Act While Thinking: Accelerating LLM Agents via Pattern-Aware Speculative Tool Execution

Yifan Sui
Shanghai Jiao Tong University
Shanghai, China
suiyifan@sjtu.edu.cn

Han Zhao
Shanghai Jiao Tong University
Shanghai, China
zhaohan_miven@sjtu.edu.cn

Rui Ma
Microsoft Research
Beijing, China
mrui@microsoft.com

Zhiyuan He
Microsoft Research
Beijing, China
zhiyuhe@microsoft.com

Hao Wang
Stevens Institute of Technology
Hoboken, NJ, USA
hwang9@stevens.edu

Jianxun Li
Shanghai Jiao Tong University
Shanghai, China
lijx@sjtu.edu.cn

Yuqing Yang
Microsoft Research
Redmond, WA, USA
yuqyang@microsoft.com

Abstract

LLM-powered agents are emerging as a dominant paradigm for autonomous task solving. Unlike standard inference workloads, agents operate in a strictly serial "LLM-tool" loop, where the LLM must wait for external tool execution at every step. This execution model introduces severe latency bottlenecks. To address this problem, we propose PASTE, a **P**attern-**A**ware **S**peculative **T**ool **E**xecution method designed to hide tool latency through speculation. PASTE is based on the insight that although agent requests are semantically diverse, they exhibit stable application level control flows (recurring tool-call sequences) and predictable data dependencies (parameter passing between tools). By exploiting these properties, PASTE improves agent serving performance through speculative tool execution. Experimental results against state of the art baselines show that PASTE reduces average task completion time by 48.5% and improves tool execution throughput by 1.8×.

1 Introduction

As large language models (LLMs) continue to improve their reasoning and understanding capabilities, LLM-powered agents have emerged as a major research focus [1–4, 31]. As exemplified by OpenAI Deep Research [33] and Manus [28], the LLM acts as the "brain" of the agent. It decomposes a complex task into a sequence of sub-steps and invokes external tools at each step to interact with the outside world. By lowering the barrier to using powerful and complex tools, agents are widely expected to drive the productivity revolution. As a result, many companies have invested tens of billions of dollars in this line of research and development [14, 26, 55].

Motivation. Figure 1 illustrates a typical workflow of a modern LLM-powered Agent. As shown, the LLM alternates between generating text and making external tool calls. These two steps operate strictly serially, as they have inherent dependencies. Our experimental results in §2.2.2 show that tool execution accounts for 35% to 61% of total request time. This execution model forces LLMs

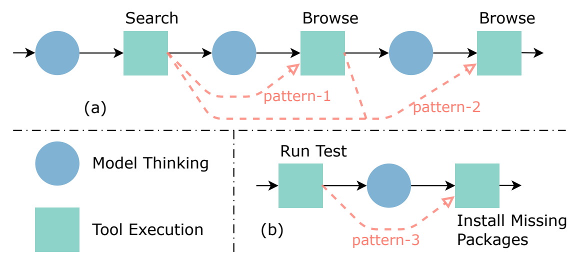


Figure 1: Example patterns observed in execution of (a) deep research agents and (b) coding agents. Patterns indicate not only tool invocations in the near future but also latent data dependencies between tool calls (orange lines in this figure).

to hold expensive memory resources, yet still delivers long end-to-end latency. These inefficiencies significantly hinder the rapid deployment and continued evolution of agent serving systems.

Limitation of state-of-the-art approaches. Although numerous existing works [13, 27, 39] have attempted to optimize the startup of agent serving systems, they either focus solely on tool startup optimization [19, 25, 27] and execution environment optimization or on general LLM serving system optimization[9, 35]. Specifically, optimizing the execution environment warmup (Docker and Conda) only addresses one-time startup overhead, which accounts for an extremely small portion of the total execution time. Similarly, general LLM serving system optimizations fail to reduce tool execution time, the primary contributor to end-to-end latency as indicated by our experiments.

Opportunities. Faced with the aforementioned problems, we identify an opportunity to optimize tool execution. Specifically, we can adopt speculative execution to pre-execute tools, thereby reducing the proportion of tool execution time in the overall execution process. For example, in a deep research task, after acquiring relevant URLs, we can directly download these web pages in advance to reduce the time consumed by web page downloading.

Challenges. However, leveraging the aforementioned opportunity is non-trivial. Since LLM requests are unique, it is highly challenging to quickly and accurately predict the next tool to be used. More importantly, merely predicting the tool is insufficient; we also need to predict the parameters required by the tool, which is even more challenging. Furthermore, inaccurate pre-execution could disrupt the original agent execution workflow, which poses another challenge to integrate tool pre-execution in the current system design.

Key insights and contributions. To address these challenges, we propose **P**attern-**A**ware **S**peculative **T**ool **E**xecution (**PASTE**), a method that accelerates agent serving by leveraging speculative tool execution. The design is motivated by two key observations about agent workloads. First, tool invocations exhibit strong application-level control flow (recurring tool-call sequences). For example, a *git clone* operation is almost always followed by *git checkout*. Second, tool parameters follow predictable data flow patterns, where arguments are implicitly derived from the outputs of earlier tools. As one example, the URL required by a *download* tool is typically extracted directly from the JSON output produced by a preceding *search* operation.

Operationalizing these insights, however, requires solving two fundamental problems: formalizing unstructured tool-call sequences and managing the risks of probabilistic execution. PASTE addresses these through two components: a novel pattern abstraction that decouples control flow from data flow, and a risk-aware scheduler that preserves the execution performance of authoritative tools.

To resolve the formalization problem, PASTE introduces the **Pattern Tuple** (context, prediction, function, probability). This abstraction achieves robustness by strictly separating execution *structure* from execution *content*. While agents may solve similar problems using diverse natural language phrasing, the tuple defines context strictly over event signatures (sequences of tool types), allowing the system to identify stable control flows amidst volatile user inputs. Furthermore, the tuple utilizes a symbolic value mapping function to encode the logical relationships between data, enabling the system to automatically resolve implicit parameter passing between tools without invoking the LLM.

To manage execution risk at runtime, the PASTE scheduler strictly partitions the workload into authoritative invocations and speculative invocations. It employs *opportunistic scheduling*: speculative jobs run *only* on slack resources (i.e., transient idle compute/memory/IO capacity) and are throttled by a dynamic slack budget, so they can harvest otherwise-wasted cycles without competing with the normal LLM inference or critical-path tool execution. Critically, PASTE guarantees non-interference by instantaneously preempting speculative work the moment resource contention arises, ensuring that mispredictions never slow down authoritative progress.

Experimental methodology and artifact availability Our evaluation demonstrates that PASTE effectively improves the end-to-end latency of LLM agents. Compared to state-of-the-art baselines, PASTE reduces the average task completion time by 48.5% and accelerates average tool execution by 1.8×. PASTE will be open-sourced after review.

The main contributions of this paper are as follows:

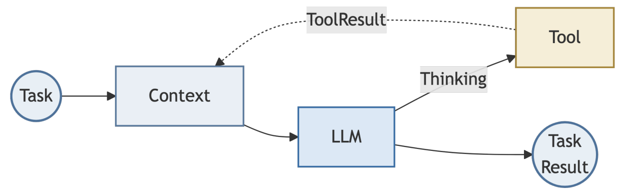


Figure 2: Workflow of LLM agent.

- **Characterization of Agent Latency.** We characterize modern LLM agent workloads, identifying the serial dependency between LLM inference and tool invocation as the dominant latency bottleneck.
- **Pattern-Driven Speculation.** We introduce a Pattern Analyzer that abstracts dynamic agent behaviors into stable application-level invocation sequences and implicit parameter derivation rules, enabling accurate prediction of future tool calls.
- **Resource-Aware Orchestration.** We design an Online Scheduler that dynamically leverages these patterns to launch speculative tool execution within available compute budgets, effectively parallelizing tool processing with LLM generation.

2 Background and Motivation

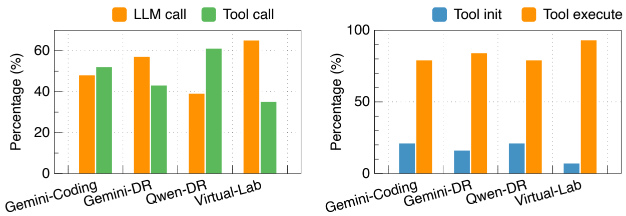
2.1 The LLM Agent Basics

2.1.1 The Era of LLM Agents. The evolution of Large Language Models (LLMs) has shifted the computing paradigm from static text generation to autonomous problem solving, including deep research[31, 33], bug-fix assistance[1, 3], and agent employee[4, 28]. While standard LLM inference treats the model as a function $f(text) \rightarrow text$, LLM Agents reframe the model as the cognitive core of a broader control loop. An agent is defined not just by its underlying model, but by its ability to orchestrate multi-step tasks through the use of external tools (e.g., Python interpreters, web retrieval APIs, vector databases) and long-term memory.

