Distributed Multi-LLM-Agent Architectures: Orchestration Patterns, Communication Protocols, and Fault Tolerance

Jiaqi Xu[1], Minxian Xu[2], and Yang Wang[3]

[1] University of Chinese Academy of Sciences, China
jq.xu2@siat.ac.cn
[2] Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
mx.xu@siat.ac.cn
[3] Shenzhen University of Advanced Technology, China
wangyang@suat-sz.edu.cn

Abstract. Large language model (LLM) agents are rapidly evolving from single-agent tool-use systems into collaborative multi-agent architectures. While existing studies have extensively discussed agent capability, role specialization, and task decomposition, relatively less attention has been paid to the distributed systems mechanisms that make multi-agent collaboration reliable, scalable, and controllable in practice. In particular, orchestration topology, communication semantics, state synchronization, persistence, and fault recovery have emerged as key system-level design dimensions.

This paper examines distributed multi-LLM-agent architectures from the perspective of orchestration patterns, communication protocols, and fault tolerance. It focuses on representative frameworks including AutoGen, LangGraph, CrewAI, and OpenAI Swarm, and compares them in terms of centralized, hierarchical, and handoff-driven coordination, message routing, state management, recovery support, and engineering trade-offs. In addition, it incorporates benchmark-oriented evidence, public serving reports, and protocol developments to examine how architectural choices relate to deployment concerns such as reliability, persistence, observability, interoperability, and long-horizon task execution.

The goal of this paper is not merely to list framework features, but to develop a systems-oriented structure for understanding how current multi-agent LLM systems are organized, how they communicate, and how they cope with failures in distributed execution settings. Based on this analysis, the paper identifies current limitations and outlines future directions toward more interoperable, observable, and fault-tolerant multi-agent ecosystems.

Keywords: large language model agents · multi-agent systems · distributed systems · orchestration · communication protocols · fault tolerance

1 Introduction

Large language models (LLMs) have rapidly extended the scope of intelligent systems from single-turn text generation to increasingly complex agentic workflows involving planning, tool use, reflection, and environment interaction. As this paradigm has matured, research has gradually moved beyond single-agent settings toward collaborative multi-agent systems, in which multiple LLM-driven agents cooperate to decompose tasks, exchange intermediate results, invoke external tools, and iteratively refine decisions. This shift reflects the growing recognition that many realistic tasks are too complex, too long-horizon, or too heterogeneous to be handled effectively by a single monolithic agent.

Recent representative frameworks illustrate this trend clearly. AutoGen popularized programmable multi-agent conversation as a practical paradigm for coordinating LLM-based agents. LangGraph emphasized graph-structured and stateful orchestration, making persistent execution and resumability central design goals. CrewAI focused on engineering-oriented workflow composition, including flows, routing logic, and state persistence. OpenAI Swarm, by contrast, adopted a lightweight handoff-based coordination model, highlighting explicit delegation and controllable interaction among agents. Although these frameworks differ substantially in abstraction level, execution model, and engineering maturity, they collectively demonstrate that multi-agent LLM systems are increasingly being treated as structured distributed software systems rather than as prompt-level compositions alone.

Figure 1 sketches the motivating setting of this paper: a user request is routed to multiple specialized agents, execution spans distributed workers and shared state, and failures must be detected and recovered rather than treated as rare exceptions.

Despite this rapid progress, much of the current literature still emphasizes agent capability at the task level, such as reasoning quality, tool use, or benchmark performance, while paying less attention to the distributed systems perspective needed for robust collaboration in practice. Real deployments must coordinate execution across multiple roles, tools, services, and sometimes heterogeneous runtimes. Under such conditions, orchestration topology, communication semantics, shared context, state persistence, resumability, and partial-failure recovery become first-class design concerns.

This paper therefore treats distributed multi-LLM-agent architectures not merely as collections of intelligent agents, but as coordination systems whose effectiveness depends on how collaboration is organized and sustained. Four design dimensions are particularly important: *orchestration topology*, *communication and routing*, *state synchronization and persistence*, and *fault tolerance*. These dimensions determine whether a system can support long-running workflows, dynamic delegation, tool failures, and human intervention without losing control or consistency.

