other user requirements (*e.g.*, per-workflow SLOs or fairness guarantees), Pythia's lookahead scheduler can be naturally extended. By directly incorporating per-workflow priority weights and explicit latency goals into our scheduling algorithm, the system can make more nuanced, SLO-aware allocation decisions that protect mission-critical workflows while still preserving aggregate optimizations.

Cold starts and workflow drift. Pythia's reliance on historical traces introduces challenges during initial deployment (cold starts) and application updates (workflow drift). To prevent insufficient or stale data from triggering unsafe routing, Pythia can pair explicit API versioning with a "shadow profiling" phase. When a new workflow version is detected via `app_metadata`, the gateway temporarily routes its traffic through a standard, reactive fallback while profiling it in the background. Once safe statistical confidence intervals are established, the system seamlessly promotes the workflow to the proactive execution tier, ensuring stability across lifecycle transitions.

Open-ended and adversarial workflows. Pythia relies on the structural predictability typical of enterprise applications, which apply routing guardrails and assume trusted inputs. In fully unconstrained or adversarial scenarios, execution graphs devolve into unpredictable meshes, rendering our proactive optimizations ineffective. To accommodate such environments, future deployments could adopt a hybrid architecture: Pythia would serve verified, predictable workflows, while offloading highly deviant or adversarial requests to an isolated, reactively scheduled fallback environment.

6 Implementation

We implemented Pythia on top of SGLang. The Workflow Profiler and Global Request Scheduler replace the default SGLang router, intercepting standard OpenAI-compatible requests to extract `app_metadata`, injecting `sys_annotations`, and evaluating the statistical capacity bounds (§4.2.2) before routing. At the engine level, we integrated the Speculative Cache Manager (§4.2.1) into SGLang's Hierarchical Cache. Furthermore, we replaced SGLang's default scheduler with our graph-driven iteration dispatcher, ensuring that requests are strictly sorted by their dynamic `priority` score at the start of every iteration. Finally, the Phase-Adaptive Autoscaler (§4.2.3) runs as a periodic daemon on the control plane, provisioning and deprovisioning SGLang model replicas across the GPU fleet based on the projected demand.

7 Evaluation

Testbed. We have evaluated Pythia on a server equipped with two AMD EPYC 7J13 CPUs, 1.7TB of DRAM, and eight NVIDIA A100 GPUs (80GB HBM each) interconnected via NVLink.

Baselines. We compare Pythia with a comprehensive suite of state-of-the-art LLM serving systems and agent-aware frameworks:

- **vLLM [1]:** vLLM (v0.17.1) deployed with vLLM Production Stack [25] (v0.1.10) router.
- **SGLang [2]:** SGLang (v0.5.9) utilized alongside the gateway.
- **Autellix [26]:** A specialized agentic LLM serving system that provides integrated scheduling optimizations across the router and inference engine. Because Autellix is not open-sourced, we implemented the same routing and scheduling policy on top of vLLM (v0.17.1).
- **Continuum [23]:** A single-node serving system optimized for ReAct [27] agents; we use it with the vLLM Production Stack router (v0.1.10) for fair comparison. We only allow agents that can potentially have cache hits to pin their cache.
- **ThunderAgent [22]:** A program-aware global LLM request scheduler for agentic applications, which we use SGLang (v0.5.9) as the underlying inference engine. Because ThunderAgent is initially designed for a single model, we changed its single-queue design to one queue per model. We also changed its scheduling interval from the default 5 seconds to 0.5 seconds to reduce unnecessary queuing delay.

All systems use Mooncake [15] for the L3 prefix cache, while vLLM-based baselines are integrated via LMCache [16].

Workloads. We evaluate Pythia using two representative agentic workflows ported to the AutoGen [7] framework: a proprietary internal coding assistant and a Deep Research Agent [28]. Using these applications, we execute two demanding benchmarks: **SWE-bench Pro** [29], a rigorous coding benchmark mirroring complex production patterns, and **Deep Research Bench** [30], designed to evaluate PhD-level research tasks.