This shift has profound implications for system design. Unlike traditional inference workloads that are relatively short and independent, agent workloads are long-running, stateful, and highly sequential. A single user request (e.g., "Analyze this dataset and plot the trends") triggers many intermediate reasoning steps and tool executions that may span minutes. Consequently, the system must transition from optimizing individual request latency (Time-to-First-Token) to optimizing the end-to-end latency.

2.1.2 The Agent Execution Model. LLM agents naturally follow an *Iterative LLM–Tool Loop*, as shown in Figure 2. In each iteration, the LLM is conditioned on the current context and execution history, and then either (i) emits the final answer and terminates, or (ii) decides to invoke a tool with concrete parameters. If a tool is invoked, the system executes the call, appends the tool output back into the session state, and resumes LLM inference on the updated context.

This loop repeats until the LLM determines that enough evidence has been gathered and produces the final result as the session output. From a system perspective, end-to-end latency is jointly determined by LLM inference and tool execution.



(a) Agent time breakdown (LLM vs. tools)

(b) Tool time breakdown (init vs. execution)

Figure 3: Time Breakdown of Agent and Tools

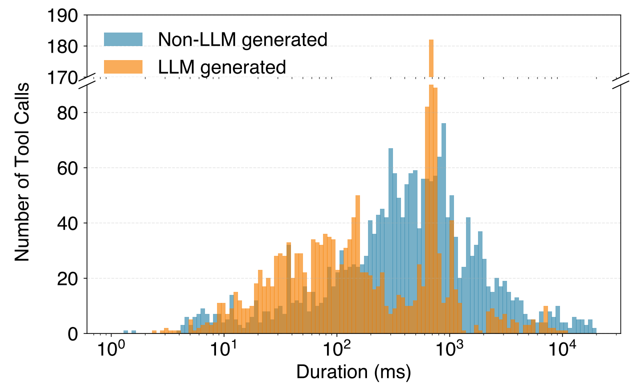


Figure 4: Histogram of tool execution time. Blue bars represent tool calls whose arguments are derived from the prompt or outputs of previous tools, while orange bars represent tool calls with LLM-generated arguments.

2.2 Problems and Challenges

2.2.1 Problems of Current Agents. The execution time of LLM agents can be divided into two parts, LLM generation and tool execution. We first measure the time spent in these two parts using three mainstream agent benchmarks (Deep Research Bench [11], SWE Bench [18], ScholarQA [8]). These benchmarks all cover the three dominant application domains of deep research, bug-fix assistance, and scientific research. Our experimental setup uses three main-stream agent products, with both LLM API (Gemini-2.5, GPT-5.2), and a local deployed Qwen-DeepResearch-30B [43] model on a server with 8 NVIDIA A100-80G GPUs.

Figure 3a shows the latency breakdown of representative requests from these benchmarks. The results show that tool execution constitutes a substantial portion of the total request lifetime. On average, tool execution accounts for 60% of the latency in coding tasks, 50% in deep research tasks, and 36% in scientific tasks. This observation remains consistent across all three benchmarks. As discussed earlier, the LLM generation and tool execution are strictly serialized because of inherent data dependencies. As a result, the large overhead of tool execution becomes a direct bottleneck, leading to suboptimal end-to-end latency for the overall system.

2.2.2 Inefficiencies of Current Approaches. Existing serverless workflow, and microservice optimizations typically require a *static*, end-to-end execution graph (often a DAG) to make scheduling decisions and optimize caching and data movement [24, 29, 30, 46, 53, 54]. When these systems perform prediction, it is largely conditioned on the complete static DAG (i.e., predicting the next branch or request given the graph). This assumption breaks for agentic workloads: the control flow is generated online, and the next tool call is not known until the LLM finishes the current step. Without the full DAG a priori, ahead-of-time scheduling and graph-based prefetching provide limited benefit.

Prior work has focused on cold start mitigation and environment reuse [19, 25, 41]. However, in our agent traces, initialization typically accounts for less than 20% of overall tool latency. In addition, speculation in agent workloads is constrained by tool side effects and a large, context dependent argument space. Prior works are ineffective in the agent serving scenario [39, 50].

2.2.3 Opportunities and Challenges. First, predicting the next tool to be used is highly challenging. Unlike traditional branch prediction in CPUs, which operates over a limited and well structured set of instructions, agent requests are highly diverse and unstructured. It is therefore difficult to design a simple method that can achieve highly accurate tool prediction. If the predictor fails to identify the tool invocation earlier than the LLM itself, speculation provides no performance benefit.

Second, predicting the next tool alone is not sufficient. The system must also accurately predict the exact input parameters required by that tool. This intent plus argument prediction is computationally difficult because the arguments, such as specific code snippets, file paths, or search queries, are often generated on the fly and depend heavily on the current context.

Third, integrating pre-execution is complicated by the state mutating nature of agent tools. Unlike read only operations, inaccurate pre-execution can corrupt the environment, such as by installing incompatible dependencies, and disrupt the original workflow. Restoring consistency then requires expensive rollback mechanisms and intricate system design.

2.3 Observations and Insights

To solve the above challenge, we conduct a comprehensive characterization study of agent requests from SWE-bench, MetaGPT and OpenHands benchmarks. Our analysis reveals that while agent behavior appears non-deterministic at a macro level, it exhibits strong temporal locality and data-flow dependencies at the micro-level. These characteristics provide the primitives necessary for speculative execution.

2.3.1 Insight 1: Predictable Control Flow Patterns. Contrary to the assumption that agents select tools randomly, we observe that tool invocations follow distinct, domain-specific state transition probabilities. We categorize these transitions into “Strong Chains” (deterministic sequences) and “Refinement Loops” (iterative patterns).

The “Edit-Verify” and “Locate-Examine” Patterns in Coding. In bug-fix assistance tasks, agents act as state machines driven by the codebase status.

(1) Edit-Verify Pattern: This is the strongest correlation observed. Among all coding traces, 55% of successful *file_editor* tool calls (write/replace) were immediately followed by a *terminal* tool call

(specifically pytest or python execution). This reflects a fundamental workflow: modification necessitates validation.

(2) Locate-Examine Pattern: During the debugging phase, we observe a causal link between output discovery and file access. Among all coding traces, 38% of *grep* tool calls were immediately followed by *file_editor* tool call.

The "Search-Visit" Pattern in Research. In DeepResearch tasks, the control flow follows a funnel structure—starting broad and narrowing down. 51% of tool calls *search* 10 related results, and then immediately trigger a tool call to *visit* the top 3-4 URLs. In addition, when a website was fetched via tool *web_fetch*, it would *pre_fetch* other URLs embedded within that website. This occurred in 20% of *web_fetch* calls.

Implication for agent system design. The high transition probability between these states suggests that the next tool prediction is feasible. We can utilize high-probability tool call patterns to accelerate the tool calling.

2.3.2 Insight 2: Implicit Data Flow and Parameter Derivation. Predicting the next tool is insufficient; speculative execution also requires predicting the tool parameters. A key finding of our study is that not all tool parameters are "hallucinated" by the LLM from scratch; they could be derived from the output of previous tools. Moreover, as shown in Figure 4, tool calls that use arguments derived from the prompt or previous tool outputs are generally more time-consuming.

(1) Producer-Consumer Pattern: In web tasks, the *visit* call requires a URL. In 95% of cases, this URL is a strict substring of the JSON output from the preceding *search* call. Agents typically select URLs containing query keywords, such as matching the term Prometheus in the page title for a Prometheus-related query. In coding tasks, the *file_editor* requires a filename, which could be distilled directly from the output of a prior *grep* call.

(2) Repetitive-Debug Pattern: In coding tasks, agents follow an iterative debugging loop that involves file edits and environment modifications. After completing a fix, the agent immediately executes the program to verify the result, which creates a tight edit verify dependency cycle.

Implication for agent system design: This observation enables Argument Prediction. Instead of generating arguments token-by-token (which is slow), PASTE can identify "candidate arguments" (URLs, filenames) from the execution history and speculatively execute on the most likely candidates.

2.3.3 Insight 3: Latent Parallelism in Serial Agent Execution. While standard agent frameworks (e.g., ReAct) force serial execution, our analysis reveals significant opportunities for parallelism.

(1) Broad Search Pattern: Once the LLM determines the keywords, we find that agents often issue multiple search queries that span Arxiv, PubMed, and Google Scholar. The search process could be parallelized.