A second motivation lies in the persistent gap between promising architectural ideas and robust real-world performance. Existing benchmarks show that long-horizon, tool-enhanced, and interactive tasks remain highly challenging even

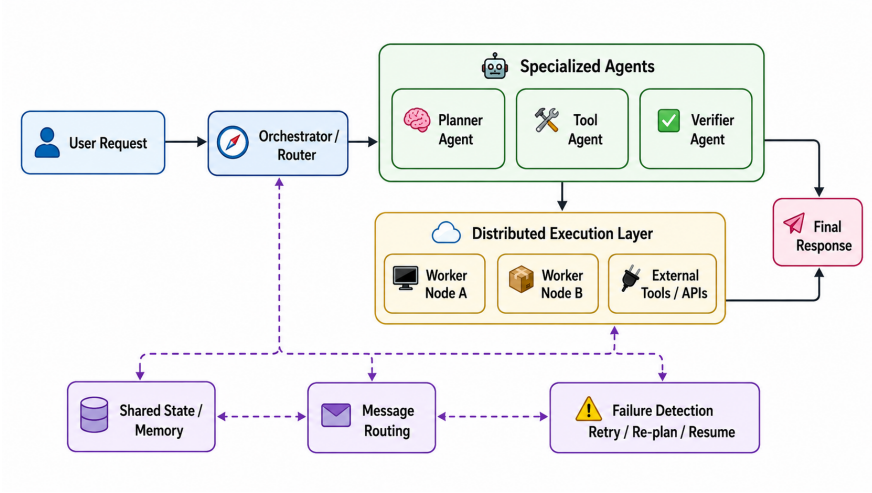

Fig. 1. A motivating distributed multi-agent execution scenario.

for strong LLM-based agents. These results suggest that performance depends not only on model quality, but also on system-level factors such as coordination overhead, message routing, memory handling, execution control, and recovery support. At the same time, the broader ecosystem is moving toward interoperability and protocolization, which further reinforces the need for a systems-oriented perspective.

Against this background, this paper examines distributed multi-LLM-agent architectures with a focus on orchestration patterns, communication protocols, and fault tolerance. Specifically, it studies four representative frameworks—*AutoGen, LangGraph, CrewAI, and OpenAI Swarm*—and compares them through a unified taxonomy of control topology, communication structure, state management, and recovery behavior. The paper does not aim to provide a feature-by-feature product comparison. Instead, it explains how architectural choices shape practical trade-offs in controllability, extensibility, resilience, and public-evidence behavior under distributed collaboration settings.

To guide the analysis, this paper addresses four research questions:

- **RQ1:** How do current multi-LLM-agent frameworks differ in orchestration topology?
- **RQ2:** How do these frameworks implement communication, routing, and state management?
- **RQ3:** What fault models and recovery strategies are supported in current systems?
- **RQ4:** What do public benchmark results and technical reports reveal about the practical trade-offs of these architectures?

The remainder of this paper is organized as follows. Section 2 identifies the main research gap and clarifies the position of this paper. Section 3 introduces

a taxonomy of distributed multi-agent collaboration. Section 4 analyzes representative frameworks together with benchmark and public evidence. Section 5 presents a comparative analysis of architectural trade-offs. Section 6 discusses broader implications, including the meaning of truly distributed coordination, the difficulty of fair comparison, and the evolution toward protocolized ecosystems. Section 7 concludes the paper and summarizes future directions.

2 Research Gap and Positioning

Existing studies have already established that LLM-based agents can perform planning, memory use, tool invocation, and task decomposition at increasingly complex levels [1]. However, most of this line of work still treats distributed coordination as a secondary implementation issue rather than as a primary analytical object. In other words, prior research has explained how agents can reason and act, but has been less explicit about how multiple agents are orchestrated, synchronized, and recovered in realistic distributed execution environments.