Our evaluation utilizes real-world traces collected from these applications across a wide portfolio of frontier models—including those from Anthropic, OpenAI, and Google—assigned to various agent roles. To ensure high-fidelity simulation of agentic behavior, our trace collection and evaluation involve live integration with real execution environments and production-grade search APIs.

To facilitate deployment on our testbed, we map these frontier models to smaller-scale models (3B–14B) from the Qwen and Llama families, employing tensor parallelism (TP) degrees ranging from one to four.

7.1 End-to-End Results

We first evaluated Pythia against all five baseline systems under different numbers of concurrent workflows. We compare average JCT, P95 JCT, and throughput (output tokens generated per second).

Average JCT. Across all workloads, Pythia reduces average JCT by 1.38–2.9× at peak concurrency. Under low load, gains stem primarily from our speculative cache management hiding prefill latency. As density increases, Pythia's graph-driven global arbitration prevents the catastrophic HoL blocking that plagues baselines—particularly in heterogeneous mixed workloads. While partially agent-aware systems (Continuum, Autellix) generally outperform agnostic engines, and ThunderAgent shows some resilience under heavy Deep Research loads, Pythia uniquely maintains steady performance across all scenarios.

P95 JCT. While approximate SRTF scheduling typically degrades tail latency by starving early-stage requests, Pythia defies this trade-off, reducing P95 JCT by 1.15–2.02× under high concurrency. Instead of simply favoring short outputs, Pythia dynamically boosts upstream requests that unblock downstream models. Paired with priority aging, this holistic

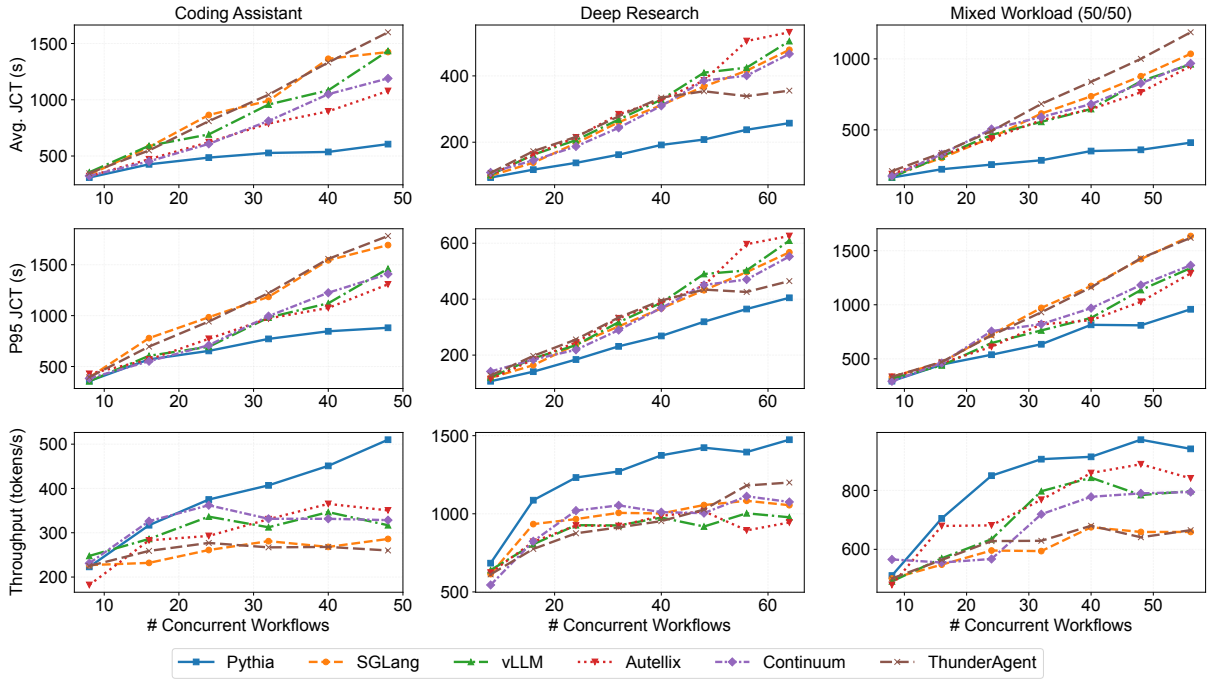
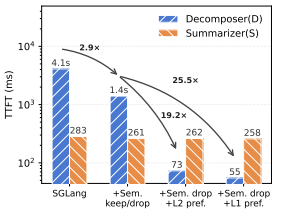
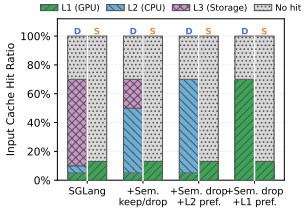


Figure 7: End-to-end experiments.