(2) Batch Fetch Pattern: Upon receiving the search results, the agents need to fetch the landing pages, PDFs, and code repositories in a sequential way. Similarly, when a code execution error occurs, the agent sequentially opens all relevant code files. The above processes could also be parallelized.

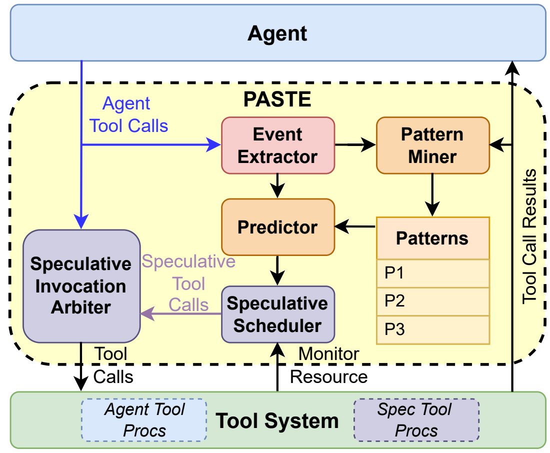


Figure 5: System architecture of PASTE.

Implication for agent system design: The logical workload is not strictly serial. By identifying **independent sub-tasks** (like fetching multiple URLs or reading multiple files) via speculation, PASTE can break the strict ReAct loop and saturate the otherwise idle network/IO resources.

3 Overview

To address the latency bottlenecks inherent in sequential agent execution, we present PASTE, a method designed to accelerate agent serving using speculative tool calling. PASTE operates on the insight that while agent behaviors are non-deterministic at the request-level, they exhibit stable application level control flows (recurring tool-call sequences) and predictable data dependencies (parameter passing between tools).

However, transforming these insights into a practical system requires solving two fundamental problems: **formalizing unstructured agent dependencies** and **managing the risks of probabilistic execution**. As shown in Figure 5, PASTE addresses these through two core components: a pattern abstraction that decouples control flow from data flow, and a scheduler that decouples speculative tool execution from authoritative tool paths.

3.1 Abstraction

The first problem lies in representation: How do we formalize the relationships between tools and the flow of parameters in a way that is embeddable into a generic agent system? Hard-coding dependencies (e.g., "always run *git clone* after *search*") is brittle because agent requests are diverse; the validity of a dependency often hinges on the specific context and data available.

To capture these dynamics, PASTE introduces the **Pattern Tuple** (context, tool prediction, function, probability). This abstraction first enables robustness by separating the execution structure from the execution content. While agents often solve similar problems using different natural language phrasing, the tuple defines the context strictly over event signatures (sequence of tool types) to identify stable control flows despite this noise.

Secondly, the pattern tuple supports late-binding value resolution. A major hurdle in speculation is parameter passing, since a

Table 1: An example pattern definition

	C	T	f	p
P1	[(Search, success)]	Web_fetch	Web_fetch: arg0 = SearchRes["list"][0]["url"]	0.9
P2	[(Search, success), (Web_fetch, fail)]	Web_fetch	Web_fetch: arg0 = SearchRes["list"][1]["url"]	0.8

tool cannot be executed without its arguments. Rather than predicting concrete values, which often leads to hallucination, PASTE predicts a value mapping function (the third element in the tuple). This function encodes the logical relationships between data.

3.2 Scheduling

The second problem is operational: How do we schedule predictions that are inherently uncertain? While application-level dependencies exist (e.g., *git clone* usually follows *repo search*), they are probabilistic ($p < 1$). Blindly executing every predicted tool would lead to severe resource contention and waste. To solve this, PASTE employs scheduling with opportunistic speculation. This design treats speculation not as a mandate, but as a mechanism for harvesting slack resources. The scheduler is designed by two principles: strict prioritization with isolation and the promotion mechanism.

The scheduler partitions the workload into authoritative invocations (generated by the agent, correctness-critical) and speculative Invocations (generated by PASTE, best-effort). Authoritative jobs are given strict priority and preemption capability. Speculative jobs are confined to a dynamic slack budget, which allows them to hide latency when resources are available. PASTE also ensures they are immediately suppressed when contention arises.

To maximize efficiency, PASTE implements a promotion protocol. When an authoritative request arrives and matches a running speculative job, that job is immediately promoted. It moves from low priority to high priority, and its result is committed directly to the agent history. This mechanism allows PASTE to safely convert uncertain bets into guaranteed latency reductions without redundant computation.

4 Pattern Abstraction

To operationalize the formalization of probabilistic dependencies, PASTE centers its design on the **Pattern Tuple**. This abstraction is defined to isolate structural regularity(control flow) from parameter dependencies (data flow). By doing so, the system can generalize across diverse agent sessions while preserving precise execution semantics.

4.1 The Pattern Tuple

We define a speculative pattern $\mathcal{P}$ as a tuple (C, T, f, p). Table 1 presents two concrete pattern definitions, P1 and P2, for a deep research agent. We explain the four components in the following paragraphs using these two examples.

(1) Context C **(Control Flow Anchor).** C defines the structural precondition for a prediction and is modeled as an order-preserving subsequence of event signatures. Each signature contains only event metadata, such as the tool type and execution status, and intentionally excludes high variance payloads like specific query strings.

This payload agnostic design ensures robustness through signature matching. For example, in Pattern P1 shown in Table 1, the context $[(Search, success)]$ matches any successful search operation, independent of whether the query concerns “distributed systems” or “quantum physics”. By removing natural language variability, PASTE is able to identify stable control flows that are shared across different users and tasks.

(2) Target T **(Prediction).** T specifies the type of the future tool invocation and represents the agent intent before the LLM explicitly generates it. In P1, the target Web_fetch indicates that after a successful search, the agent intends to retrieve a webpage.

(3) Value Mapping Function f **(Data Flow Logic).** f encodes the dependency logic required for late-binding value resolution. Instead of predicting concrete argument values (which risks hallucination), f is a symbolic function that specifies how to derive arguments from the payloads of historical events in C. This design captures the producer consumer relationships that commonly arise in agent workloads.

In P1, f is defined as $arg0 = SearchRes[”list”][0][”url”]$, which means that the URL for the fetch tool is derived dynamically from the first result of the preceding search. Pattern P2 handles a failure case in a similar way. When the first fetch fails, as indicated by $(Web\_fetch, fail)$ in C, f adapts by selecting the second URL ($index[1]$). This function is evaluated lazily at runtime, which allows PASTE to generate precise parameters as soon as the required upstream data becomes available.

(4) Probability p **(Confidence Metric).** p represents the empirical success rate of a pattern and is derived from validation over historical traces. This score turns prediction from a binary decision into a graded signal. For example, P1 has a confidence of 0.9, while the fallback pattern P2 has a confidence of 0.8. This distinction allows the risk aware scheduler to favor high confidence speculations as p approaches one, while deprioritizing weaker ones. As a result, the system allocates resources based on expected utility rather than treating all predictions equally.

4.2 Pattern Mining and Validation

Constructing the pattern pool requires distinguishing between structural regularities (which are frequent but coarse) and valid parameter flows (which are precise but hard to verify). To achieve this, PASTE employs a Two-Phase Mining Strategy (summarized in algorithm 1) that explicitly mirrors the abstraction’s decoupling of control flow and data flow.

4.2.1 Phase I: Structural Mining (The "Where"). The process starts by treating execution traces as streams of event signatures. As shown in algorithm 1 (line 2), PASTE removes high cardinality payload data, which reduces the search space to a small set of tool types. For each unique tool invocation type t, the system aggregates the preceding signature sequences (line 5) and applies sequential pattern mining (e.g., PrefixSpan) to identify recurring short horizon subsequences (line 8). This phase filters out low frequency noise (lines 6-7) and establishes candidate contexts C and targets T based solely on control flow. Because it ignores payloads, this approach remains lightweight and scalable even when applied to massive execution logs.