A similar limitation appears in recent work on multi-agent collaboration. Studies on collaboration mechanisms and communication have clarified how multiple agents may cooperate through role allocation, conversational exchange, or communication patterns [2, 3]. Yet these studies still stop short of a unified systems-level comparison of orchestration topology, communication structure, persistence, and fault tolerance across representative frameworks. What is still missing is not another detailed description of what multi-agent frameworks contain, but a more explicit account of what they *do not* solve equally well.

Protocol-oriented work and evaluation-oriented work provide further evidence of this gap. Interoperability efforts such as MCP and A2A indicate that communication boundaries and interface design are becoming increasingly important [4–8]. Meanwhile, benchmarks such as AgentBench, GAIA, and WebArena show that long-horizon, tool-augmented, and environment-interactive tasks remain difficult in practice [9–12]. These works offer useful evidence, but they do not by themselves explain how concrete framework-level choices in orchestration, state, and recovery contribute to those outcomes.

Therefore, the main gap addressed in this paper is the lack of a unified distributed-systems view across representative multi-agent frameworks. Rather than centering the analysis on framework features alone, this paper focuses on four system-level dimensions: orchestration topology, communication semantics, state synchronization and persistence, and fault tolerance. It then uses this structure to analyze AutoGen, LangGraph, CrewAI, and OpenAI Swarm, and to connect framework design with benchmark and public-evidence signals. In this sense, the contribution of the paper lies not in introducing another catalog of agent functions, but in clarifying architectural trade-offs that are often discussed separately but rarely analyzed together.

3 Taxonomy of Distributed Multi-LLM-Agent Collaboration

To compare current multi-LLM-agent frameworks in a consistent and system-oriented manner, this paper adopts a taxonomy centered on four key dimensions: *orchestration topology*, *communication semantics*, *state synchronization and persistence*, and *fault tolerance and recovery*. This taxonomy is motivated by both recent literature and the practical design choices exhibited by representative frameworks [1–3]. Rather than treating multi-agent systems as simple collections of role-specialized agents, the proposed taxonomy views them as distributed coordination systems whose practical behavior depends on how control is organized, how information is exchanged, how intermediate state is maintained, and how failures are handled.

3.1 Orchestration Topology

The first dimension is *orchestration topology*, which describes how control authority is distributed during execution. In current multi-agent frameworks, at least three recurring coordination patterns can be identified [2].

The first pattern is **centralized orchestration**, in which one coordinating component determines which agent should act next, aggregates intermediate outputs, and often decides when execution should terminate. This design is attractive because it simplifies control, observability, and debugging. However, it may also introduce bottlenecks and single points of failure, especially in long-running workflows or systems with heavy tool usage.

The second pattern is **hierarchical orchestration**, in which a supervisor or planner delegates subtasks to specialized downstream agents. Hierarchical coordination improves modularity and specialization, but it also increases routing complexity and makes recovery more dependent on the correctness of the upper-layer controller.

The third pattern is **handoff-based or near-peer coordination**, in which responsibility is transferred more dynamically among agents. In this pattern, execution does not always rely on a single persistent controller at every step; instead, one agent can pass context and control to another according to local decisions or task-specific rules. This design may improve flexibility and reduce excessive centralization, but it also makes consistency and observability more difficult to guarantee.

These topologies should not be interpreted as mutually exclusive categories. In practice, many frameworks combine them. For example, a system may use centralized routing at the workflow level while still allowing handoffs among worker agents at the task level. Therefore, orchestration is best understood as a spectrum rather than as a strict binary distinction between centralized and decentralized systems.

3.2 Communication Semantics

The second dimension is *communication semantics*, which concerns how agents exchange information, requests, decisions, and intermediate outputs. Existing frameworks differ not only in whether agents communicate, but also in *what* they exchange and *how* that exchange is structured [3].

One common mechanism is **shared-thread communication**, where multiple agents contribute to a single conversation context. This design simplifies collaboration because every agent can observe the same running history, but it may also lead to context bloat, ambiguity, and information redundancy as the interaction grows.

