MAESTRO: Multi-Agent Evaluation Suite for Testing, Reliability, and Observability

Tie Ma*†
Beihang University

Yixi Chen*
KAUST

Vaastav Anand
MPI-SWS

Alessandro Cornacchia
KAUST

Amândio R. Faustino
KAUST

Guanheng Liu
Beihang University

Shan Zhang
Beihang University

Hongbin Luo
Beihang University

Suhaib A. Fahmy
KAUST

Zafar A. Qazi
LUMS & KAUST

Marco Canini
KAUST

Abstract

Large language model (LLM)-based multi-agent systems (MAS) are rapidly moving from demos to production, yet their dynamic execution makes them stochastic, failure-prone, and difficult to reproduce or debug. Existing benchmarks largely emphasize application-level outcomes (e.g., task success) and provide limited, non-standardized visibility into execution behavior, making controlled, apples-to-apples comparisons across heterogeneous MAS architectures challenging.

We present MAESTRO, an evaluation suite for the testing, reliability, and observability of LLM-based MAS. MAESTRO standardizes MAS configuration and execution through a unified interface, supports integrating both native and third-party MAS via a repository of examples and lightweight adapters, and exports framework-agnostic execution traces together with system-level signals (e.g., latency, cost, and failures). We instantiate MAESTRO with 12 representative MAS spanning popular agentic frameworks and interaction patterns, and conduct controlled experiments across repeated runs, backend models, and tool configurations. Our case studies show that MAS executions can be structurally stable yet temporally variable, leading to substantial run-to-run variance in performance and reliability. We further find that MAS architecture is the dominant driver of resource profiles, reproducibility, and cost–latency–accuracy trade-off, often outweighing changes in backend models or tool settings. Overall, MAESTRO enables systematic evaluation and provides empirical guidance for designing and optimizing agentic systems.

1 Introduction

LLM-based multi-agent systems (MAS) enable flexible task solvers that can handle diverse and multimodal workloads [20] with minimal modification to the underlying system architecture. However, this flexibility also introduces substantial uncertainty in system load and execution behavior. Unlike traditional deterministic workflows, LLM-based MAS operate under a dynamic execution model in which decisions are made on the fly during runtime, driven by LLM outputs rather than by statically defined control flow.

Importantly, MAS should not be viewed merely as a collection of lightweight client-side frameworks. Instead, they constitute complex systems characterized by dynamic interactions [53], emergent behaviors [20], and a broad spectrum of failure modes [5]. These characteristics challenge conventional assumptions such as predictability, observability, and performance isolation, making traditional system optimization techniques less effective in this context. Therefore, a benchmark suite that systematically characterizes MAS execution behavior is essential for both system operators seeking performance optimization and researchers aiming to identify open challenges and opportunities for innovation.

Unfortunately, existing standardized benchmarks for LLM-based MAS remain limited and often lack broad coverage of MAS execution behavior. Prior work has largely focused on LLM serving and inference efficiency [3, 7, 23, 31, 32, 50], evaluating server-side model performance rather than the execution behavior of agent systems. With the emergence of LLM-based MAS, recent benchmarks [2, 4, 15, 22, 26, 35, 47, 49, 53, 56] have begun to assess individual agent capabilities (e.g., tool use and communication strategies); however, they largely remain centered on application-level performance (e.g., task success and response quality) and fall short of offering a standardized, comprehensive observability perspective on the system-level impact of MAS execution and corresponding workload management challenges. This fragmentation makes it difficult to reason about complex runtime behavior and to compare systems consistently across settings. Consistent with this gap, a recent survey [41] reports that nearly 75% of teams operating production MAS evaluate their systems without benchmarks, while 25% build custom benchmarks, limiting portability and reuse across scenarios.

Based on these observations, we define the following core objectives necessary for a good benchmark for LLM-based MAS:

O1: Architectural heterogeneity. The execution stack of LLM-based MAS is highly malleable. A single objective can be realized through diverse configurations, including the number of agents, role assignments, interaction topologies (e.g., centralized, hierarchical, or peer-to-peer), and communication protocols. Furthermore, the design space encompasses choices regarding orchestration frameworks, backend LLMs, budget constraints, tool availability, and memory mechanisms, as well as policies for reflection and termination.

O2: Functional representativeness. The rapid proliferation of agentic workflows and real-world deployments has led to a growing diversity of MAS architectures, many of which are optimized for

---

*Equal contribution.
†Work done while Tie Ma was interning at KAUST.

T. Ma, Y. Chen, et al.

Table 1: **Summary of Systematic Findings across Case Studies (§4)**

Subject	Ref.	Finding
Resources	§4.2	Execution requires minimal resources: sub-GB memory, <20% of a CPU core, and MB-scale traffic.
General	§4.3	Interaction structures remain stable while call sequences exhibit temporal instability.
	§4.7	Tool integration mitigates speculative generation, reducing latency and cost.
Backend	§4.5	Model scaling yields inconsistent gains; execution dynamics dominate performance.
	§4.6	Model-specific failures are significantly amplified by execution dynamics.
Architecture	§4.2	MAS architecture significantly dominates resource consumption profiles.
	§4.3	Architecture governs call graph similarity and determines system reproducibility.
	§4.4	Generalized architectures incur higher resource overhead without accuracy gains.
	§4.7	Accuracy gains are architecture-dependent and contingent on low execution overhead.

specific task patterns or application domains. Recent designs explore increasingly sophisticated coordination and reasoning strategies [45, 52, 57, 58]. As a result, no single architecture can be considered representative of the broader MAS design space.

O3: Execution traceability. Current commercial agentic systems often expose high-level reasoning traces but offer limited, non-standardized visibility into execution-level details and internal system states. Furthermore, existing MAS modules lack a unified telemetry standard, often resulting in "silent" information consumption where different LLM providers and frameworks fail to expose critical operational data to the user [44].

To address this gap, we present MAESTRO, a comprehensive, open-source evaluation suite for LLM-based MAS. MAESTRO is designed to enable systematic characterization of execution behavior across diverse agent architectures, interaction patterns, and runtime conditions, with the goal of informing principled system optimization.

Our contributions are threefold:

Rich and extensible benchmarks. MAESTRO incorporates 12 representative MAS examples, each characterized by distinct architectural differences, to serve as a foundation for deriving systematic insights, as shown in Table 1. Moreover, MAESTRO is designed for extensibility, allowing the community to integrate and reuse existing MAS implementations within our evaluation framework with minimal effort.

Framework-agnostic system integration. MAESTRO is built upon a collection of widely used, open-source agentic frameworks and examples [10, 17, 25], aiming to capture common architectural patterns observed in practice rather than favoring a single workflow design.

Unified execution-level telemetry standards. MAESTRO defines and implements a unified telemetry interface designed to capture comprehensive execution data across diverse modules. This architecture establishes a common protocol that various MAS components can conform to, ensuring consistent and transparent monitoring throughout the system lifecycle.

MAESTRO is available at https://github.com/sands-lab/maestro.

2 Background

2.1 Anatomy of an LLM-based MAS

LLM-based Multi-Agent Systems (MAS) are collections of LLM agents that operate in tandem to complete large tasks that are beyond the capabilities of individual agents [21]. In a typical MAS, multiple specialized agents collaborate together to plan, coordinate, and execute large tasks with each individual agent focusing on a specific sub-task.

Building blocks. An LLM agent is an entity that autonomously executes multi-step tasks by combining generative foundational models with external tools, memory, and reasoning and planning capabilities [33]. Agents are designed to operate autonomously in highly dynamic environments where adaptability and strategic decision making are essential. Each agent is comprised of four key parts: (i) inputs that may include user instructions, developer-specified constraints, multimodal observations, retrieved knowledge, and internal state; (ii) a generative Large Language Model (LLM) that maps the current state to decisions; (iii) an action interface that enables tool interactions such as data retrieval, API calls, code execution; (iv) outputs including user-facing responses and structured actions and artifacts along with updated state.

Orchestration and deployment. Practitioners orchestrate MAS through workflows written in agentic programming frameworks such as LangGraph [25], CrewAI [24], AutoGen [10], LlamaIndex [36], and Agno [37]. Despite the popularity of these third-party frameworks according to surveys [39, 41], detailed interviews with practitioners revealed that practitioners preferred to build agentic applications from scratch in 85% of the cases [41]. These workflows may be static or dynamic depending on the degree of autonomy allowed by developers in these systems. Currently, MAS are deployed as single monolithic applications; however, they are increasingly developed and deployed as distributed applications [9, 46].

Workflow structure. MAS workflows often follow a hierarchical structure with task structures as a tree of sub-tasks. Individual subtasks follow a mix of sequential and parallel flows. Workflows may also contain recursive calls for individual agents [39].

Failure types. MAS applications showcase three main failure types — System Design Issues, Inter-Agent Misalignment, Task Verification [5]. System design issues include configuration issues, API and system issues, and resource mismanagement. Inter-agent misalignment issues result from a breakdown in critical information flow from inter-agent interaction and coordination during execution. This includes planning and coordination errors, incorrect output generation, individual LLM hallucinations, and incorrect information processing. Task Verification failures arise when verification strategies are inadequate at identifying issues.

Sources of non-determinism. Due to the dynamic and heterogeneous nature of MAS applications, they exhibit non-determinism due to a multitude of reasons. First, LLMs are stochastic in nature and often produce different outputs for the same input. Second, external tool executions are not pre-planned or programmed. Additionally, tools may produce non-deterministic results. Third, workflows are dynamic and change at runtime [53]. Non-determinism in dynamic workflows may further be exacerbated due to the availability of agents. Fourth, built-in reliability mechanisms impact the performance and structure of MAS executions. For example, quality-driven retries change the execution graph.

Reliability as a first-class citizen. Typical MAS applications treat reliability as a first-class citizen as part of the design and implementation of these systems. They do so in multiple ways. First, most MAS applications rely on Human-in-the-loop evaluation, with almost half of the applications executing fewer than five steps before seeking human-in-the-loop evaluation [41]. In addition, developers often augment applications with LLM-as-a-judge to automate quality checks. MAS applications also automate retries to improve quality if quality checks fail. Second, practitioners prioritize quality over real-time responsiveness, with 66% of respondents to a recent survey allowing response times of more than a minute [41]. Third, practitioners prefer static workflows over dynamic workflows to constrain the autonomy of deployed agents [41].