Algorithm 1: Pattern Mining and Validation in PASTE

Input: Execution traces $\mathcal{E}$, max context length k, min support σ, confidence threshold τ
Output: Pattern set $\mathcal{P}$
1 $\mathcal{P} \leftarrow \emptyset$
// Step 0: Extract event signatures (temporal order preserved)
2 $\mathcal{S} \leftarrow$ ExtractEventSignatures($\mathcal{E}$)
3 $\mathcal{T} \leftarrow$ UniqueToolInvocationEventSignatures($\mathcal{S}$)
4 **foreach** $t \in \mathcal{T}$ **do**
  // Step 1: Mine frequent structural contexts preceding tool t
  // extracts all sequences of length $\leq k$ immediately preceding t
5   $\mathcal{D}_t \leftarrow$ CollectPrecedingSignatures($\mathcal{S}, t, k$)
6   **if** $|\mathcal{D}_t| < \sigma$ **then**
7   **continue**   // Skip tools with insufficient support
8   $Q \leftarrow$ MineFrequentSubsequences($\mathcal{D}_t$)
9   **foreach** $c \in Q$ **do**
    // Step 2: Infer value mapping for $(c \rightarrow t)$
10     $\mathcal{M} \leftarrow$ MatchOccurrences($\mathcal{E}, c \rightarrow t$)
11     $f \leftarrow$ InferValueMapping($\mathcal{M}$)
    // Step 3: Validate and score the pattern
12     $C \leftarrow$ MatchOccurrences($\mathcal{E}, c$)
13   **if** $f \neq \emptyset$ **then**
      // retaining only those occurrences where f holds
14       $\mathcal{M}^\star \leftarrow$ FilterByValueMapping($\mathcal{M}, f$)
15   **else**
16       $\mathcal{M}^\star \leftarrow \mathcal{M}$
17     $p \leftarrow |\mathcal{M}^\star|/|C|$
18   **if** $p \geq \tau$ **then**
19   Add pattern (c, t, f, p) to $\mathcal{P}$
20 **return** $\mathcal{P}$

Algorithm 2: Runtime Pattern Prediction

Input: Recent execution events $E = \langle e_1, \ldots, e_k \rangle$;
Pattern pool $\mathcal{P} = \{(C_i, T_i, f_i, p_i)\}$;
Historical statistics table H
Output: Predicted tool invocations $\mathcal{T}$ with probability
1 $\mathcal{T} \leftarrow \emptyset$
2 **foreach** *pattern* $(c, t, f, p) \in \mathcal{P}$ **do**
3   $E_C \leftarrow$ MatchOccurrences(E, c)
4   **if** $E_C \neq \emptyset$ **then**
5     $a \leftarrow f(E_C)$
6     $e_{pred} \leftarrow$ CreateEvent(t, a)   // with signature and args
7   Add (e_{pred}, p) to $\mathcal{T}$
8 **return** $\mathcal{T}$

4.2.2 *Phase II: Symbolic Dependency Inference (The "How").* In the second phase, PASTE infers a value mapping function f (line 11) to realize *late-binding value resolution*: it explains how the arguments of the predicted tool T can be derived directly from the payloads in C. To keep inference simple, PASTE only considers a few recurring transformations that we repeatedly observe in agent traces: (i) field/-path lookup in a structured result (e.g., $SearchRes["list"][i]["url"]$), (ii) choosing an element by index with a fallback (e.g., use the first result; if it fails, try the next), and (iii) basic string formatting/normalization when passing values across tools. If such an f is found, the pattern becomes fully parameterized; otherwise, it predicts only the tool type.

4.2.3 *Validation.* Finally, the algorithm validates each candidate tuple (C, T, f) against the trace dataset to estimate its reliability. PASTE computes the empirical success probability p (line 17), defined as the fraction of context matches in which the predicted invocation actually occurred and satisfied the value mapping f. Patterns with confidence below the threshold τ are discarded (lines 18-19), which ensures that the runtime is not burdened with low-probability speculation.

4.3 Online Pattern Prediction

At runtime, PASTE uses the validated pattern pool to perform continuous, non-blocking prediction of near-future tool invocations, as detailed in algorithm 2. The process operates incrementally on the live event stream and updates predictions without interrupting the agent's critical execution path.

4.3.1 *Context Matching (Control Flow).* When each new event arrives, PASTE updates a bounded window of recent execution events E (line 1). The system then iterates over the pattern pool and identifies all patterns whose context C matches a suffix of the observed event signatures (line 3). This matching relies only on metadata and preserves temporal order, which ensures that predictions are triggered only under execution conditions that are consistent with historical observations. Because multiple patterns may match the current state, PASTE generates candidate predictions for all matches without performing early arbitration.

4.3.2 *Function Evaluation (Data Flow).* For each matched pattern (C, T, f, p), PASTE attempts to evaluate the associated value mapping function f against the concrete payloads of the matched events E_C (line 5). This step connects the abstract structure with executable actions. Depending on whether the required inputs are available in E_C, the evaluation produces a predictive event e_{pred} (line 6). This event may be a fully specified tool invocation, a partially parameterized invocation, or only the tool identity.

4.3.3 *Probabilistic Hint Generation.* Finally, the resulting predictive invocation is annotated with the empirical probability p learned during offline validation (line 7). These annotated predictions are collected into the set $\mathcal{T}$ and passed to the scheduler as probabilistic look ahead hints. This design allows the scheduler to weigh the expected utility of each speculation against its resource cost, using the confidence score p as the primary factor.

5 Scheduling with Opportunistic Speculation

The performance of PASTE is largely determined by how effectively it schedules tool executions under uncertainty. Unlike conventional cluster schedulers, PASTE must continuously arbitrate between two classes of tool invocations: *authoritative invocations*, which are issued by the agent's actual execution and are correctness-critical,

Algorithm 3: Scheduling with Opportunistic Speculation

Input: JobTable $\mathcal{J}$, AvailableResources R, SpecResourcesBudget B

Output: Scheduling decisions

while PendingJobsExist($\mathcal{J}$) **do**
  $J_r \leftarrow$ FilterPendingAuthoritativeJobs($\mathcal{J}$)
  // Step 1: Confirm speculative jobs if possible
  foreach $j \in \mathcal{J}$ **do**
    $s \leftarrow$ FindMatchingSpecJob(j)
    if $s \neq \varnothing$ **then**
      ConfirmSpecJob(s, j)
      $J_r \leftarrow J_r \setminus \{j\}$
  // Step 2: Ensure resources for real jobs
  while RequiredResources(J_r) > R **do**
    $j \leftarrow$ PreemptSpecJob(J)
    if $j = \varnothing$ **then**
      break
    AbortJob(j)
    ReleaseResources(j, R)
  // Step 3: Primary scheduling of real jobs
  while $J_r \neq \emptyset$ **do**
    $j \leftarrow$ PrimaryScheduling(J_r, R)
    AllocateResources(j, R)
    LaunchJob(j)
    $J_r \leftarrow J_r \setminus \{j\}$
  // Step 4: Opportunistic scheduling of speculative jobs
  $J_s \leftarrow$ FilterPendingSpecJobs($\mathcal{J}$)
  while $J_s \neq \emptyset$ **and** $R > 0$ **and** $B > 0$ **do**
    $j \leftarrow$ OpportunisticSpecScheduling(J_s, R, B)
    AllocateResources(j, R)
    LaunchJob(j)
    $B \leftarrow B -$ ResourceCost(j)
    $J_s \leftarrow J_s \setminus \{j\}$

and *speculative invocations*, which are generated by the pattern predictor with associated uncertainty and may never be consumed.

The scheduler's objective is to maximize the expected reduction in future tool latency by exploiting idle resources, while strictly guaranteeing that authoritative invocations are never delayed, reordered, or otherwise affected by speculation. Algorithm 3 illustrates how PASTE achieves this objective through opportunistic speculation.

At each scheduling decision point, authoritative invocations are always prioritized via the existing scheduling policy, denoted as `PrimaryScheduling`. Speculative invocations are treated as best-effort tasks: they may only execute on resources not required by authoritative work and are immediately preempted when contention arises. This design ensures non-interference with the agent's normal execution while enabling aggressive latency hiding.

The scheduling process proceeds in four stages. First, when an authoritative invocation arrives, the scheduler first checks whether a matching speculative job already exists. If the speculative execution has completed, PASTE directly reuses its result, eliminating redundant execution. If the speculative execution is still in progress, PASTE *promotes* the running speculative job to authoritative: the execution continues without interruption, its result is committed upon completion, and the job becomes non-preemptible. In both cases, the authoritative invocation is satisfied without launching a new execution. This promotion mechanism allows PASTE to safely harvest ongoing speculative work while preserving correctness and minimizing response latency.

Second, before scheduling any new work, the scheduler ensures that sufficient resources are available for all pending authoritative jobs by preempting speculative jobs if necessary. Third, authoritative jobs are scheduled according to the existing primary policy without modification. Finally, if slack resources remain within a predefined speculative resource budget B, the scheduler opportunistically dispatches speculative jobs.

Overall, this design guarantees that existing scheduling objectives for authoritative invocations are preserved, while speculative invocations are confined to slack resources and remain opportunistic, best-effort, and preemptible by construction.

5.1 Opportunistic Speculative Scheduling

The goal of speculative scheduling in PASTE is to utilize transient slack resources to proactively execute predicted tool invocations, thereby reducing the latency of future authoritative executions, while strictly preserving isolation and performance guarantees for real agent-driven invocations.