(a) Pythia's semantic-aware keep/drop and speculative prefetch improve TTFT.



(b) Input cache hit ratio broken down by tier.

Figure 8: Workflow-aware speculative prefix cache management improves TTFT latency and prefix cache hit ratio from the lower cache tier.

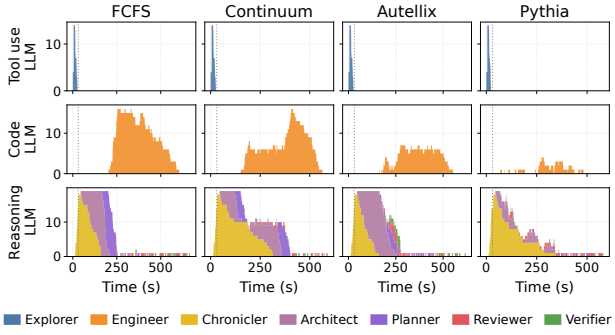


Figure 9: GPU queue dynamics under different scheduling policies for coding assistant. Pythia achieves lower queuing delay and more balanced GPU utilizations.

strategy prevents complex multi-step workflows from languishing in queues, ensuring highly predictable tail performance even under extreme load.

Throughput. Pythia delivers a 1.12–1.96× throughput improvement across all evaluated workloads. This gain stems from two complementary mechanisms: the statistical capacity router leverages output length predictions to evenly distribute load and minimize localized memory contention, while the graph-driven scheduler prioritizes critical-path tasks to prevent downstream model idleness, ultimately maximizing aggregate cluster utilization.

7.2 Performance Analysis

Cache efficiency. We take the deep research workflow as an example to show the effectiveness of Pythia's speculative cache management. A deep research workflow can repeat a few turns where Decomposer shares context across turns but Summarizer receives a large amount of new context from Researchers. As shown in Figure 8, collaboratively applying proactive keep/drop actions and timely prefetch, TTFT of Decomposer is reduced by 25.5× with cached tokens reused from L1/L2 cache.

Comparison of different scheduling algorithms. To analyze the effectiveness of our graph-driven global priority assignment, we break down the queuing dynamics when serving bursts of coding-assistant workflows. Figure 9 shows that all three baselines build substantially larger queues on both the codeLLM and the shared reasoning LLM. FCFS blindly serves earlier-arriving requests, causing upstream reasoning requests to block later-stage review and verification, which

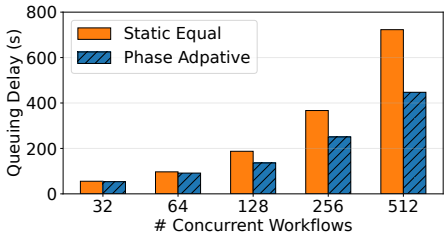


Figure 10: Queuing delay under different scaling strategies.

delays completion and eventually creates backlog on the codeLLM. Continuum similarly preserves job-level arrival order across the workflow, slowing the drain of requests that are already near completion. Autellix improves over FCFS, but because it is unaware of workflow position and downstream bottlenecks, it still over-admits upstream requests and leaves significant queue buildup. In contrast, Pythia uses workflow structure and remaining work to prioritize completion-critical requests and throttle upstream injection when bottlenecks emerge, thereby preventing queue waves across GPUs and yielding lower JCT and higher throughput.