2.2 Limitations of existing benchmarks

Evaluating and benchmarking the performance of Large Language Models has been an important aspect in measuring the efficiency and efficacy of LLMs at executing real-world tasks [3, 23, 31, 32]. With the recent rise of LLM agents and MAS applications, benchmarking the performance of agentic systems has garnered a great deal of interest from the scientific community.

Agent benchmarks. Typical agent benchmarks evaluate capabilities of individual LLMs as agents [4, 27, 35]. These benchmarks have been further extended to multi-agent settings. To do so, researchers have developed specialized benchmarks that evaluate a specific property of agentic systems such as Tool Calling [22, 56], Task Planning [15], communication strategies [53], sequential flows [2], privacy preservation [26], and collaboration efficacy [47, 49]. Such specialized benchmarks solely focus on one specific property or dimension of MAS applications and lack the holistic view required to effectively understand the end-to-end emergent behavior and performance of MAS applications.

Bespoke benchmarks. Due to the lack of standardization and the diversity of MAS design space, MAS application developers instead opt to create custom benchmarks specific to their application. For example, authors of Autogen [51] created a bespoke benchmark called Autogenbench [1] for tasks developed in the Autogen framework. According to a recent survey, 25% of teams for production MAS applications construct custom benchmarks for their applications, 75% of teams evaluate their agents without formal benchmarks and instead rely on A/B testing and direct expert/user feedback [41]. Although these benchmarks are suitable for a given specific application, such benchmarks do not capture the diversity of the MAS design space and do not provide insight in a broad setting.

Observability tools and benchmarks. Observability tools and observability-based benchmarks such as Opik [8], TRAIL [11], TAMAS [39] capture spans and traces of MAS executions which developers use to further analyze traces to triage issues and to understand MAS executions. Beyond standard metrics, such as agent call frequency, external API usage, and per-call token costs, there remains a significant gap in deep application-semantic telemetry. Addressing this requirement involves capturing granular retry logic details (e.g., attempt counts, triggers, and parent span IDs), agent-specific status conditions (e.g., failure categorization and error reasoning), and output quality assessments. Such telemetry is essential for providing the execution-level transparency needed to diagnose stochastic failures and understand complex multi-agent interactions.

3 MAESTRO

We present MAESTRO, a Multi-Agent Evaluation Suite for Testing, Reliability, and Observability, as a comprehensive framework for evaluating LLM-based MAS. Building upon goals, we first outline in §1, we detail the architecture and design of the framework (§3.1), illustrating how standalone MAS implementations are adapted and integrated into our suite. To demonstrate MAESTRO's capacity for generating informative telemetry, we present a collection of representative MAS instances (i.e., the concrete evaluation units in a benchmark suite). These are categorized according to our proposed taxonomy (§3.2), while §3.3 details the specific instances used and the formulation of evaluation suites designed to derive our experimental findings.

3.1 Benchmark design

3.1.1 MAESTRO architecture. Figure 1 presents an overview of the MAESTRO architecture. Conceptually, MAESTRO follows a linear control flow: preparation of MAS instances, a user-defined configuration specifies how a MAS is instantiated and executed, execution traces are collected during runtime, and post-hoc processing transforms these traces into interpretable metrics and summaries. The workflow consists of five core components:

MAS instances preparation. To use MAESTRO, users first need to prepare MAS instances to be evaluated. The details of the preparation process and the supported integration modes are described in §3.1.2.

Configuration. Based on the prepared MAS instances, users specify the evaluation setup, including which input sources, the number and configuration of agent instances, and whether external tool
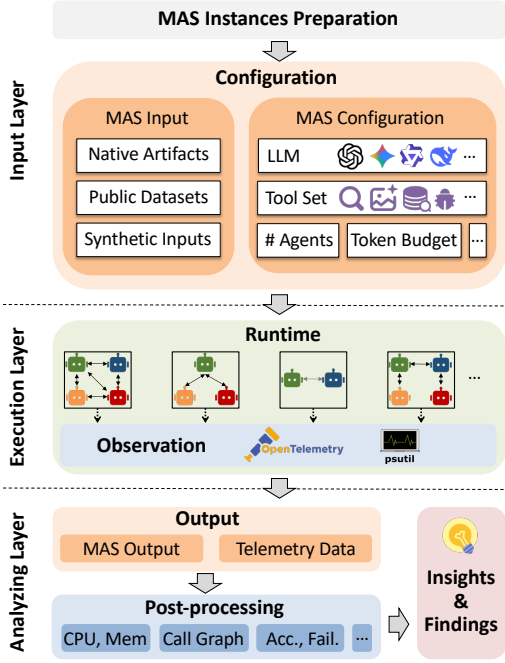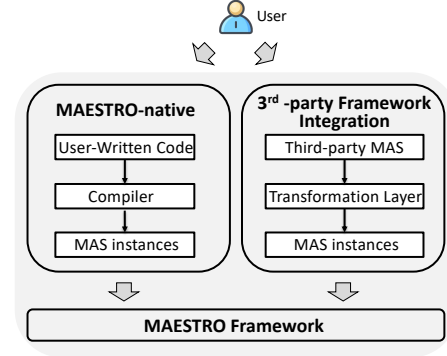
Figure 1: **MAESTRO architecture overview.**

Figure 2: **Two ways to prepare MAS instances for MAESTRO, note that MAESTRO ships with a set of built-in MAS instances that can be used and compared directly.**

access is enabled[1]. This configuration is passed to the *Runtime* component for system instantiation and execution.

Runtime. Based on the provided configuration, *Runtime* component orchestrates the execution of the MAS instances. Task inputs are continuously fed into the runtime, triggering agent interactions, tool invocations, and control-flow decisions as defined by the configuration.

Observation. During execution, the *Observation* component monitors system behavior through function-call hooks or sampling-based instrumentation. Built on top of OpenTelemetry [40] and psutil [43], it records both default execution metrics (e.g., latency, token usage; see §A.2.1) and additional signals specified in the configuration. Collected traces are forwarded, either online or offline, to the *Post-processing* component.

Post-processing. The *Post-processing* component aggregates and analyzes execution traces to make MAS behavior inspectable (e.g., CPU, call graph; see §A.1). These summaries enable users to explore execution trajectories, compare configurations across runs, and identify performance bottlenecks and sources of instability.

3.1.2 MAS instances preparation. Before performing any evaluation, MAS instances must be integrated into MAESTRO. As illustrated in Figure 2, users can prepare MAS instances in two ways:[2]

- **MAESTRO-native**. Users can implement MAS instances directly using MAESTRO's native specification language and configuration interfaces. This mode leverages compile-based techniques [9] to automatically generate executable instances from high-level

descriptions, ensuring optimal compatibility. This mode reduces manual coding effort by generating reusable scaffolding and integration code for common components.

- **Third-party framework integration**. Through MAESTRO's transformation layer, users can build MAS instances in their preferred agent frameworks (e.g., ADK, LangGraph, AutoGen) or import existing open-source implementations, and connect them to MAESTRO for evaluation. The transformation layer provides a set of adapters that map framework-specific components onto MAESTRO's standard interfaces, exposing unified entry points for configuration, execution, and telemetry collection.

MAESTRO contributors can also use these two integration modes to add new built-in MAS instances to the framework. Currently, MAESTRO has 12 built-in MAS instances (described in detail in §3.3), allowing users to perform evaluations and comparisons directly without additional integration effort.

3.2 MAS instances taxonomy

A well-designed benchmark should cover a broad (**O1**) and representative (**O2**) range of system configurations and use cases; otherwise, conclusions may overfit to a narrow slice of the MAS design space and fail to generalize. To enable systematic coverage and controlled comparisons, we characterize each MAS instance using a small set of well-defined dimensions. Specifically, we describe each instance along the following axes: application field, framework, interaction pattern, and data specification. These dimensions together capture the primary sources of variation in modern MAS deployments.

Application field. The high-level domain that the MAS instance targets, which may influence task complexity, required agent capabilities, and evaluation criteria. Common fields include question answering, creative generation, finance, and others.

Framework. The underlying multi-agent framework used to implement the MAS instance, which may affect agent orchestration, communication protocols, and tool integration. General frameworks include AutoGen [51], ADK [17], LangGraph [25], and others.

Interaction pattern. The specific configuration of agents within the MAS instance, including the number of agents, the number

---

[1]Currently, MAESTRO only supports the adjustment of a few parameters, such as model choice and tool usage.

[2]At present, MAESTRO supports only *pre-defined* MAS instances.

of tools, and the cooperation type. Specifically, cooperation types include:

- *Planning*. There is a dedicated planning agent that decomposes the task into subtasks and assigns them to other agents.
- *Coordination*. Agents coordinate their actions through explicit communication.
- *Debate*. Agents evaluate and compare candidate solutions to reach a consensus.
- *Correction*. Agents collaboratively refine and improve a specific solution through iterative feedback.

These interaction patterns could affect the overall system dynamics and performance.

Data specification. The concrete input-output format and ground truth used to instantiate the MAS instance, which may influence task complexity, communication pattern, and evaluation criteria. The data specification could be divided into input and output.

- *Input.* A task is typically instantiated via a system prompt defining its core objectives and constraints. For tasks requiring open-ended exploration, the configuration phase may also incorporate external information retrieved through auxiliary tools, such as web search engines or private databases, as supplemental inputs. Collectively, we define these structured inputs and retrieved data as *artifacts*.
- *Output.* According to whether the output has a determined ground truth, the output could be divided into *Open-End* and *Closed-Form*.

3.3 MAS example suites studied

We carefully select 12 representative MAS instances to serve as the pre-defined evaluation set in MAESTRO. These instances are designed to provide sufficient coverage of common MAS configurations and use cases (**O1, O2**), and to act as a baseline for subsequent studies. As summarized in Table 2, the selected instances are chosen according to the following criteria:

- **Framework diversity (O2, O3)**: We include examples implemented using different popular MAS frameworks, such as *MCP-Agent*, *LangGraph*, *ADK*, and *Autogen*, to capture a wide range of design patterns and interaction paradigms.
- **Official sources (O2)**: We collect examples that are provided in the official example repositories or tutorials of these frameworks, ensuring that they reflect best practices and standard usage patterns.
- **Domain variety (O2)**: We select examples that cover diverse application domains, including question answering, planning, creative writing, marketing strategy, and so on, to evaluate MAS performance across different application scenarios.
- **Interaction diversity (O1)**: We prioritize examples that exhibit varied interaction patterns among agents, such as cooperative problem solving, debate-style discussions, and role-based collaborations, to assess how different interaction styles affect MAS behavior.