Before enqueuing speculative actions, the scheduler preprocesses predicted tool invocations according to the speculation eligibility policy, merging duplicate predictions and discarding those that violate safety or side-effect constraints. Only policy-compliant speculative actions are admitted for opportunistic execution and prioritized based on their expected performance benefit.

At each scheduling decision point, the scheduler observes the available slack resources R_{slack}, defined as the residual capacity after all authoritative jobs have been admitted, together with a speculation budget B that caps the total resources allocable to speculative execution. Let $\mathcal{J}_s$ denote the set of pending speculative jobs generated by the pattern predictor. Each speculative job $j \in \mathcal{J}_s$ is characterized by a predicted consumption probability $p_j \in (0, 1]$, an estimated latency reduction T_j if the speculative result is eventually consumed, an estimated resource cost c_j, and an expected execution duration d_j.

The speculative scheduling decision is to select a subset of jobs for execution. We introduce a binary decision variable $x_j \in \{0, 1\}$ for each $j \in \mathcal{J}_s$, where $x_j = 1$ indicates that job j is admitted for speculative execution. The objective can be formulated as the following constrained maximization problem:

$$\begin{aligned} \max \quad & \sum_{j \in \mathcal{J}_s} x_j \cdot p_j \cdot T_j \\ \text{s.t.} \quad & \sum_{j \in \mathcal{J}_s} x_j \cdot c_j \leq \min(R_{\text{slack}}, B). \end{aligned} \tag{1}$$

Solving Eq. 1 optimally in an online setting is intractable, as speculative jobs arrive dynamically, their benefits are probabilistic, and available slack resources fluctuate over time. Accordingly, PASTE adopts a greedy, best-effort heuristic that prioritizes speculative jobs based on their expected utility per unit resource. Specifically, for each speculative job j_i, the scheduler computes a priority score:

$$U(j_i) = \frac{p_i \cdot T_i}{c_i \cdot d_i}, \tag{2}$$

which jointly captures the likelihood of consumption, the potential latency benefit, and the resource and time cost of speculative execution.

During opportunistic scheduling (Algorithm 3, Step 4), speculative jobs are ranked in descending order of $U(j_i)$ and launched greedily as long as sufficient slack resources remain. All speculative jobs are explicitly marked as preemptible and may be aborted at any time to reclaim resources for authoritative executions. When resource contention arises, the scheduler preempts running speculative jobs with the lowest utility scores first, ensuring that speculative execution never delays or degrades the performance of real agent-driven tool invocations.

5.2 Speculation Eligibility Policy

To ensure correctness, safety, and efficient resource utilization, PASTE governs the use of predictive tool invocations through a *speculation eligibility policy* specified by users or system operators. This policy resolves conflicts among multiple predictions, constrains speculative actions based on prediction confidence and side-effect risk, and provides explicit control to system operators.

Speculative Invocation Arbitration. Because patterns are mined from data and are not guaranteed to be unique, multiple patterns may predict the same future tool invocation, potentially with different probabilities or levels of parameter completeness. PASTE therefore performs arbitration at the level of speculative *invocations*, rather than patterns. For predicted invocations targeting the same tool, PASTE retains at most one speculative action according to an *expected speculative utility* criterion. By default, utility is defined as the product of the prediction probability and the estimated latency reduction of speculative execution, though the framework allows alternative utility metrics. All other speculative instantiations of the same tool are discarded to avoid redundant work.

Policy-Constrained Graded Speculation. PASTE supports *graded speculation* along two orthogonal dimensions: the completeness of prediction and the risk of side effects. When a tool invocation is fully parameterized and the operation is declared side-effect-free by policy, PASTE may speculatively execute the invocation end-to-end. When predictions are partial, such as when only the tool identity or coarse parameters are known, PASTE limits speculation to shallow preparatory actions, including runtime warm-up, container or environment initialization, and dependency loading.

For operations that may incur side effects, speculative execution is further constrained by policy. PASTE supports transformed speculation, in which potentially unsafe operations are rewritten into safe variants, such as dry-run execution or execution against staging resources that prefetch data without committing irreversible state changes. Importantly, PASTE never speculates beyond the bounds specified by the policy and does not infer side-effect freedom automatically.

```yaml
speculation_policy:
  default:
    allow: false

  tools:
    web_search:
      allow: true
      max_speculation: full

    pip_install:
      allow: true
      max_speculation: dry_run

  deduplication:
    strategy: max_expected_speculative_utility
```

Figure 6: Example user-defined speculation eligibility policy. The policy constrains which tools may be speculated, the allowed speculation level, and how duplicate predictions are merged.

Figure 6 illustrates an example speculation eligibility policy. The policy is explicitly defined by users or system operators and specifies, for each tool, whether speculation is permitted, the maximum allowed speculation level, and the handling of potential side effects. When predictions are duplicated across multiple patterns, PASTE applies a configurable deduplication strategy, retaining only the speculative action with the highest expected latency reduction. This policy-driven design ensures that speculative execution is both safe and predictable, while allowing operators to trade off aggressiveness and resource usage according to deployment requirements.

Together, these mechanisms ensure that incorrect predictions at worst consume bounded resources, while preserving the correctness of agent execution and enabling operators to control the aggressiveness of speculation.

6 Implementation

We implement PASTE as a middleware layer that interposes between LLM agents and tool execution backends. The prototype consists of approximately 8,000 lines of TypeScript (Gemini-CLI integration) and 4,000 lines of Python (Qwen-DeepResearch and Virtual-Lab integrations). PASTE runs as a tool-serving proxy: agents send tool requests through PASTE, which records execution traces and coordinates speculative executions while preserving the original tool API.

Event Extractor. The event extractor ingests tool invocation logs and converts them into a normalized event stream. It delineates sessions using a configurable inactivity threshold. For LLM interactions, we wrap the generation API to collect request/response timing and streaming-chunk metadata, enabling consistent alignment between model-side activity and tool executions.

Pattern Mining and Prediction State. We implement the mining and prediction pipeline as in-memory summaries keyed by compact encodings of recent tool context. Each tool call is mapped to a discrete signature using lightweight classifiers for common argument forms (e.g., URLs, file paths, and free-form queries). These summaries are updated online and checkpointed periodically to disk for durability.

Pattern Predictor. The predictor serves next-step candidates via hash-map lookups over serialized context keys (constant-time per key in our implementation). Parameter suggestions reuse previously observed values when available, with simple extraction rules for common value types.
Speculative Scheduler. The scheduler maintains two asynchronous task queues: an active queue for agent-issued invocations and a shadow queue for speculative executions. Both share a unified result cache keyed by tool name and argument hash; when an authoritative request matches an in-flight speculative execution, PASTE joins the existing task and returns its result.
Agent Integrations. We integrate with agents by wrapping their tool-dispatch interfaces and routing calls through the PASTE HTTP proxy.

7 Evaluation

This section evaluates PASTE along five axes: (1) End-to-End (E2E) latency and throughput improvement, (2) where the improvement comes from (time breakdown and overlap), (3) scalability under concurrent sessions and bursty arrivals, (4) prediction quality, and (5) speculation overhead and safety under mispredictions. Across all experiments, we hold hardware, LLM configuration, tool interfaces, and resource budgets constant to isolate system effects.

7.1 Evaluation Methodology

Experimental setup and controls. Unless stated otherwise, all systems run on the same hardware/software stack (Table 2). We apply the same per-task timeouts and retry policy to all systems. For pattern prediction, we mine tool-call patterns from a corpus of historical tasks and evaluate on a disjoint set of new tasks (no train/test overlap).

Workloads and baselines. We evaluate three agents: (i) **Virtual-Lab** [42], a science-focused agent designed to support end-to-end research workflows; (ii) **Qwen Deep Research** [7], an agent tailored for *deep research*—multi-step, tool-assisted information gathering, synthesis, and report writing over open-ended questions; and (iii) **gemini-cli** [2], Google's official open-source, production-grade agent for code modification as well as general-purpose tasks. We use three benchmark families: **DeepResearchBench** [11] for deep-research-style tasks, **SWE-bench** [18] for software engineering and code-writing tasks, and **ScholarQA** [8] as a domain-specific scientific research benchmark.

We compare PASTE against **ORION** [27] (serverless DAG execution) and **SpecFaaS** [39] (speculative serverless execution) under the same tool interfaces and resource limits.

Real-world trace-driven request arrivals. We replay a production Azure Functions invocation trace [38] to generate realistic, bursty arrivals. Each trace record becomes one agent request issued at its logged timestamp, preserving the *arrival process.*

LLM configuration. As Table 2 shows, we choose both state-of-the-art proprietary LLM APIs and locally hosted open-source models, and keep the model configuration identical across all experiments.