Effectiveness of autoscaling. Figure 10 compares Pythia with static resource allocation as we vary the number of concurrent workflows. Pythia reduces queuing delay by 1.49× relative to static model placement by proactively rebalancing resources via model autoscaling. The key advantage is that Pythia predicts workflow phase load patterns and allocates additional model instances to stages undergoing bursty request arrivals. Static placement, by contrast, cannot respond to these shifting bottlenecks. As concurrency increases, more workflows simultaneously traverse different stages, amplifying burstiness and causing long request queues under fixed provisioning.

7.3 Ablation Studies

Sensitivity to load burstiness. As agentic workflows often arrive in a batch, it's important for the underlying serving system to saturate bursts. In Figure 11, we show average JCT and throughput under different CVs for arrival interval (0: stable, 1: Poisson-like, >1: bursty). While Pythia's performance is relatively consistent, turbulence is observed across baselines.

Accuracy of the predictor. In Figure 12, we show the predicted output length intervals against the ground truth on the test set. For both coding assistant and deep research, Pythia's predictor is able to provide well-bounded predictions across most data points except a few outliers. Pythia can also accurately provide workflow graphs of these applications whose patterns are generally stable.

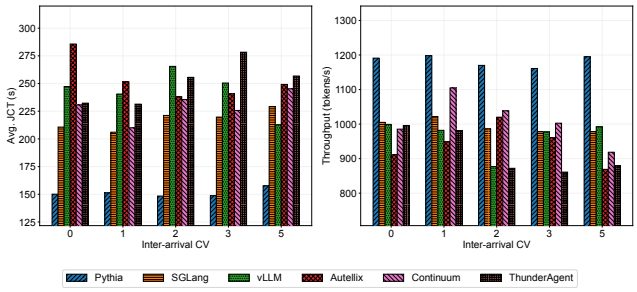
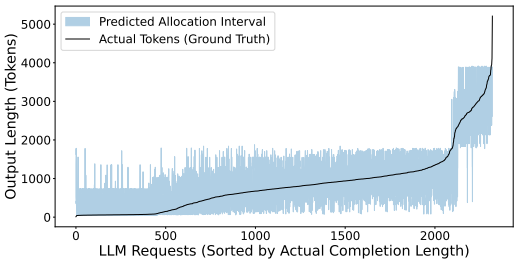
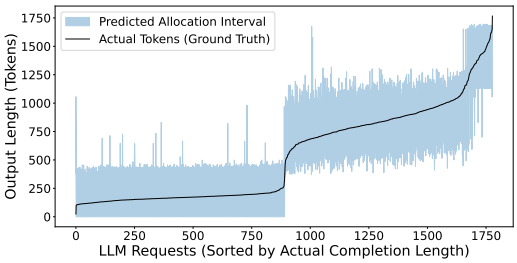


Figure 11: Sensitivity to load burstiness.



(a) Coding assistant.



(b) Deep research.

Figure 12: Predicted output lengths vs. actual lengths.

8 Related Work

LLM serving systems. Modern serving engines are highly optimized across multiple layers. For scheduling, Orca [11] introduces iteration-level scheduling to handle varying request lengths, while FastServe [31] uses preemption to mitigate HoL blocking. To manage heterogeneous execution stages, systems explore either disaggregating [12, 32–34] (*e.g.*, Prefill–Decode disaggregation) or co-locating [3, 35] (*e.g.*, Chunked Prefill) these phases. Memory management is equally critical: PagedAttention [1] uses OS virtual memory abstractions to reduce GPU memory fragmentation, while other works accelerate the prefill stage by exploiting caching and memory hierarchies [2, 15, 16, 36, 37]. At the operator level, libraries like FlashInfer [38] and FlashAttention [39] provide the foundational high-performance attention kernels. Finally, multi-LLM serving systems focus on spatial and temporal multiplexing to maximize the density of models co-located on shared hardware [4, 40–43]. Unlike Pythia, which optimizes

for end-to-end agentic workflows, these systems primarily target granular request- and token-level metrics.