 As a prerequisite for meaningful analytical post-processing, MAS instances must be grouped into coherent categories that share relevant characteristics. Such grouping enables comparative analysis across multiple configurations, ensuring that observed behaviors
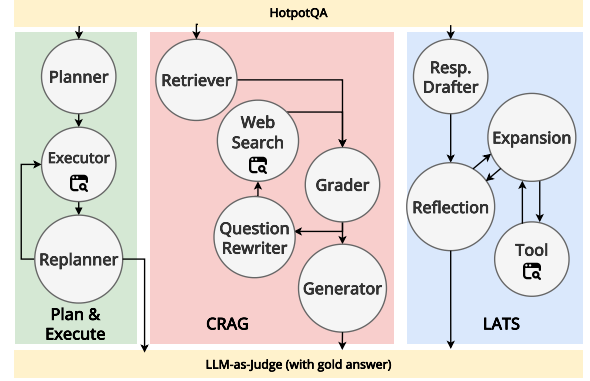

Figure 3: Solving the same given tasks with 3 different MAS architectures.

reflect systematic trends rather than ad hoc artifacts of individual runs. To demonstrate the analytical capabilities of MAESTRO, we derive two evaluation suites, each designed to surface distinct system-level insights.

- **Full-suite (F)**: This suite includes all selected MAS examples. We treat this suite as a representative subset of real-world MAS deployments, and use it to study the overall performance and behavior of LLM-based MAS in realistic settings.
- **Architecture-focused suite (A)**: This suite includes three representative MAS examples that implement different representative multi-agent architectures but solve the same set of tasks. This suite is used to study the impact of agent architectures on MAS behavior.

 In the architecture-focused suite, we select three representative MAS architectures: CRAG (Corrective RAG) [54], Plan&Execute [48], and LATS (Language Agent Tree Search) [59]. They are designed as general-purpose agent architectures that can operate across tasks without being tightly coupled to specific applications. However, their design goals differ. As shown in Figure 3, CRAG is optimized for retrieval-centric workloads, whereas LATS and Plan-and-Execute target more general problem-solving settings, employing tree-search–based divide-and-conquer and greedy iterative refinement strategies, respectively. Such variety in the task-solver architectures enables the following comparative studies.

 In the following section, we present case studies and analyses using MAESTRO, organized around these two MAS example suites.

4 Case studies

To demonstrate how researchers can benefit from MAESTRO, we conduct a series of case studies that illustrate the types of insights enabled by its fine-grained telemetry. We organize our evaluation into two complementary sets of case studies:

- **General system-level analysis.** In §4.2 and §4.3, we examine system-level metrics that are not specific to MAS, but instead provide a familiar baseline for reasoning about performance, resource consumption, and reliability, analogous to evaluations in traditional systems.

Table 2: **Selected MAS examples overview. The "Suite" column indicates membership, F: Full Suite; A: Architecture Suite.**

Example	App. Field	Framework	Interaction			Data Spec.		Suite
			Type	#Agt	#Tool	In	Out	
Fin. Analyzer [29]	Finance	MCP-Agent	Correct	6	1	Artifacts	Opn-End	F
Img. Scr. [16]	Creativity	ADK	Debate	4	2	Artifacts	Cls-Form	F
Marketing [16]	Marketing	ADK	Coord.	4	1	Artifacts	Opn-End	F
Brand SEO [16]	Marketing	ADK	Coord.	4	10	Artifacts	Opn-End	F
Content Creat. [42]	Creativity	ADK	Plan.	4	1	Artifacts	Opn-End	F
Mag.-One [14]	Cross-domain	Autogen	Plan	4	0	Artifacts	Opn-End	F
Stock Res. [10]	Finance	Autogen	Coord.	4	2	Artifacts	Opn-End	F
Travel Plan. [38]	Travel	Autogen	Coord.	4	0	Artifacts	Opn-End	F
ToT [57]	Cross-domain	LangGraph	Debate	3	0	Artifacts	Cls-Form	F
CRAG [54]	Cross-domain	LangGraph	Coord.	5	2	Datasets	Opn-End	F,A
Plan&Exec. [48]	Cross-domain	LangGraph	Plan	3	1	Datasets	Opn-End	F,A
LATS [59]	Cross-domain	LangGraph	Plan	3	1	Datasets	Opn-End	F,A

- **Application- and semantics-aware analysis.** Using Evaluation Suite 2, we investigate how different MAS solver architectures affect cost, latency, and accuracy (§4.4, §4.5, §4.6, and §4.7). This analysis explores whether architectural choices and structural optimizations lead to consistent performance trade-offs, in a manner analogous to ablation studies.

4.1 Methodology

Inputs. For each MAS instance, we generate evaluation inputs using one of the following three approaches:

- **Naive artifacts.** Direct reuse of input prompts provided in the README files of official example repositories.
- **Public datasets.** Inputs drawn from publicly available benchmark datasets aligned with the task domain of the MAS instance (e.g., question-answering datasets for QA-oriented agents [55]).
- **Synthetic inputs.** LLM-generated prompts that enable controlled variation and increased input diversity.

Setup. For each MAS instance, we conduct at least 20 independent runs to characterize execution behavior. Each run consists of submitting a single user-level task input to the MAS (e.g., a single "write a blog post" prompt for *Content Creation*). For MAS instances that require human-in-the-loop interaction, user responses are simulated using an LLM-as-user approach, where a designated LLM (`gemini-2.5-flash` in our current setup) generates replies conditioned on the MAS outputs. To prevent non-terminating execution, each run is capped at 10 minutes. For the architecture-focused suite, LLM responses are additionally limited to a maximum of 8,192 tokens. As for the external tool usage mentioned in Table 2, we use Tavily or Google Search [18] for web search, and use Google `imagen-3.0-generate-002` [19] model for image generation. When correctness evaluation is required, we employ `gpt-4o-mini` as an LLM-as-judge. To evaluate the impact of different backbone models, we vary the underlying LLM across several

configurations: Gemini-2.0-Flash-Lite (Ge20FL), Gemini-2.5-Flash-Lite (Ge25FL), Gemini-2.5-Flash (Ge25F), GPT-4o-mini (G4oM), GPT-5-mini (G5M), and GPT-5-nano (G5N).

4.2 What are the systems usage patterns and implications?

Resource consumption such as CPU, memory, and network usage are important factors to consider when deploying MAS in real-world systems. Understanding the resource usage patterns of MAS can help optimize their performance and scalability. In this subsection, we analyze the resource consumption of MAS and investigate the factors that influence their usage patterns.

Per-task CPU and memory footprints are modest and bounded in our setup. Figure 4 reports per-run CPU and memory usage across the 12 MAS examples under our runtime configuration. For CPU, the maximum observed utilization reaches 61.9%, while the boxplot distributions for most examples remain substantially below this peak, suggesting that these workloads typically do not require sustained heavy local compute. For memory, excluding the Content Creation example which peaks at 1726.8 MB, the average memory usage across examples is 200.2 MB, placing most single-task executions in a sub-GB regime in our measurements. The Content Creation example's higher memory footprint stems from its distributed design, where each agent runs in a separate process, increasing the aggregate resident memory. We leave a broader study of distributed MAS deployments and their resource trade-offs to future work. We also observe framework-dependent memory patterns; for example, examples implemented with *ADK* tend to exhibit higher memory footprints in our setup.

Per-task communication volume is in the MB scale and varies by architecture and model. We further measure total communication volume across architectures with and without tools (Figure 5). Across all tested configurations, the observed communication volumes stay within the MB scale (with most cases within a few MB), indicating that, per task, network payload is typically small compared
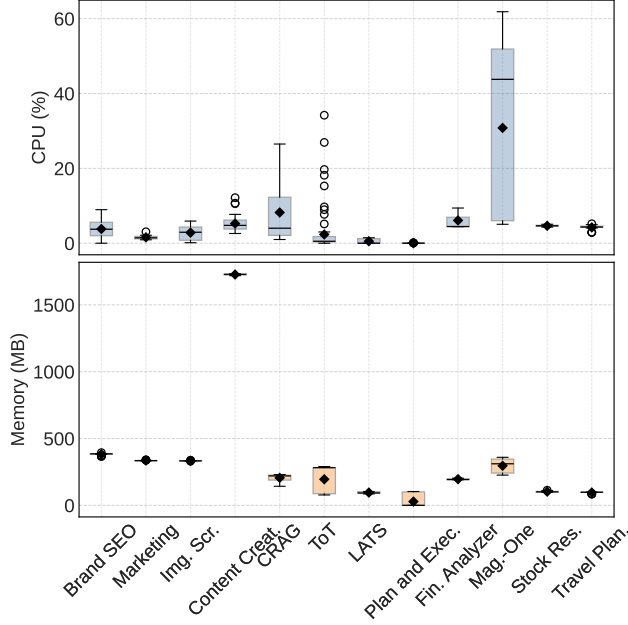
Figure 4: **CPU and memory usage across different examples.**

to CPU and memory footprints. It can be observed that tool usage has a minor impact on communication volume. Model choice also interacts with architecture; for example, in CRAG, Gemini-family models show larger communication volumes than GPT-family models in our measurements.

> *Finding 1:* In our setup, most single-task executions stay within a sub-GB memory regime and bounded CPU utilization, while per-request communication remains at the MB scale.

CPU and memory usage patterns are architecture-dependent.
Figure 6 illustrates the CPU and memory usage grouped by architecture. Consistent with the communication patterns observed in Figure 5, resource consumption is highly architecture-dependent. CRAG exhibits the highest resource footprint, with an average CPU usage of 9.7% and memory usage of 405.3MB (averaged over all model and tool configurations). This is followed by Language Agent Tree Search (1.36% CPU) and Plan-and-Execute (0.07% CPU). We further observe that while model choice influences CPU load, it has a negligible impact on memory. Surprisingly, enabling tools reduces global average CPU usage by 3.1% and memory by 4.8MB. This phenomenon could be explained by the fact that tool usage could help reduce the number of LLM calls, which are CPU and memory-intensive.

> *Finding 2:* Architecture dominates resource patterns, while model choice introduces smaller shifts.

4.3 How stable are MAS call graphs, and what factors influence their variability?

A distinguishing characteristic of LLM-based MAS, in contrast to traditional systems like microservices, is the inherent stochasticity of their execution behavior. In microservice architectures, call graph variability is widely used as an indicator of anomalous executions and edge cases. By extending this concept to MAS, one can leverage variability to sample a diverse and representative set of execution traces. However, a foundational stability analysis is a prerequisite for such methodologies. From a reproducibility perspective, higher call graph similarity across repeated runs implies stronger run-to-run consistency in agent interactions, and thus more reproducible MAS executions. Therefore, quantifying the stability of agent interactions across different runs and identifying the factors influencing their variability are crucial for designing robust and reproducible MAS.