Table 2: Evaluation setup

Component	Setting
Cluster	4 nodes
Per-node CPU	96 vCPUs, AMD EPYC 7V13
Per-node Memory	512 GB
Per-node GPU	8 GPU, NVIDIA A100 80GB
Total GPUs	32× NVIDIA A100 80GB
LLM API	Gemini-2.5-Pro, GPT-5.2
Local Model	Qwen-DeepResearch-30B [43]

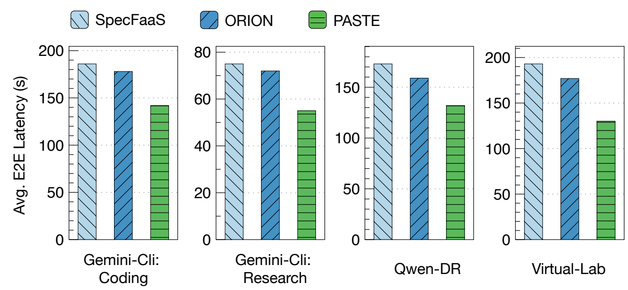


Figure 7: Average E2E latency comparison.

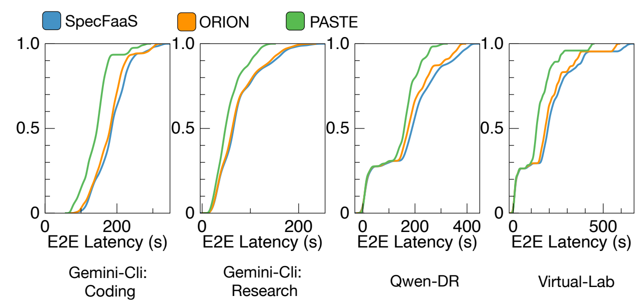


Figure 8: CDF of E2E latency for each task.

Metrics. Our primary metric is **E2E latency** (request arrival to final agent response) and tool execution latency. We also report tail latency (p95/p99), **tool stall time** (time the agent waits for tool execution results), throughput, speculative **hit rate**, and resource overhead.

7.2 E2E Latency Reduction

We first quantify whether PASTE reduces E2E latency for each agent. Figure 7 summarizes the results. Across all agents, PASTE consistently reduces E2E latency relative to both ORION and SpecFaaS. PASTE reduces average latency by up to 48.5%, and reduces p95/p99 tail latency by up to 48.6%/61.9%. Aggregating across all configurations, PASTE achieves an average speedup of **1.25×/1.32×** over ORION/SpecFaaS.

To further characterize the full latency distribution, Figure 8 plots the CDF of E2E latency aggregated across all configurations. PASTE stochastically dominates both baselines: for a given latency threshold, a larger fraction of requests complete within the threshold under PASTE. This indicates that speculation improves not only the tail but also the "bulk" of requests by reducing tool stalls on the critical path. We further quantify this mechanism (tool-stall reduction and overlap) in Sec 7.3.

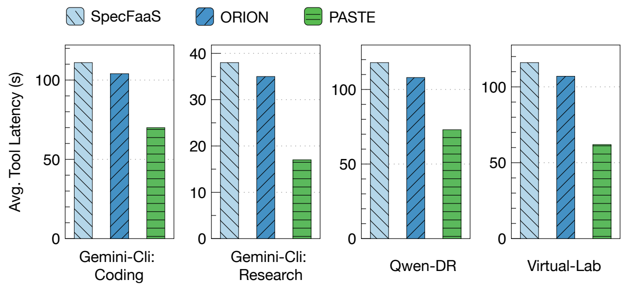


Figure 9: Average tool latency comparison.

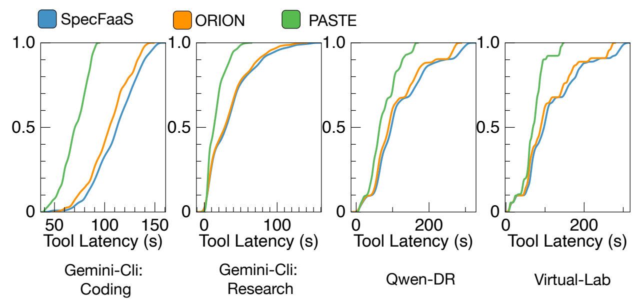


Figure 10: CDF of tool latency for each task.

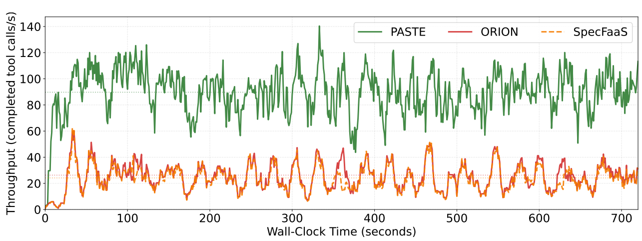
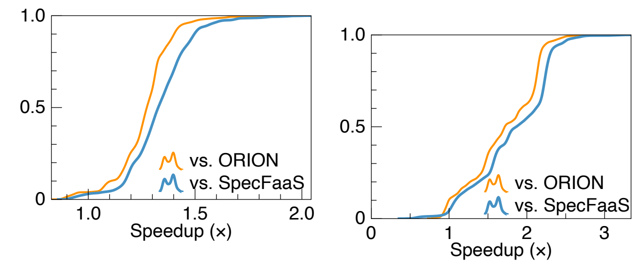


Figure 11: Throughput evaluation.

In addition to end-to-end improvements, we evaluate PASTE's ability to accelerate tool execution. Figure 9 reports average tool latency aggregated across all agents, and Figure 10 plots the corresponding per-task CDF. Across agents and benchmarks, PASTE significantly reduces tool latency relative to both ORION and SpecFaaS. PASTE reduces average tool latency by up to 55.2%, and reduces p95/p99 tool latency by up to 59.3%/60.6%. Aggregating across all configurations, PASTE achieves an average tool speedup of 1.71×/1.83× over ORION/SpecFaaS. These gains are consistent with speculative execution overlapping tool work with LLM generation and thus shortening the effective tool critical path.

Finally, we aggregate all tasks and examine PASTE's overall speedup relative to the baselines. Figure 12 reports the CDF of per-request speedup over ORION and SpecFaaS after pooling all requests. Most requests observe positive speedup (97% above 1×), while the worst cases remain close to parity, consistent with PASTE's low speculation overhead when predictions do not hit.

These gains are robust across agents, benchmarks, and LLM settings. Improvements are largest for tool-heavy tasks where tool stalls dominate the critical path, but remain non-negative for more LLM-dominated tasks, indicating that PASTE introduces low overhead when speculation does not hit. Overall, speculative tool execution reliably converts a portion of tool-wait time into overlap with LLM generation, improving both median and tail latency.



(a) CDF of per-request E2E speedup over ORION/SpecFaaS. **(b) CDF of per-request Tool speedup over ORION/SpecFaaS.**

Figure 12: CDF of Speedup for E2E and Tool scenarios.

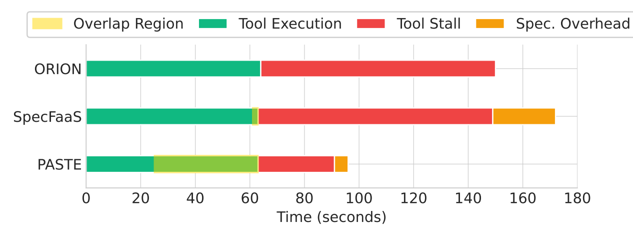


Figure 13: Time breakdown and overlap analysis, showing where PASTE reduces tool stall time by overlapping speculative tool work with LLM generation.

7.3 Time Breakdown and Overlap Analysis

To explain *why* PASTE reduces E2E latency, we instrument the runtime and attribute time to three components: (1) tool execution (active runtime), (2) tool stall (time blocked on waiting for next tool call due to LLM thinking), and (3) speculation overhead. We additionally measure **overlap**, defined as tool execution while the LLM is still thinking. Overlap reflects the effectiveness of speculative tool execution.

Figure 13 reports the breakdown for ORION, SpecFaaS, and PASTE. Compared with both baselines, PASTE reduces tool-wait time by 67%. Compared with SpecFaaS, PASTE increases the measured overlap by over 10×.

These results confirm that PASTE's speedups come from overlap: tool work that previously executed strictly after an LLM step is shifted earlier and completed in parallel with LLM generation. In contrast, ORION exposes tool latency directly after LLM generation, and SpecFaaS overlaps less effectively because its speculative policy is designed for static DAG applications but not agentic workflow. As a result, it cannot predict the argument of each tool call. Overall, PASTE reduces latency by hiding tool stalls rather than shifting cost elsewhere.