System optimizations targeting agentic AI. Recent work has recognized that serving agentic AI applications introduces new system challenges beyond traditional LLM inference, motivating dedicated optimizations in specialized runtimes and serving frameworks. Pancanke [44] designs an efficient multi-tier agent memory system, and DualPath [21] optimizes the network bottleneck of remote prefix cache for agentic workloads. Systems such as ThunderAgent [22], Ayo [45], and Autellix [26] focus on improving request orchestration of long-horizon complex agent pipelines, while KVFlow [46] and Continuum [23] propose memory optimizations through cross-step prefix cache sharing in order to reduce redundant prefills. However, these systems do not fully exploit predictable program semantics for fine-grained serving optimizations as Pythia does.

9 Conclusion

The structured nature of multi-agent LLM workloads fundamentally changes the serving design space. By exposing workflow semantics, Pythia transforms opaque requests into predictable executions, unlocking proactive optimizations impossible in conventional systems. Our evaluation proves that leveraging this predictability is critical to overcoming cache inefficiency, resource imbalance, and burst-induced congestion.

References

[1] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *ACM SOSP*, 2023.

[2] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: Efficient Execution of Structured Language Model Programs," in *NeurIPS*, 2024.

[3] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve," in *USENIX OSDI*, 2024.

[4] S. Yu, J. Xing, Y. Qiao, M. Ma, Y. Li, Y. Wang, S. Yang, Z. Xie, S. Cao, K. Bao, I. Stoica, H. Xu, and Y. Sheng, "Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving," 2025.

[5] P. Steinberger, "OpenClaw." https://openclaw.ai/, 2026. Retrieved Mar 9, 2026.

[6] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," GPCE 2016, 2016.

[7] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations," in *Conference on Language Modeling*, 2024.

[8] "LangGraph." https://www.langchain.com/langgraph, 2026. Retrieved Mar 9, 2026.

[9] "LangChain." https://www.langchain.com, 2026. Retrieved Mar 9, 2026.

[10] "OpenAI Python API library." https://github.com/openai/openai-python, 2026. Retrieved Mar 9, 2026.

[11] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *USENIX OSDI*, 2022.

[12] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving," in *USENIX OSDI*, 2024.

[13] "Claude Code bypassPermission Mode." https://code.claude.com/docs/en/permission-modes, 2026. Retrieved Mar 9, 2026.

[14] "Codex Command Line Options." https://developers.openai.com/codex/cli/reference, 2026. Retrieved Mar 9, 2026.

[15] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot," in *USENIX FAST*, 2025.

[16] Y. Liu, Y. Cheng, J. Yao, Y. An, X. Chen, S. Feng, Y. Huang, S. Shen, R. Zhang, K. Du, and J. Jiang, "LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference," 2025.

[17] "Coding Plan Overview." https://www.alibabacloud.com/help/en/model-studio/coding-plan, 2026. Retrieved Mar 9, 2026.

[18] "ModelArk Coding Plan." https://www.byteplus.com/en/activity/codingplan, 2026. Retrieved Mar 9, 2026.

[19] "MiniMax Token Plan." https://platform.minimax.io/subscribe/token-plan, 2026. Retrieved Mar 9, 2026.

[20] "GLM Coding Plan." https://z.ai/subscribe, 2026. Retrieved Mar 9, 2026.

[21] Y. Wu, S. Chen, Y. Zhong, R. Huang, Y. Tan, W. Zhang, L. Zhang, S. Zhou, Y. Liu, S. Zhou, M. Zhang, X. Jin, and P. Huang, "DualPath: Breaking the Storage Bandwidth Bottleneck in Agentic LLM Inference," 2026.

[22] H. Kang, Z. Li, X. Yang, W. Xu, Y. Chen, J. Wang, B. Chen, T. Krishna, C. Xu, and S. Arora, "ThunderAgent: A Simple, Fast and Program-Aware Agentic Inference System," 2026.

[23] H. Li, Q. Mang, R. He, Q. Zhang, H. Mao, X. Chen, H. Zhou, A. Cheung, J. Gonzalez, and I. Stoica, "Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live," 2026.

[24] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *ACM POPL*, 2002.

[25] "vLLM Production Stack." https://github.com/vllm-project/production-stack, 2026. Retrieved Mar 9, 2026.

[26] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez, and I. Stoica, "Autellix: An Efficient Serving Engine for LLM Agents as General Programs," 2025.