We use two metrics to measure the call graph similarity: Jaccard similarity and Largest Common Sequence (LCS) similarity. These metrics capture different aspects of the call graph structure and provide insights into the agent interactions.

- **Jaccard similarity (edge-set overlap)**: For each run i, we construct a directed call graph G_i and denote by E_i its (unweighted) edge set. For runs i, j, we compute

$$J(E_i, E_j) = \frac{|E_i \cap E_j|}{|E_i \cup E_j|},$$

  with the convention $J(E_i, E_j) = 0$ when $E_i \cup E_j = \emptyset$. This captures whether the same interaction edges appear at least once, regardless of frequency.

- **LCS (order consistency)**: For each run i, we linearize the calls into an ordered edge sequence S_i. Let $\text{LCSlen}(S_i, S_j)$ denote the length of the longest common subsequence between S_i and S_j. We define the normalized LCS similarity as

$$\text{LCS}(S_i, S_j) = \frac{\text{LCSlen}(S_i, S_j)}{\max(|S_i|, |S_j|)},$$

  with the convention that two empty sequences yield 1 and one empty sequence yields 0. This measures the consistency of interaction order.

To summarize similarity at different granularities (e.g., per example, per model, or per experimental condition such as tool-on vs. tool-off), we first partition runs into groups according to the dimension of interest. For a group with n runs, we compute similarity values for all unordered run pairs (i, j) with $1 \leq i < j \leq n$. For each pair, we compute $J(E_i, E_j)$ and $\text{LCS}(S_i, S_j)$. We define the *pairwise average similarity* for a group as the mean of these pairwise values over all unordered run pairs in that same group. If $n < 2$, we set the pairwise average similarity to 0 (no pairwise comparisons are available).

MAS execution exhibits structural stability but sequential variance. We first examine the stability of call graphs across repeated runs of the same example. Figure 7 presents the intra-example average pairwise similarity for the Full Suite. We observe that across all cases there exists high Jaccard similarities (average 0.86 across all examples), indicating that the *set of agent-to-agent interactions*
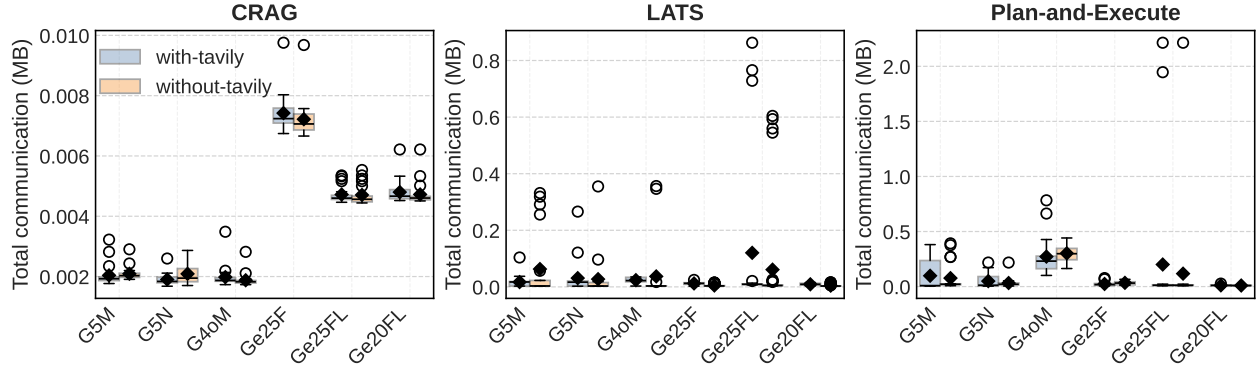
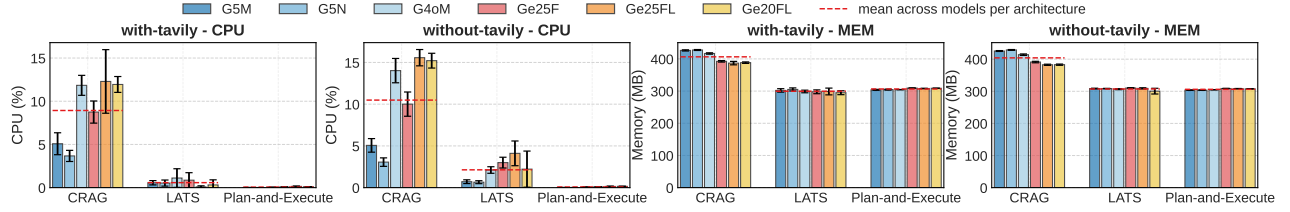Figure 5: **Communication usage across different architectures.**

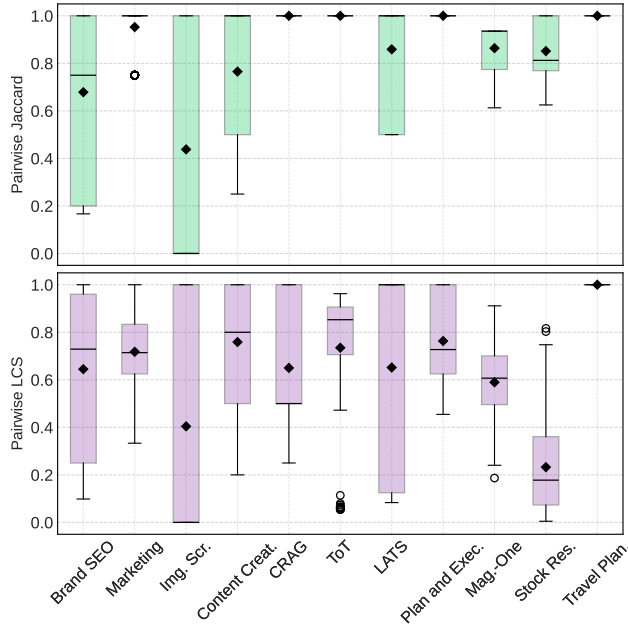Figure 6: **CPU and memory usage across different architectures.**

Figure 7: **Cross-model call graph similarity (Jaccard Similarity: Order-agnostic overlap of agent interactions, LCS Similarity: Order-aware similarity of execution traces).**
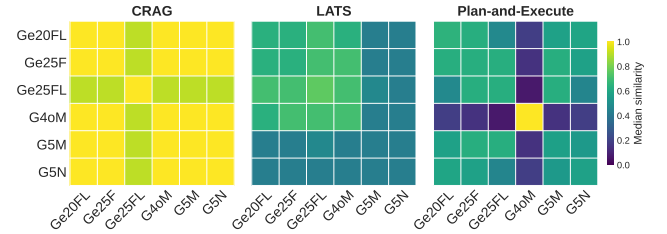

Figure 8: **Cross-model call graph LCS similarity.**

agent calls fluctuates significantly across runs. Notably, examples like CRAG and Tree-of-Thoughts demonstrate high Jaccard but low LCS scores, confirming that while the participating agents and their connections remain consistent, the temporal order of their interactions is highly dynamic. A distinct exception is the travel-planning example, which employs the *RoundRobinGroupChat* mechanism from *Autogen*; this enforces a deterministic execution order, resulting in perfect stability (1.0) for both metrics.

> *Finding 3:* Across runs, MAS call graphs are largely stable in which agent-to-agent interactions occur, but often unstable in the order those interactions unfold; consequently, reproducibility is stronger at the interaction-structure level than at the execution-order level.

remains robust against execution variance. In contrast, LCS similarity is moderate (average 0.65), suggesting that the *sequence of*

Architecture determines stability, while model impact is architecture-specific. We further investigate the factors influencing call graph similarity by comparing execution patterns across different models and architectures. Figure 8 presents the cross-model LCS call graph similarity heatmap, where each sub-figure corresponds to an architecture and each cell represents the median pairwise similarity between two models. We observe distinct stability profiles across architectures: CRAG exhibits extremely high consistency (average similarity 0.97 across all model pairs), whereas Language Agent Tree Search (0.54) and Plan-and-Execute (0.47) show significantly more variation. Furthermore, the impact of model choice is architecture-dependent. In CRAG, most models share identical call graphs, with `gemini-2.5-flash-lite` being the sole outlier. In Language Agent Tree Search, all models produce similar call graphs. Conversely, for Plan-and-Execute, `gpt-4o-mini` diverges significantly from other models by showing low similarity to them, yet it remains highly self-consistent across its own runs, while the remaining models tend to resemble one another.

> *Finding 4:* MAS architecture dominates the call graph similarity, with model choice having different effects depending on the architecture.

We then move to application-level metrics, including cost, task duration, and accuracy. Due to the inherent non-determinism of LLMs, accuracy can vary across runs; ensuring result quality therefore often correlates with increased cost and longer execution time. MAESTRO enables systematic analysis of such behavior despite the chaotic nature of LLM-driven execution. To ensure fair comparison across configurations, the following evaluation focuses exclusively on the Architecture Suite, which supports finer-grained analysis.

4.4 How do different agent architectures affect task performance and stability?

More general-purpose solver architectures, designed to handle a wide range of complex tasks, tend to progress more cautiously. To ensure robustness, they often pause at each iteration to reflect on intermediate states. For example, Plan-and-Execute first decomposes the overall goal into a sequence of milestones and then solves each subtask incrementally. This approach helps the model maintain a comprehensive understanding of task context and often yields more reliable outcomes, but at the cost of increased execution time and resource consumption.

In contrast, when the task type is known in advance, a more specialized architecture can be employed. CRAG, for instance, is explicitly designed for retrieval-based workloads. Rather than exploring alternative reasoning paths, it prioritizes directly answering the query with minimal detours. This objective-driven design attempts to solve the task as early as possible, even with incomplete background information, trading exploration for efficiency. Such differences in design philosophy lead to substantial divergence in execution behavior across architectures.

Specialized solver minimizes resource consumption. As shown in Figure 9a, CRAG consistently occupies the lower-cost and lower-latency region across different model choices. In particular, CRAG achieves a median cost of $0.0010 per task, which is more than

an order of magnitude lower than both Plan-and-Execute (median $0.0126) and LATS (median $0.0101). CRAG also executes faster, with a median task duration of 42.8 s, compared to 101.5 s for Plan-and-Execute.