7.4 System Scalability

We next evaluate scalability to determine whether PASTE maintains low latency under many concurrent agent sessions, and to verify that speculation does not harm isolation by increasing queueing for

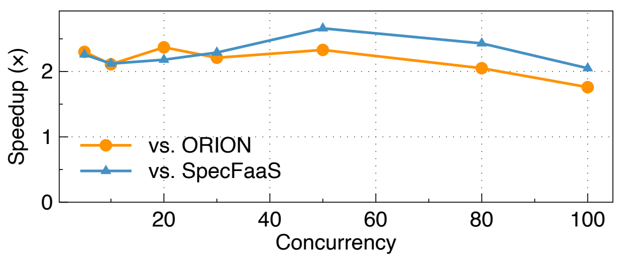


Figure 14: Scalability under multi-session concurrent agent requests: speedup compared with baselines.

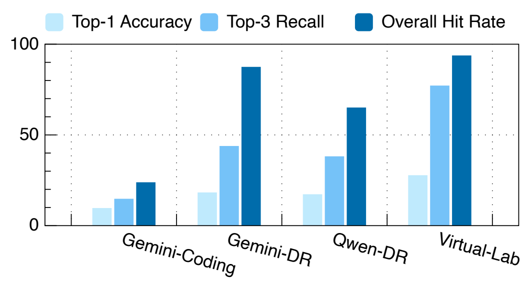


Figure 15: Pattern prediction quality: Top-1 accuracy, Top-3 recall, and overall hit rate.

authoritative tool calls. We stress the system by sweeping arrival rate and concurrent sessions while holding resource budgets fixed.

Figure 14 summarizes PASTE's E2E speedup performance as workload increases. At each concurrency, PASTE sustains at least 1.76×/2.05× higher speedup compared with ORION/SpecFaaS. These results demonstrate that PASTE scales without violating isolation: speculative work is throttled by explicit budgets and remains preemptible, so it does not crowd out authoritative tool execution.

7.5 Pattern Prediction Accuracy

Speculative execution is only effective when the system can correctly anticipate near-future tool calls. We therefore measure prediction quality using **Top-1 accuracy** (the highest-probability predicted tool matches the next tool actually invoked) and **Top-3 recall** (the next tool appears among the three most likely predictions). Top-3 recall is particularly relevant because PASTE may speculatively execute multiple candidates within a bounded budget. We also report **overall hit rate**, defined per tool call as the probability that *any* speculatively executed prediction matches the next tool actually invoked. Because PASTE can issue multiple predictions (and increase the number of candidates when the system is lightly loaded and spare resources are available), overall hit rate can be high even when Top-1 accuracy is low.

Figure 15 reports prediction quality by benchmark family. Overall, the predictor achieves up to 27.8% Top-1 accuracy, 43.9% Top-3 recall, and 93.8% overall hit rate. Accuracy is higher for structured tool sequences (e.g., compilation/test loops) and lower for open-ended exploration patterns typical of broad research tasks.

Despite imperfect Top-1 accuracy, strong Top-3 recall is sufficient in practice: PASTE can speculate on a small set of likely tools and still obtain overlap when any candidate hits. When prediction is uncertain, the explicit speculation budget bounds wasted work and prevents negative interference with authoritative execution.

7.6 Side-effect Evaluation

Finally, we evaluate safety to ensure that enabling speculation does not change externally visible behavior or violate isolation, even under mispredictions. We audit speculative executions to measure how many speculative actions would have caused external side effects, (ii) whether such effects were prevented from committing, and (iii) any divergences in final outputs relative to authoritative-only execution. Across all workloads, PASTE detects 602 potentially side-effecting speculative actions among over 20,000 speculative actions and prevents them from committing. No task produces a different final result compared with the baselines. Overall, the results support PASTE's safety claim: speculative actions are contained by policy and sandboxing and only become externally visible after authoritative confirmation.

7.7 Resource Overhead

For the latency–overhead tradeoff, for every 1 second of latency reduction, the PASTE consumes 0.02 core-seconds of CPU, 2.6 MB of memory, and 0.9 MB of network bandwidth. At moderate settings, PASTE achieves 48% tool execution latency reduction with extra 1-3 idle CPU cores and 250 MB additional memory. These results show that PASTE can be tuned to a practical "sweet spot" in which most speculative work is converted into useful overlap, while the remaining waste is bounded by explicit budgets. Overall, PASTE is lightweight enough to be deployed as a sidecar alongside existing agent runtimes, delivering latency wins without requiring dedicated infrastructure or disruptive changes to the execution stack. For the pattern predict and scheduling policy, the overall latency overhead is less than 100 ms.

8 Related Works

Agentic LLM serving acceleration. Recent systems treat agentic applications as *structured programs/workflows* and optimize serving via semantics/workflow-aware execution and orchestration (including agent communication/runtime co-design) [14, 15, 23, 26, 52, 55, 56]. Other work accelerates *augmented/tool-interrupt inference* with intercept and scheduling support [5, 44, 48, 51], while general LLM serving advances improve throughput/latency via better scheduling, multiplexing, and distributed serving (often applicable to agent workloads) [6, 12, 22, 36, 37, 45]. Because agent loops intensify context reuse and long-context pressure, many efforts focus on *KV/prefix/context caching and memory management*, including KV reuse/compression and context caching schemes [10, 16, 20, 21, 34, 35, 47, 49]. However, these systems primarily optimize LLM inference/serving and largely treat tool execution as an external interruption, leaving end-to-end tool latency unaddressed compared to PASTE.

Tool & runtime acceleration. Tool execution performance is shaped by serverless/workflow runtimes, so recent work reduces *workflow orchestration overhead* and mitigates *cold starts* via better provisioning/worker reuse [19, 24, 25, 40]. Additional systems target *cross-service caching/state management* for microservice graphs and serverless settings [53, 54], and speed up tool-heavy pipelines with more efficient *data passing/transfer paths* [29, 30, 46]. Yet, most of these techniques assume a largely static workflow/microservice graph and cannot directly exploit the online, LLM-generated control flow and context-dependent argument binding in agent loops, which PASTE is designed to handle. Yet, most of these techniques assume a static workflow/microservice graph and cannot directly exploit the online, LLM-generated control flow and context-dependent argument binding in agent loops, which PASTE is designed to handle.

Speculation. Speculation has been applied to *serverless functions* to accelerate workflows by executing likely-needed tasks early [39]. Emerging agentic work explores *speculative actions and speculative tool calls* to overlap tool latency with reasoning/search while preserving correctness constraints [17, 32, 50]. Compared to PASTE, existing speculation mechanisms struggle to (i) infer the concrete, context-dependent arguments that are only produced online by the agent, making accurate tool-call speculation difficult, and (ii) manage the correctness and side-effect risks of executing wrong calls, which requires explicit, risk-aware control.

9 Conclusion

In this paper, we address the fundamental serialization bottleneck in modern LLM powered agents, where a strictly sequential "LLM-tool" loop forces expensive resources to remain idle during external tool execution. We introduce PASTE, a speculative tool execution method that transforms this reactive workflow into a proactive and pipelined architecture. Using a new Pattern Tuple abstraction, PASTE decouples stable application-level control flows from dynamic data flow dependencies, which enables robust and late binding tool prediction. Combined with a risk-aware and opportunistic scheduler, the system exploits transient slack resources to hide tool latency while preserving strict performance for authoritative tasks.