[27] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in *ICLR*, 2023.

[28] A. Prabhakar, R. Ram, Z. Chen, S. Savarese, F. Wang, C. Xiong, H. Wang, and W. Yao, "Enterprise Deep Research: Steerable Multi-Agent Deep Research for Enterprise Analytics," 2025.

[29] X. Deng, J. Da, E. Pan, Y. Y. He, C. Ide, K. Garg, N. Lauffer, A. Park, N. Pasari, C. Rane, K. Sampath, M. Krishnan, S. Kundurthy, S. Hendryx, Z. Wang, V. Bharadwaj, J. Holm, R. Aluri, C. B. C. Zhang, N. Jacobson, B. Liu, and B. Kenstler, "SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?," 2025.

[30] M. Du, B. Xu, C. Zhu, X. Wang, and Z. Mao, "DeepResearch Bench: A Comprehensive Benchmark for Deep Research Agents," 2025.

[31] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast Distributed Inference Serving for Large Language Models," 2024.

[32] P. Patel, E. Choukse, C. Zhang, A. Shah, I. n. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient Generative LLM Inference Using Phase Splitting," in *ACM/IEEE ISCA*, 2025.

[33] F. Strati, S. Mcallister, A. Phanishayee, J. Tarnawski, and A. Klimovic, "DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving," in *ICML*, 2024.

[34] R. Zhu, Z. Jiang, C. Jin, P. Wu, C. A. Stuardo, D. Wang, X. Zhang, H. Zhou, H. Wei, Y. Cheng, J. Xiao, X. Zhang, L. Liu, H. Lin, L.-W.

Chang, J. Ye, X. Yu, X. Liu, X. Jin, and X. Liu, "MegaScale-Infer: Efficient Mixture-of-Experts Model Serving with Disaggregated Expert Parallelism," in *ACM SIGCOMM*, 2025.

[35] K. Zhu, Y. Gao, Y. Zhao, L. Zhao, G. Zuo, Y. Gu, D. Xie, T. Tang, Q. Xu, Z. Ye, K. Kamahori, C.-Y. Lin, Z. Wang, S. Wang, A. Krishnamurthy, and B. Kasikci, "NanoFlow: towards optimal large language model serving throughput," in *USENIX OSDI*, 2025.

[36] S. Agarwal, A. Mao, A. Akella, and S. Venkataraman, "Symphony: Improving memory management for llm inference workloads," 2024.

[37] Z. Xie, Z. Xu, M. Zhao, Y. An, V. S. Mailthody, S. Mahlke, M. Garland, and C. Kozyrakis, "Strata: Hierarchical context caching for long context language model serving," 2025.

[38] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, "FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving," in *MLSys*, 2025.

[39] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," in *NeurIPS*, 2022.

[40] Y. Xiang, X. Li, K. Qian, Y. Yang, D. Zhu, W. Yu, E. Zhai, X. Liu, X. Jin, and J. Zhou, "Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market," in *ACM SOSP*, 2025.

[41] J. Duan, R. Lu, H. Duanmu, X. Li, X. Zhang, D. Lin, I. Stoica, and H. Zhang, "MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving," in *ICML*, 2024.

[42] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "ServerlessLLM: Low-Latency Serverless Inference for Large Language Models," in *USENIX OSDI*, 2024.

[43] C. Lou, S. Qi, C. Jin, D. Nie, H. Yang, Y. Ding, X. Liu, and X. Jin, "HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds," 2025.

[44] Z. Hu, Z. Pan, P. Kaur, V. Murthy, Z. Yu, Y. Guan, Z. Wang, S. Swanson, and Y. Ding, "Pancake: Hierarchical Memory System for Multi-Agent LLM Serving," 2026.

[45] X. Tan, Y. Jiang, Y. Yang, and H. Xu, "Towards End-to-End Optimization of LLM-based Applications with Ayo," in *ACM ASPLOS*, 2025.

[46] Z. Pan, A. Patel, Z. Hu, Y. Shen, Y. Guan, W.-L. Li, L. Qin, Y. Wang, and Y. Ding, "KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows," 2025.