In contrast, Plan-and-Execute exhibits substantially higher variance in task duration (interquartile range 30.6–356.6 s), reflecting the overhead introduced by iterative planning and execution. LATS achieves relatively low median latency (32.3 s), but incurs higher resource cost overall.

Accuracy degrades with increasing architectural complexity. Furthermore, Figure 9b shows that CRAG attains accuracy comparable to, and in some cases exceeding, more general architectures. CRAG achieves an average accuracy of 70.6%, compared to 48.3% for Plan-and-Execute, while also exhibiting lower variability across runs. These results indicate that task-specialized agent architectures can simultaneously reduce resource consumption and maintain strong task performance.

Notably, increased architectural complexity does not necessarily translate into higher accuracy and may even be detrimental. While it is tempting to introduce additional agents – such as fact-checkers or verification stages – to enforce desired behavior, such designs inevitably increase execution cost and prolong interaction histories. In our evaluation, Plan-and-Execute spends substantially more time reasoning over tasks yet achieves lower accuracy (average 48.3% vs. 70.6% for CRAG), despite incurring significantly higher execution cost. This behavior aligns with prior findings that model performance degrades as interaction histories grow longer, due to diminishing attention to earlier context and error accumulation in extended reasoning chains [34].

> *Finding 5:* More general agent architectures consume more resources and do not consistently improve accuracy.

4.5 How does model choice affect MAS behavior?

A natural assumption in LLM-based MAS design is that upgrading the underlying model should improve system performance. Intuitively, scaling to more capable models is expected to increase cost while yielding higher accuracy. However, our experimental results challenge this assumption. We find that stronger models do not necessarily incur substantially higher costs in practice, nor do they consistently lead to improved correctness. Instead, model choice affects MAS behavior in more nuanced and sometimes counterintuitive ways.

Stronger models reduce iteration overhead rather than total cost. More capable models often complete subtasks with fewer iterations, reducing pathological behaviors such as repeated retries or prolonged refinement loops. However, these efficiency gains primarily offset higher per-token pricing rather than translating into lower overall cost. For example, `gpt-5-mini` and `gpt-5-nano` exhibit comparable mean cost per task (0.033 vs. 0.043), despite differences in model size, while `gpt-4o-mini` achieves substantially lower median cost (0.0034) than both. Similarly, execution latency is non-monotonic: `gpt-4o-mini` completes tasks faster (median 45.3 s) than the larger 5-series models, whereas `gpt-5-nano`
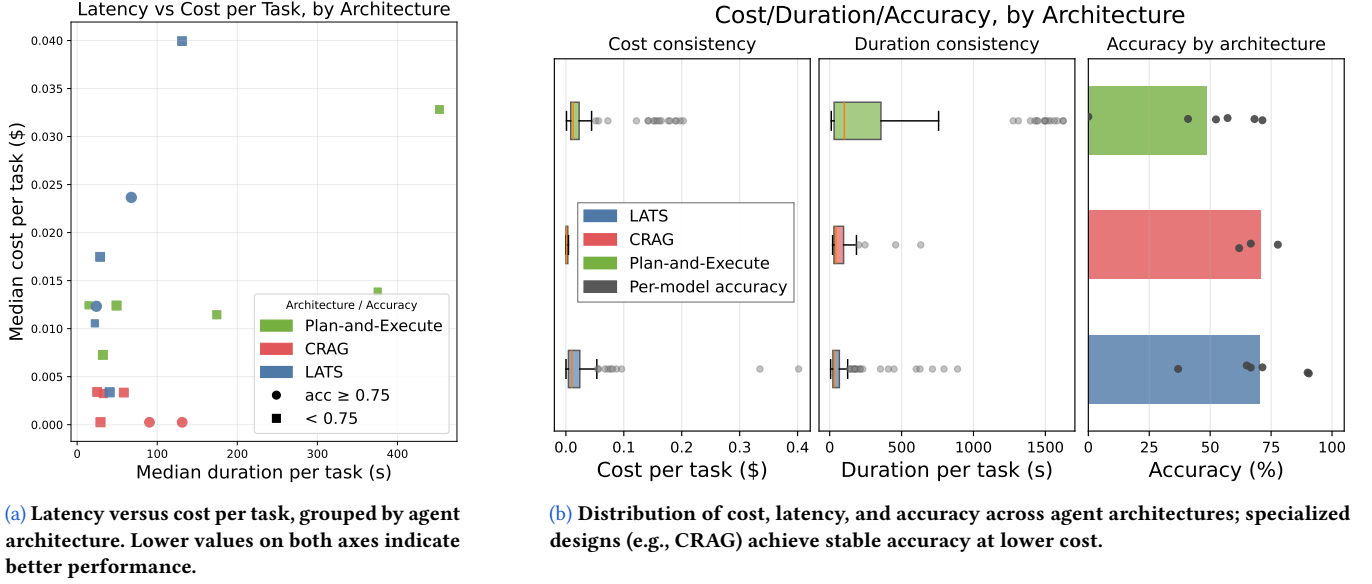
(a) **Latency versus cost per task, grouped by agent architecture. Lower values on both axes indicate better performance.**

(b) **Distribution of cost, latency, and accuracy across agent architectures; specialized designs (e.g., CRAG) achieve stable accuracy at lower cost.**

Figure 9: **Resource cost versus accuracy across agent architectures. Task-specific designs such as CRAG achieve comparable accuracy with lower resource cost than more general architectures like LATS.**

is slower than `gpt-5-mini` despite being nominally smaller. These results indicate that model choice influences iteration efficiency and tail behavior, but does not induce a clear cost hierarchy.

Accuracy exhibits non-monotonic and unstable trends across models. We further observe no consistent relationship between model strength and task accuracy. While `gpt-5-mini` achieves the highest accuracy (median 81%), weaker or similarly priced models do not follow a predictable trend: `gpt-5-nano` trails at 65%, and `gpt-4o-mini` exhibits high median accuracy (71%) but a substantially lower mean (48%), indicating unstable behavior with heavy failure cases. Gemini models cluster around similar accuracy levels (approximately 66%), with the 2.0-lite variant performing worse overall. These results suggest that MAS accuracy is highly sensitive to execution dynamics and variance amplification, rather than model capacity alone, and that upgrading the base model is insufficient to guarantee improved correctness.

> *Finding 6:* Upgrading the base LLM does not reliably reduce cost or improve accuracy in MAS, as execution dynamics dominate model-level gains.

4.6 What are the dominant failure modes in LLM-based multi-agent systems?

We find that most failures manifest as silent gray errors (75.17% in Table 3), which do not trigger explicit system failures and are therefore not immediately visible to users. These errors only become apparent upon manual inspection of the output. Importantly, such failures are not system-level exceptions, but rather plausible-looking yet unusable responses. As a result, failure attribution in LLM-based MAS is particularly challenging, since erroneous executions often complete without emitting any hard error signals.
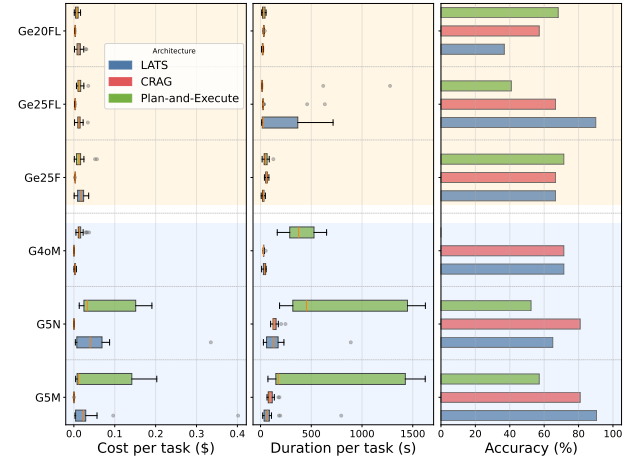

Figure 10: **Cost–duration–accuracy trade-offs across LLMs; efficiency improves for Gemini-family models, while accuracy shows no clear scaling trend.**

We further break down failure causes by model in Figure 11a, which reveals distinct, model-specific failure signatures. Rather than failing uniformly, different LLM backends exhibit characteristic behaviors when errors occur.

Model-specific failure patterns.

- **Gemini-2.0-flash-lite** predominantly fails by producing underspecified or incomplete outputs, where a response is returned but lacks sufficient detail to satisfy task requirements.

Table 3: **Global failure composition across all experiments.**

Failure category	Percentage (%)
Missing / underspecified output	47.61
Wrong fact / entity	27.66
Empty prediction	15.96
Exception	6.38
Timeout	1.86
Other	0.54
Silent semantic failures	**75.17**
Explicit failures	24.84

- **Gemini-2.5-flash-lite** exhibits a more conservative failure mode, frequently abstaining and returning empty or null outputs when uncertain.
- **GPT-4o-mini** tends to produce fully formed but factually incorrect responses, committing confidently to wrong entities or facts rather than omitting answers.

These distinct failure signatures indicate that MAS failures are not only model-dependent, but also shaped by how agent architectures interpret and propagate partial outputs. Consequently, failures emerge as execution-path–dependent phenomena rather than isolated faults attributable to a single component.

> *Finding 7:* MAS failures predominantly manifest as silent semantic errors, with distinct, model-specific failure signatures that are amplified by execution dynamics.

Divergent failure attribution across LLM-as-judges. To assess the reliability of LLM-as-judge–based failure attribution, we perform offline analysis using three additional judge models, each provided with the final MAS response and the corresponding gold answer. As shown in Figure 11b, offline attribution struggles to correctly identify system-level failures, such as exceptions or timeouts, due to the absence of runtime execution signals.

For example, a MAS execution may enter a non-terminating review loop that repeatedly generates responses containing the correct answer but never produces a valid final output. During online execution, such behavior is correctly identified as a failure, since the task does not terminate successfully. In contrast, an offline judge, which only observes the final response and history, may incorrectly classify the execution as successful because the correct answer appears in the trace.

Even for semantic-based gray failures, where judges often agree on whether an execution is broadly correct or incorrect (e.g., all judges consistently identify CRAG executions as successful), substantial divergence arises in the attribution of failure *types*. For instance, when a MAS responds with: *"I am sorry, I cannot answer this question. The available tools do not have the functionality to determine the country of a member of the Gujarat Legislative Assembly and parliament."* the gpt-oss-120b judge classifies this outcome as an *empty prediction*, whereas gemini-2.5-flash attributes it to a *wrong fact/entity*.

These discrepancies highlight that, even under identical inputs and failure definitions, LLM-based judges may disagree on fine-grained failure attribution, underscoring the inherent subjectivity and instability of offline, semantics-only failure analysis.