References

[1] 2025. Agent Skills. https://agentskills.io/home. Accessed: 2026-01-25.
[2] 2025. Build, Debug & Deploy with AI. https://geminicli.com/.
[3] 2025. Claude Code | Claude. https://www.claude.com/product/claude-code.
[4] 2025. GitHub Copilot. https://github.com/features/copilot.
[5] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 81–95.
[6] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association.
[7] Alibaba-NLP. 2025. *DeepResearch: Tongyi Deep Research, the Leading Open-source Deep Research Agent.* https://github.com/Alibaba-NLP/DeepResearch
[8] Akari Asai, Jacqueline He, Rulin Shao, Weijia Shi, Amanpreet Singh, Joseph Chee Chang, Kyle Lo, Luca Soldaini, Sergey Feldman, Mike D'arcy, David Wadden, Matt Latzke, Minyang Tian, Pan Ji, Shengyan Liu, Hao Tong, Bohao Wu, Yanyu Xiong, Luke Zettlemoyer, Graham Neubig, Dan Weld, Doug Downey, Wen tau Yih, Pang Wei Koh, and Hannaneh Hajishirzi. 2024. OpenScholar: Synthesizing Scientific Literature with Retrieval-augmented LMs. arXiv:2411.14199 [cs.CL] doi:10.48550/arXiv.2411.14199
[9] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, Vol. 235. PMLR.
[10] Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, Anima Anandkumar, Abedelkadir Asi, and Junjie Hu. 2025. R-KV: Redundancy-aware KV Cache Compression for Training-Free Reasoning Models Acceleration. *Advances in Neural Information Processing Systems* (2025).
[11] Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. 2025. DeepResearch Bench: A Comprehensive Benchmark for Deep Research Agents. *arXiv preprint* (2025).
[12] Jiangfei Duan, Runyu Lu, Haojie Duanmu, et al. 2024. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *Forty-first International Conference on Machine Learning (ICML)*.
[13] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 135–153.
[14] Lorenzo Giusti, Ole Anton Werner, Riccardo Taiello, Matilde Carvalho Costa, Emre Tosun, Andrea Protani, Marc Molina, Rodrigo Lopes de Almeida, Paolo Cacace, Diogo Reis Santos, and Luigi Serio. 2025. Federation of Agents: A Semantics-Aware Communication Fabric for Large-Scale Agentic AI. *arXiv preprint arXiv:2509.20175* (2025).
[15] Liangxuan Guo, Bin Zhu, Qingqian Tao, Kangning Liu, Xun Zhao, Xianzhe Qin, Jin Gao, and Guangfu Hao. 2025. Agentic Lybic: Multi-Agent Execution System with Tiered Reasoning and Orchestration. *arXiv preprint arXiv:2509.11067* (2025).
[16] Junhao Hu, Wenrui Huang, Haoyi Wang, Weidong Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. 2024. EPIC: Efficient Position-Independent Context Caching for Serving Large Language Models. *arXiv preprint arXiv:2410.15332* (2024).
[17] Zixiao Huang, Wen Zeng, Tianyu Fu, Tengxuan Liu, Yizhou Sun, Ke Hong, Xinhao Yang, Chengchun Liu, Yan Li, Quanlu Zhang, Guohao Dai, Zhenhua Zhu, and Yu Wang. 2025. Reducing Latency of LLM Search Agent via Speculation-based Algorithm-System Co-Design. *arXiv preprint arXiv:2511.20048* (2025).
[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*.
[19] Artjom Joosen, Arjun Agarwal, Cristina L. Abad, Gregory Van Seghbroeck, Sam Deckers, Alexander Lemmens, and Mohammad Shahrad. 2025. Serverless Cold Starts and Where to Find Them. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*. doi:10.1145/3689031.3696073
[20] Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W Lee, Sangdoo Yun, and Hyun Oh Song. 2025. KVzip: Query-Agnostic KV Cache Compression with Context Reconstruction. *Advances in Neural Information Processing Systems* (2025).
[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.
[22] Yueying Li, Jim Dai, and Tianyi Peng. 2025. Throughput-Optimal Scheduling Algorithms for LLM Inference and AI Agents. *arXiv preprint arXiv:2504.07347* (2025).
[23] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: efficient serving of LLM-based applications with semantic variable. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI)*. USENIX Association, USA, Article 50, 17 pages.
[24] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 1505–1519.
[25] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. 2024. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *2024 USENIX Annual Technical Conference (ATC)*. USENIX Association, 1–17.
[26] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. *arXiv preprint arXiv:2502.13965* (2025).
[27] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, 303–320.
[28] Manus. 2025. Manus: Hands On AI. https://manus.im/.
[29] Cynthia Marcelino, Leonard Guelmino, Thomas Pusztai, and Stefan Nastic. 2025. Databelt: A Continuous Data Path for Serverless Workflows in the 3D Compute Continuum. *Journal of Systems Architecture* (2025). arXiv:2508.15351
[30] Cynthia Marcelino and Stefan Nastic. 2024. Truffle: Efficient Data Passing for Data-Intensive Serverless Workflows in the Edge-Cloud Continuum. In *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*. 53–62. doi:10.1109/UCC63386.2024.00017
[31] Moonshot AI. 2025. *Kimi-Researcher: End-to-End RL Training for Emerging Agentic Capabilities.* https://moonshotai.github.io/Kimi-Researcher/
[32] Daniel Nichols, Prajwal Singhania, Charles Jekel, Abhinav Bhatele, and Harshitha Menon. 2025. Optimizing Agentic Language Model Inference via Speculative Tool Calls. *arXiv preprint arXiv:2512.15834* (2025).

[33] OpenAI. 2025. Introducing Deep Research. https://openai.com/index/introducing-deep-research/.

[34] Zaifeng Pan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. *arXiv preprint arXiv:2507.07400* (2025).

[35] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. arXiv:2405.04437 [cs.LG]

[36] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Yang Wang, Miryung Kim, and Harry Xu. 2024. ConServe: Harvesting GPUs for Low-Latency and High-Throughput Large Language Model Serving. *arXiv preprint arXiv:2410.01228* (2024).

[37] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction. *arXiv preprint arXiv:2404.08509* (2024).

[38] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 205–218.

[39] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 814–827. doi:10.1109/HPCA56546.2023.10071120

[40] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2024. Pre-Warming is Not Enough: Accelerating Serverless Inference With Opportunistic Pre-Loading. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC)*. Association for Computing Machinery, New York, NY, USA, 178–195.

[41] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2026. Accelerating ML Inference via Opportunistic Pre-Loading on Serverless Clusters. *IEEE Transactions on Parallel and Distributed Systems* 37, 2 (2026), 472–488.

[42] Kyle Swanson, Wesley Wu, Nash L. Bulaong, John E. Pak, and James Zou. 2025. The Virtual Lab of AI agents designs new SARS-CoV-2 nanobodies. *Nature* 646, 8085 (Oct. 2025), 716–723. doi:10.1038/s41586-025-09442-9 Epub 2025-07-29.

[43] Tongyi DeepResearch Team. 2025. Tongyi DeepResearch: A New Era of Open-Source AI Researchers. https://github.com/Alibaba-NLP/DeepResearch.

[44] Ying Wang, Zhen Jin, Jiexiong Xu, Wenhai Lin, Yiquan Chen, and Wenzhi Chen. 2025. AugServe: Adaptive Request Scheduling for Augmented Large Language Model Inference Serving. *arXiv preprint arXiv:2512.04013* (2025).

[45] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Wang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023).

[46] Hao Wu, Junxiao Deng, Minchen Yu, Yue Yu, Yaochen Liu, Hao Fan, Song Wu, and Wei Wang. 2024. FaaSTube: Optimizing GPU-oriented Data Transfer for Serverless Computing. *arXiv preprint arXiv:2411.01830* (2024).

[47] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. 2025. Strata: Hierarchical Context Caching for Long Context Language Model Serving. *arXiv preprint arXiv:2508.18572* (2025).

[48] Hongshen Xu, Zihan Wang, Zichen Zhu, Lei Pan, Xingyu Chen, Shuai Fan, Lu Chen, and Kai Yu. 2025. Alignment for Efficient Tool Calling of Large Language Models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 17776–17792.

[49] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. 2025. KVLink: Accelerating Large Language Models via Efficient KV Cache Reuse. *arXiv preprint arXiv:2502.16002* (2025).

[50] Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. 2025. Speculative Actions: A Lossless Framework for Faster Agentic Systems. *arXiv preprint arXiv:2510.04371* (2025).

[51] Yi Zhai, Dian Shen, Junzhou Luo, and Bin Yang. 2026. ToolCaching: Towards Efficient Caching for LLM Tool-calling. *arXiv preprint arXiv:2601.15335* (2026).

[52] Chaoyun Zhang, Liqun Li, He Huang, Chiming Ni, Bo Qiao, Si Qin, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2025. UFO^3: Weaving the Digital Agent Galaxy. *arXiv preprint arXiv:2511.11332* (2025).

[53] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2024. MuCache: A General Framework for Caching in Microservice Graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 221–238.

[54] Haoran Zhang, Shuai Mu, Sebastian Angel, and Vincent Liu. 2025. CausalMesh: A Causal Cache for Stateful Serverless Computing. *Proceedings of the VLDB Endowment* (2025).

[55] Lei Zhang, Mouxiang Chen, Ruisheng Cao, Jiawei Chen, Fan Zhou, Yiheng Xu, Jiaxi Yang, Zeyao Ma, Liang Chen, Changwei Luo, Kai Zhang, Fan Yan, KaShun Shum, Jiajun Zhang, Zeyu Cui, Feng Hu, Junyang Lin, Binyuan Hui, and Min Yang. 2026. MegaFlow: Large-Scale Distributed Orchestration System for the Agentic Era. *arXiv preprint arXiv:2601.07526* (2026).

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*.