4.7 How does tool usage impact cost and accuracy?

A common assumption in LLM-based MAS design is that enabling external tools should improve task performance. By equipping agents with additional information sources or capabilities, one would expect higher-quality outputs and, consequently, improved accuracy. However, our results indicate that the impact of tool usage is highly dependent on the underlying agent architecture.

Enabling web search commonly increases resource consumption. Overall, enabling external tools tends to increase resource consumption, but accuracy gains are not uniform across architectures. As shown in Figure 12a, tool usage introduces different overheads depending on how tools are integrated into the execution workflow. For CRAG, external tools primarily increase monetary cost, with a median cost increase of $0.0010 per task and a modest median latency increase of 8.1 s, reflecting additional retrieval and processing steps. In contrast, Plan-and-Execute experiences a substantial increase in task duration, with a median latency increase of 34.1 s, while its monetary cost slightly decreases, indicating that overhead is shifted toward longer execution rather than additional token usage. LATS exhibits the highest overall overhead, with tool usage increasing both execution time and cost, suggesting compounded interaction and coordination overheads.

When web search reduces task duration. While external tools typically introduce additional overhead, we observe notable outliers where web search reduces overall execution cost and latency. In particular, for CRAG with gpt-5-nano, enabling web search results in faster task completion (by approximately 2 s on average). This effect arises because, in the absence of external evidence, the model tends to generate longer, more speculative responses, increasing both token usage and per-round LLM latency. Trace-level analysis confirms this behavior: in no-search executions, the generator and grader produce longer outputs, substantially increasing per-call latency, whereas providing web evidence shortens responses and reduces LLM latency (median generator time 11.2 s to 6.1 s). As a result, CRAG with web search achieves 13.9% lower mean task duration despite the additional retrieval step, indicating that external context can reduce speculative reasoning and offset tool overhead.

When web search reduces planning cost. For Plan-and-Execute, enabling web search often leads to a net reduction in cost, as external evidence allows the planner to generate more concrete and concise plans. Without web search, the planner tends to produce longer, speculative plans, and the replanner emits more verbose messages to justify or revise these plans, inflating token usage.

Trace-level evidence supports this observation: across models, planner messages are substantially shorter when web search is enabled (e.g., average planner tokens drop from over 1,500 to a few hundred per call), and replanner turns are also more concise. Although the number of planning or replanning iterations may

(a) Online failure attribution via LLM-as-judge (GPT-4o-mini).

(b) Offline failure attribution via LLM-as-judge (Gemini-2.5-Flash, GPT-4o, GPT-OSS-120B).
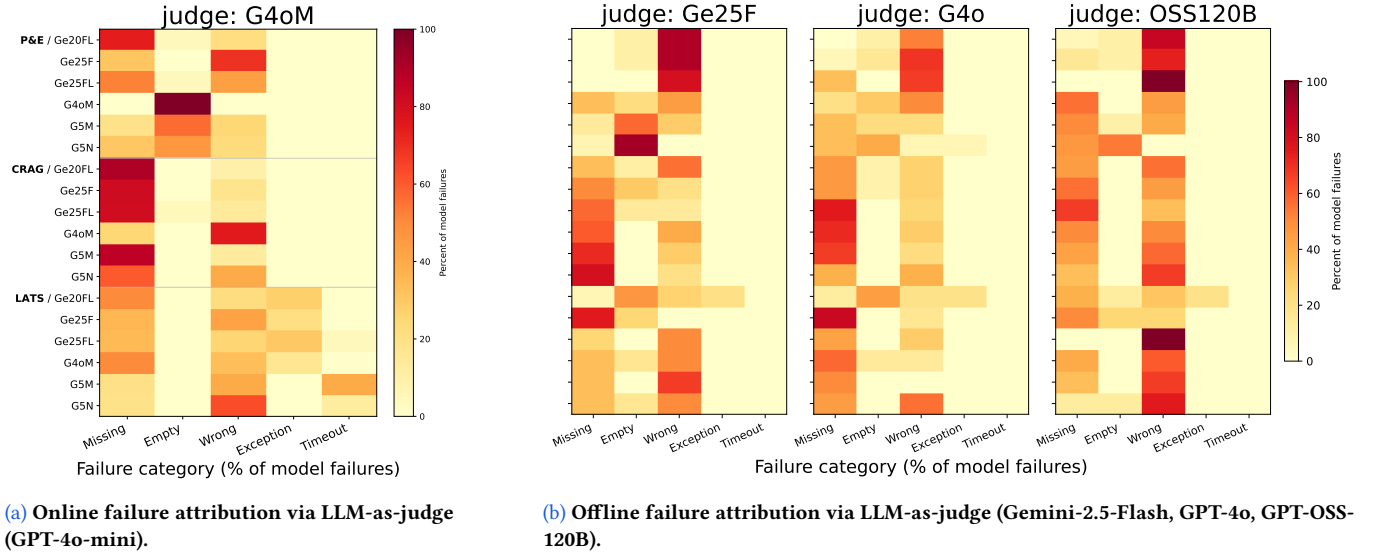
Figure 11: Failure attribution using different LLM judges. We prioritize online attribution when possible, as it incorporates runtime signals unavailable offline, such as execution stalls and incomplete system outputs. For offline attribution, judges are provided with the final MAS response, the corresponding gold answer, and an identical failure taxonomy. Despite controlling inputs, attribution results exhibit substantial variance across judge models, highlighting the inherent subjectivity and instability of LLM-based failure attribution.

remain similar – or even increase slightly – the reduction in per-turn token usage outweighs the cost of the additional web retrieval step, resulting in lower overall execution cost.

> *Finding 8:* By providing external context, tools can reduce speculative generation, lowering inference time and cost.

Web search boosts accuracy, but non-uniformly across architectures. Tool usage yields markedly different outcomes across agent architectures. As shown in Figure 12b, CRAG consistently benefits from external tools, achieving a median accuracy improvement of 35.7% and improving accuracy in 83.3% of evaluated runs. In contrast, Plan-and-Execute loss minor median accuracy, with improvements observed in only one third of runs. LATS shows marginal and unstable gains, with a median accuracy improvement of 4.2% and positive effects in only half of the cases.

In conclusion, these accuracy trends align with the associated cost and latency overheads. CRAG incurs only modest increases in cost and execution time, whereas Plan-and-Execute primarily shifts overhead to longer execution latency, and LATS experiences increases in both cost and latency. Together, these results indicate that external tools improve MAS performance only when the underlying architecture can incorporate them without amplifying execution complexity or instability.

> *Finding 9:* External tools improve accuracy only when the agent architecture can integrate them without amplifying execution overhead or variance.
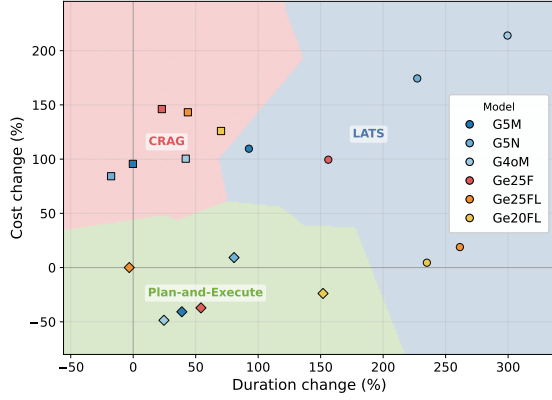
5 Discussion

While MAESTRO already provides valuable insights into the behavior of LLM-based MAS, significant opportunities for extension remain.
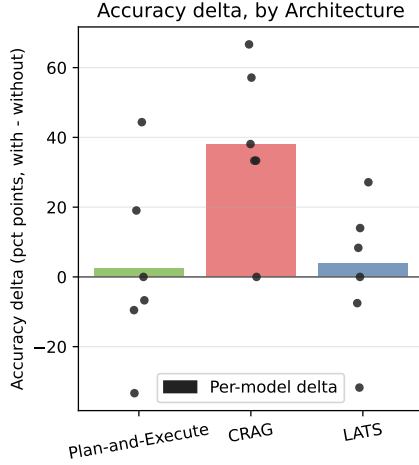
5.1 Limitation

Generalizability. Given the inherent heterogeneity (**D1**) of LLM-based MAS, it is difficult to identify a single canonical architecture or execution pattern that generalizes across all agentic systems. The design space of MAS continues to evolve rapidly, with new coordination strategies, tooling abstractions, and execution models emerging at a fast pace.

While MAESTRO is designed to cover a diverse set of widely used MAS architectures and workflows, the insights derived from our evaluation are necessarily grounded in the specific instances and configurations studied. As a result, some findings may not directly transfer to future MAS designs or to application domains not represented in our benchmark. In particular, advances in agent orchestration or model capabilities may invalidate certain observations over time, highlighting the need for continuously evolving benchmarks alongside the MAS ecosystem.

Overhead of telemetry. MAESTRO incorporates fine-grained telemetry to examine per-step behavior in LLM-based MAS and derive detailed insights into execution dynamics. However, such instrumentation introduces profiling overhead that may degrade system performance. Before real-world deployment, MAESTRO must therefore optimize telemetry collection to minimize overhead, for example, through sampling strategies, adaptive logging, or lightweight monitoring mechanisms.

(a) **Impact of enabling the Web Search tool. We measure the deltas in execution latency and monetary costs, after enabling the web search tool.**

(b) **Accuracy deltas after enabling web search.**

Figure 12: **Resource cost versus accuracy across agent architectures. Task-specific designs such as CRAG achieve comparable accuracy with lower resource cost than more general architectures like LATS.**

5.2 Future works

Automated integration for MAS instances. Currently, MAESTRO includes a limited representative set of MAS examples, which may not fully capture the diversity and complexity of real-world deployments. To improve representativeness, the benchmark must incorporate a broader range of MAS examples that reflect the diversity of real-world deployments and use cases. At present, fine-grained telemetry is enabled through ad-hoc instrumentation tailored to individual MAS frameworks. As future work, we plan to develop an automated translation layer that maps heterogeneous agent implementations into a uniform execution representation, enabling systematic behavior capture with minimal manual intervention. Such automated integration would also lower the barrier for external contributions, allowing developers to more easily evaluate their own MAS implementations using our test suite.

Monolith vs. distributed. Similar to the relationship between monolithic applications and microservices in traditional software architectures, LLM-based MAS can be deployed either as a single, unified system or as a collection of distributed agents communicating over a network. Distributed deployments could bring benefits such as improved fault tolerance, scalability, and modularity. However, they also introduce challenges related to network latency, synchronization, and consistency. Future work could explore the trade-offs between monolithic and distributed MAS architectures, evaluating their performance, reliability, and resource utilization under various workloads and deployment scenarios. Also, it would be interesting to investigate the impact of the underlying network infrastructure on the behavior and performance of distributed LLM-based MAS.

MAS-specific failure attribution. The inherent non-determinism of LLMs introduces failure modes that are rarely encountered in traditional deterministic systems. When multiple agents are composed into a pipeline, these effects are further amplified, increasing the likelihood of inconsistent or emergent failure behaviors. Such phenomena are already observed in our evaluation. For instance, in the Plan-and-Execute architecture, we identify recurring execution patterns in which the executor successfully retrieves and returns the gold answer, yet the replanner repeatedly rejects the intermediate result. This mismatch prevents the system from reaching a terminal state, ultimately leading to timeouts despite the presence of a correct solution in the execution trace. These observations highlight the difficulty of failure attribution in LLM-based MAS. Due to the extensive fault space induced by LLM heterogeneity (**D1**), which grows combinatorially as multiple agents and models interact, failures often cannot be localized to a single component or decision point. Developing principled failure taxonomies and robust attribution mechanisms for such systems therefore remains an important direction for future work.

Communication mechanisms. Our experiments reveal substantial variation in how different MAS frameworks implement inter-agent communication. In many of the frameworks we evaluate, agents primarily interact through structured function calls. Others rely on a shared global scratchpad that allows agents to read from and write to a common intermediate state.

For interactions beyond a single host or for accessing external data sources, some frameworks additionally support standardized communication protocols, such as agent-to-agent (A2A) [12] messaging or the Model Context Protocol (MCP) [13]. These differences in communication mechanisms introduce distinct execution semantics and coordination patterns, yet their impact on system performance, robustness, and failure behavior remains largely unexplored. This observation highlights an open research area in understanding how communication design choices influence the behavior of LLM-based MAS.

Parallelism and coordination effects. Parallelism fundamentally alters the execution dynamics of LLM-based MAS, affecting not only throughput and resource utilization but also coordination behavior and failure modes. While parallel execution and load balancing are well-established techniques in traditional systems, their impact in asynchronous MAS remains poorly understood.

Existing work has extensively studied single-LLM optimizations, such as speculative inference and parallel decoding [6, 30], to improve accuracy or cost–performance trade-offs. However, it is unclear how these techniques translate to multi-agent settings, where multiple agents may operate concurrently and interact through shared state or tools.

In particular, the effects of parallel agents with overlapping or partially redundant roles are not yet well characterized. Such configurations may introduce new coordination overheads, contention, or emergent behaviors that differ fundamentally from single-model parallelism, highlighting an important direction for future investigation.

Framework overhead investigation. In our evaluation, we observe that kagent [46], a framework designed to facilitate building distributed LLM-based multi-agent systems, can incur non-trivial communication overhead and may also trigger operational failures (e.g., a disk-full error on the kagent controller node). Future work should systematically characterize the overheads introduced by MAS frameworks and quantify their impact on end-to-end performance and reliability.

6 Conclusion

We argue that LLM-based multi-agent systems (MAS) must be evaluated not merely by task completion, but as complex systems characterized by dynamic, stochastic execution. To this end, we introduce MAESTRO, an open-source evaluation suite that standardizes the configuration and execution of heterogeneous MAS while exporting fine-grained, system-level telemetry to enable cross-stack comparison.

Our evaluation of 12 representative MAS instances reveals that while agentic workflows are often *structurally* stable, they exhibit significant *temporal* instability, driving high run-to-run variance in latency, cost, and failure modes. Crucially, we find that MAS architecture dominates backend model and toolset choices in determining resource profiles, reproducibility, and the cost–latency–accuracy trade-off. These findings indicate that optimizing reliability and efficiency in agentic systems is fundamentally an architectural challenge, necessitating benchmarks that prioritize deep execution visibility over simple application-level scores.

Looking ahead, we plan to extend MAESTRO to support distributed architectures and automated agent integration, while refining failure attribution to better diagnose stochastic errors. Our ultimate goal is to establish standardized observability contracts, ensuring that benchmarking keeps pace with the evolving complexity of agentic systems.

References

[1] AutoGen. 2025. *AutoGenBench*. https://github.com/microsoft/autogen/tree/main/python/packages/agbench Accessed: 2025-12-28.

[2] Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, et al. 2025. Nestful: A benchmark for evaluating llms on nested sequences of api calls. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 33526–33535.

[3] BIG bench authors. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research (TMLR)* (2023). https://openreview.net/forum?id=uyTL5Bvosj

[4] Tara Bogavelli, Roshnee Sharma, and Hari Subramani. 2025. AgentArch: a comprehensive benchmark to evaluate agent architectures in enterprise. *arXiv preprint arXiv:2509.10769* (2025).

[5] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. 2025. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657* (2025).

[6] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318* (2023).

[7] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. 2024. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1362–1379.

[8] Comet ML. 2024. *Opik: open-source LLM evaluation and observability platform*. https://github.com/comet-ml/opik Accessed: 2025-12-14.

[9] Alessandro Cornacchia, Vaastav Anand, Muhammad Bilal, Zafar Qazi, and Marco Canini. 2025. DMAS-Forge: A Framework for transparent deployment of AI applications as distributed systems. *Workshop on Systems for Agentic AI (SAA)* (2025).

[10] Microsoft Corporation. 2024. *AutoGen*. https://microsoft.github.io/autogen/stable//index.html Accessed: 2025-12-28.

[11] Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. 2025. TRAIL: trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638* (2025).

[12] A2A Developers. 2025. *A2A protocol specification*. https://a2a-protocol.org/latest/topics/what-is-a2a/ Accessed: 2025-12-09.

[13] MCP Developers. 2025. *Model Context Protocol specification*. https://modelcontextprotocol.io/docs/getting-started/intro Accessed: 2025-12-09.

[14] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang (Eric) Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. 2024. *Magentic-One: A generalist multi-agent system for solving complex tasks*. Technical Report MSR-TR-2024-47. Microsoft. https://www.microsoft.com/en-us/research/publication/magentic-one-a-generalist-multi-agent-system-for-solving-complex-tasks/

[15] Longling Geng and Edward Y Chang. 2025. Realm-bench: a real-world planning benchmark for LLMs and multi-agent systems. *arXiv preprint arXiv:2502.18836* (2025).

[16] Google. 2025. *Google ADK samples repository*. https://github.com/google/adk-samples/tree/main Accessed: 2025-12-28.

[17] Google. 2025. *Google agent development kit (ADK)*. https://google.github.io/adk-docs/ Accessed: 2025-12-28.

[18] Google. 2025. *Google GenAI search tool*. https://googleapis.github.io/python-genai/genai.html#genai.types.GoogleSearch Accessed: 2025-12-28.

[19] Google. 2025. *Google Imagen 3 model*. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/imagen/3-0-generate Accessed: 2025-12-28.

[20] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. In *International Joint Conferences on Artificial Intelligence Organization (IJCAI)*, Kate Larson (Ed.). 8048–8057. doi:10.24963/ijcai.2024/890 Survey Track.

[21] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: a survey of progress and challenges. *arXiv preprint arXiv:2402.01680* (2024).

[22] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714* (2024).

[23] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[24] CrewAI Inc. 2024. *CrewAI*. https://www.crewai.com Accessed: 2025-12-28.

[25] LangChain Inc. 2024. *LangGraph.* https://langchain-ai.github.io/langgraph/ Accessed: 2025-12-28.

[26] Gurusha Juneja, Jayanth Naga Sai Pasupulati, Alon Albalak, Wenyue Hua, and William Yang Wang. 2025. MAGPIE: a benchmark for multi-agent contextual privacy evaluation. *arXiv preprint arXiv:2510.15186* (2025).

[27] Fan Lai, Yinwei Dai, Sanjay Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha Madhyastha, and Mosharaf Chowdhury. 2022. Fedscale: Benchmarking model and system performance of federated learning at scale. In *International Conference on Machine Learning (ICML).* PMLR, 11814–11827.

[28] LangChain Documentation. 2025. *ChatVertexAI — LangChain Vertex AI reference.* https://reference.langchain.com/python/integrations/langchain_google_vertexai/ChatVertexAI/ Accessed: 2025-12-28.

[29] LastMile AI. 2024. *MCP Financial Analyzer: A Multi-Agent MCP Application Example.* Accessed: 2025-12-31.

[30] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML).* PMLR, 19274–19286.

[31] Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. AlpacaEval: An automatic evaluator of instruction-following models. https://github.com/tatsu-lab/alpaca_eval.

[32] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Alexander Cosgrove, Christopher D Manning, Christopher Re, Diana Acosta-Navas, Drew Arad Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue WANG, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Andrew Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. 2023. Holistic evaluation of language models. *Transactions on Machine Learning Research (TMLR)* (2023). https://openreview.net/forum?id=iO4LZibEqW Featured Certification, Expert Certification.

[33] Wang Lilian. 2023. *LLM-powered autonomous agents.* https://lilianweng.github.io/posts/2023-06-23-agent/ Accessed: 2025-12-28.

[34] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics (TACL)* 12 (2024), 157–173.

[35] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. 2023. AgentBench: evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688* (2023).

[36] LlamaIndex. 2024. *LlamaIndex.* https://www.llamaindex.ai Accessed: 2025-12-28.

[37] Agno maintainers. 2025. *Agno teams.* https://docs.agno.com/concepts/teams/introduction Accessed: 2025-12-28.

[38] Microsoft. 2025. *AutoGen AgentChat user guide.* https://microsoft.github.io/autogen/stable//user-guide/agentchat-user-guide/ Accessed: 2025-12-28.

[39] Dany Moshkovich, Hadar Mulian, Sergey Zeltyn, Natti Eder, Inna Skarbovsky, and Roy Abitbol. 2025. Beyond black-box benchmarking: observability, analytics, and optimization of agentic systems. *arXiv preprint arXiv:2503.06745* (2025).

[40] OpenTelemetry Authors. 2025. *OpenTelemetry.* https://opentelemetry.io/ Accessed: 2025-12-28.

[41] Melissa Z Pan, Negar Arabzadeh, Riccardo Cogo, Yuxuan Zhu, Alexander Xiong, Lakshya A Agrawal, Huanzhi Mao, Emma Shen, Sid Pallerla, Liana Patel, et al. 2025. Measuring agents in production. *arXiv preprint arXiv:2512.04123* (2025).

[42] A2A project. 2025. *A2A samples repository.* https://github.com/a2aproject/a2a-samples Accessed: 2025-12-28.

[43] Psutil Developers. 2025. *Psutil: process and system utilities.* https://github.com/giampaolo/psutil Accessed: 2025-12-28.

[44] Bronson Schoen, Evgenia Nitishinskaya, Mikita Balesni, Axel Højmark, Felix Hofstätter, Jérémy Scheurer, Alexander Meinke, Jason Wolfe, Teun van der Weij, Alex Lloyd, et al. 2025. Stress testing deliberative alignment for anti-scheming training. *arXiv preprint arXiv:2509.15541* (2025).

[45] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2023), 8634–8652.

[46] Solo.io. 2025. *kagent: cloud native agentic AI framework.* https://kagent.dev/ Accessed: 2025-12-28.

[47] Haochen Sun, Shuwen Zhang, Lujie Niu, Lei Ren, Hao Xu, Hao Fu, Fangkun Zhao, Caixia Yuan, and Xiaojie Wang. 2025. Collab-Overcooked: benchmarking and evaluating large language models as collaborative agents. *arXiv preprint arXiv:2502.20073* (2025).

[48] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (2023).

[49] Wei Wang, Dan Zhang, Tao Feng, Boyan Wang, and Jie Tang. 2024. Battleagentbench: a benchmark for evaluating cooperation and competition capabilities of language models in multi-agent systems. *arXiv preprint arXiv:2408.15971* (2024).

[50] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, et al. 2025. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *ACM Conference on Knowledge Discovery and Data Mining (SIGKDD).* 5831–5841.

[51] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *Conference on Language Modeling (COLM).*

[52] Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323* (2023).

[53] Bingyu Yan, Zhibo Zhou, Litian Zhang, Lian Zhang, Ziyi Zhou, Dezhuang Miao, Zhoujun Li, Chaozhuo Li, and Xiaoming Zhang. 2025. Beyond self-talk: a communication-centric survey of LLM-based multi-agent systems. *arXiv preprint arXiv:2502.14321* (2025).

[54] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. 2024. Corrective Retrieval Augmented Generation. (2024).

[55] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP).*

[56] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. Tau-Bench: a benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045* (2024).

[57] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2023), 11809–11822.

[58] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR).*

[59] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).

A Appendix

A.1 Details of post-processing component

We build an observation component that characterizes MAS be-havior across benchmark tasks and configurations. By default, we generate a common set of plots for each workload. We report results both for individual executions and aggregated across multiple runs. Because the full set of figures is large, we plan to release them as a dataset and provide only a brief summary here.

- **Token consumption**. For the single run, we plot the to-ken consumption of each agent over time, including prompt tokens and completion tokens; we also plot the total token consumption of all agents over time. For multiple runs, we plot the average total token consumption of all agents over time.
- **Delay**. For the single run, we plot the end-to-end delay of the whole system, and decompose it into agent processing delay, agent-to-LLM communication delay, and agent-to-agent communication delay. For multiple runs, we plot the average end-to-end delay of the whole system. We plot the breakdown of the average delay into different components. We also plot a flame graph for delay.
- **CPU and memory usage**. For the single run, we plot the time series of CPU and memory usage of the whole system. We also give a correlation analysis between CPU/memory usage and system events (e.g., agent invocations, LLM calls, etc.). For multiple runs, we plot the mean, peak, and mini-mum CPU and memory usage of the whole system.
- **Message size**. For the single run, we plot the average input and output message size of each agent for both agent-to-agent messages and agent-to-LLM messages. For multiple runs, we plot the total input and output message size of agent-to-agent messages and agent-to-LLM messages. We also plot per-agent, per-agent-pair, and per-agent-to-LLM message sizes.
- **Call graph**. We visualize the call graph of the agents in the system. We also show the similarity between the call graphs of different runs using graph similarity metrics. We use two similarity metrics: Jaccard similarity and Largest Common Subgraph (LCS) similarity. Jaccard similarity measures the similarity between two sets of edges in the call graphs, while LCS similarity measures the sequence similarity of the edges in the call graphs.

A.2 Collector implementation.

A.2.1 Telemetry field collection. The following listing (Listing 1) shows a collection of telemetry fields in MAESTRO.

Listing 1: Telemetry field collection

```json
[
  {
    "trace_id": "<32-hex-trace-id>",
    "span_id": "<16-hex-span-id>",
    "parent_span_id": "<16-hex-parent-span-id-or-null>",
    "name": "<operation-name>",
    "agent_name": "<agent-name>",
    "start_time": 0,
    "end_time": 0,
    "duration_ns": 0,
    "kind": "<INTERNAL|SERVER|CLIENT|PRODUCER|CONSUMER>",
    "status": {
      "status_code": "<UNSET|OK|ERROR>",
      "description": "<optional-description>"
    },
    "attributes": {
      "gen_ai.operation.name": "<call_llm|execute_tool|invoke_agent>",
      "gen_ai.system": "<provider>",
      "gen_ai.agent.name": "<agent-name>",
      "gen_ai.agent.description": "<optional-description>",
      "gen_ai.request.model": "<model>",
      "gen_ai.conversation.id": "<conversation-id>",
      "gen_ai.tool.name": "<tool-name>",
      "gen_ai.tool.type": "<FunctionTool|Builtin>",
      "gen_ai.tool.call.id": "<tool-call-id>",
      "gen_ai.tool.description": "<tool-description>",
      "gen_ai.usage.input_tokens": 0,
      "gen_ai.usage.output_tokens": 0,
      "gen_ai.usage.total_tokens": 0,
      "gen_ai.llm.call.count": 0,
      "gen_ai.mcp.call.count": 0,
      "gen_ai.response.finish_reasons": [],
      "mcp.server": "<server-name>",
      "mcp.tool": "<tool-name>",
      "gcp.vertex.agent.llm_request": "<raw-request-json>",
      "gcp.vertex.agent.llm_response": "<raw-response-json>",
      "gcp.vertex.agent.tool_call_args": "<tool-call-args>",
      "gcp.vertex.agent.tool_response": "<tool-response>",
      "gcp.vertex.agent.invocation_id": "<invocation-id>",
      "gcp.vertex.agent.session_id": "<session-id>",
      "gcp.vertex.agent.event_id": "<event-id>",
      "agent.log": "<optional-log-line>",
      "agent.retry.attempt_number": 0,
      "agent.retry.trigger": "<quality|relevance_guard|guard_fail|timeout|system|upstream>",
      "agent.retry.previous_span_id": "<16-hex-span-id-or-null>",
      "agent.retry.reason": "<optional-retry-trigger>",
      "run.outcome": "<success|failure>",
      "run.outcome_reason": "<optional-reason>",
      "run.judgement": "<correct|wrong|unknown>",
      "run.judgement_reason": "<optional-reason>",
      "agent.failure.category": "<guard|quality|system|timeout|upstream>",
      "agent.failure.reason": "<free-text>",
```

```
53        "agent.output.useless": false,
54        "agent.output.useless_reason": "<free-text
              >",
55        "communication.input_message_size_bytes": 0,
56        "communication.output_message_size_bytes":
              0,
57        "communication.total_message_size_bytes": 0
58      },
59      "communication": {
60        "is_in_process_call": false,
61        "input_message_size_bytes": 0,
62        "output_message_size_bytes": 0,
63        "total_message_size_bytes": 0
64      },
65      "resource": {
66        "attributes": {
67          "service.name": "<service-name>",
68          "service.version": "<semver>",
69          "deployment.environment": "<local|dev|
                staging|prod>",
70          "telemetry.sdk.name": "<sdk-name>",
71          "telemetry.sdk.language": "<language>",
72          "telemetry.sdk.version": "<version>",
73          "host.name": "<optional-host>"
74        }
75      }
76    }
77 ]
```

A.2.2 Lacking of standardized observability contracts.

A.2.2 Lacking of standardized observability contracts. Even when a common observability schema (e.g., OTEL) is imposed, orchestration stacks differ substantially in which telemetry signals are surfaced, transformed, or suppressed. While some frameworks propagate execution metadata such as token usage, termination reasons, or payload sizes to application-level hooks, others consume these signals within internal execution layers without exposing them externally. As a result, identical agent workflows may exhibit markedly different observability characteristics depending on the combination of model backend, transport mechanism, and orchestration framework.

A key source of this discrepancy is that, unlike generated text, token usage is not treated as a first-class execution artifact with a well-defined exposure contract, but rather as auxiliary metadata. Consequently, whether token usage is observable depends jointly on (i) the underlying model API and its response schema, (ii) the transport layer through which inference results are delivered (e.g., streaming versus non-streaming), and (iii) the framework's instrumentation and log-propagation strategy. In the absence of an agreed-upon contract, each layer independently decides how token usage is represented and whether it is forwarded, making end-to-end observability fragile and stack-dependent.

Backend- and modality-dependent loss of usage metadata. This lack of standardization manifests across both model backends and invocation modalities. For example, Gemini and Vertex AI do expose token usage information, but under response layouts and terminology that differ from OpenAI- or Anthropic-style APIs. Token usage may be reported through backend-specific metadata fields (e.g., reporting generation-side token usage as *candidate tokens*, rather than OpenAI-style output or completion tokens [28]), requiring backend-aware parsing logic to recover usage information. Beyond generation APIs, we further observe that usage metadata may be dropped entirely at the framework level for non-generative calls. In LangGraph, embedding model invocations (e.g., OpenAIEmbeddings and VertexAIEmbeddings) do not propagate token usage information, even when the underlying provider APIs support usage accounting. In such cases, the framework consumes partial response metadata internally without forwarding it to application-level telemetry or accounting hooks. As a consequence, orchestration frameworks such as LangGraph and MCP-Agent – many of which implicitly assume a synchronous, OpenAI-style usage schema – may fail to capture token usage across a range of execution paths unless explicit, backend- and modality-aware instrumentation is implemented. Importantly, these limitations do not arise from agent logic or missing backend signals, but from the absence of a stable, cross-provider observability contract that defines how usage metadata should be structured, preserved, and forwarded across abstraction boundaries.

Implications for MAS benchmarking. This inconsistency introduces blind spots in cost and efficiency analysis, particularly in heterogeneous multi-agent settings where different LLM backends coexist. It further demonstrates that observability properties cannot be assumed to be model-agnostic, motivating the need for standardized observability contracts that explicitly define which execution signals must be exposed by LLM APIs and agent frameworks.