A second mechanism is **explicit routing through control logic**. In this case, communication is mediated by a scheduler, router, or graph transition rule that determines which message, state fragment, or tool output should be passed forward. Compared with shared-thread interaction, explicit routing provides stronger control and can reduce unnecessary context propagation, but it requires more careful workflow design.

A third mechanism is **protocol-mediated interaction**, which becomes increasingly important when agents, tools, and services are distributed across different runtimes or platforms. In this setting, communication is no longer limited to internal framework calls; instead, it may depend on explicit interface standards, typed messages, capability descriptions, or external discovery mechanisms [8, 4–7]. This distinction is important because internal framework communication and cross-platform interoperability do not solve the same problem, even though both belong to the broader communication layer.

Accordingly, this paper distinguishes between **intra-framework communication** and **inter-system interoperability**. The former concerns how agents exchange information within one framework instance, while the latter concerns how independently implemented agents, tools, or services communicate across ecosystem boundaries [8].

3.3 State Synchronization and Persistence

The third dimension is *state synchronization and persistence*. In distributed multi-agent systems, successful collaboration depends not only on message passing but also on how intermediate state is represented, updated, stored, and resumed. This issue is especially important in long-horizon tasks, tool-intensive workflows, and human-in-the-loop settings [1, 3].

A first distinction concerns **state representation**. Some systems primarily rely on conversational history as implicit state, while others define an explicit shared state object that can be read and updated by different components. Explicit state often improves inspectability and structured control, whereas implicit conversational state may be more flexible but harder to manage at scale.

A second distinction concerns **state scope**. State may be local to an individual agent, shared across a team of agents, or maintained globally at the workflow

level. Differences in scope directly affect information visibility, synchronization overhead, and the risk of stale or inconsistent context.

A third distinction concerns **persistence**. Some frameworks support checkpointing or resumable execution, allowing workflows to pause and continue after interruption. Others are lightweight and stateless between calls, placing the burden of persistence on the external application layer. This difference is highly consequential in practice: persistent systems are better suited for long-running and failure-prone execution, while stateless systems may remain simpler and easier to reason about in small-scale workflows.

For this reason, state management in this paper is analyzed along four sub-dimensions: **state unit** (message history, shared object, structured memory), **state scope** (local, shared, global), **persistence support** (none, partial, built-in), and **resumability** (manual reconstruction, checkpoint-based resume, durable execution).

3.4 Fault Tolerance and Recovery

The fourth dimension is *fault tolerance and recovery*. Existing multi-agent literature frequently emphasizes the benefits of collaboration, but real deployments must also cope with failure. In distributed multi-agent settings, failures may arise not only from infrastructure, but also from coordination logic and semantic drift [2, 3].

This paper groups failures into five broad categories. The first is **tool and environment failure**, such as API errors, timeouts, unavailable resources, or failed external actions. The second is **coordination failure**, including incorrect delegation, dead-end planning, and repeated handoff loops. The third is **state failure**, such as stale context, missing updates, or inconsistent intermediate memory. The fourth is **execution interruption**, where a workflow halts before completion because of system crashes, user pauses, or external dependency failure. The fifth is **semantic failure**, where agents continue operating but gradually converge on incorrect assumptions, misleading summaries, or false consensus.

Correspondingly, recovery mechanisms can also be classified. **Detection** mechanisms include guardrails, timeouts, validation checks, and human oversight. **Containment** mechanisms attempt to isolate errors by rerouting tasks, limiting propagation, or interrupting unstable branches. **Recovery** mechanisms include retry, rollback, checkpoint restore, human-in-the-loop correction, and explicit re-planning. A framework need not implement all of these natively, but its architecture strongly influences which of them can be supported efficiently.

3.5 Taxonomy Summary

Based on the four dimensions above, this paper evaluates representative frameworks using the following analytical questions:

- **Topology:** Is coordination primarily centralized, hierarchical, handoff-based, or mixed?

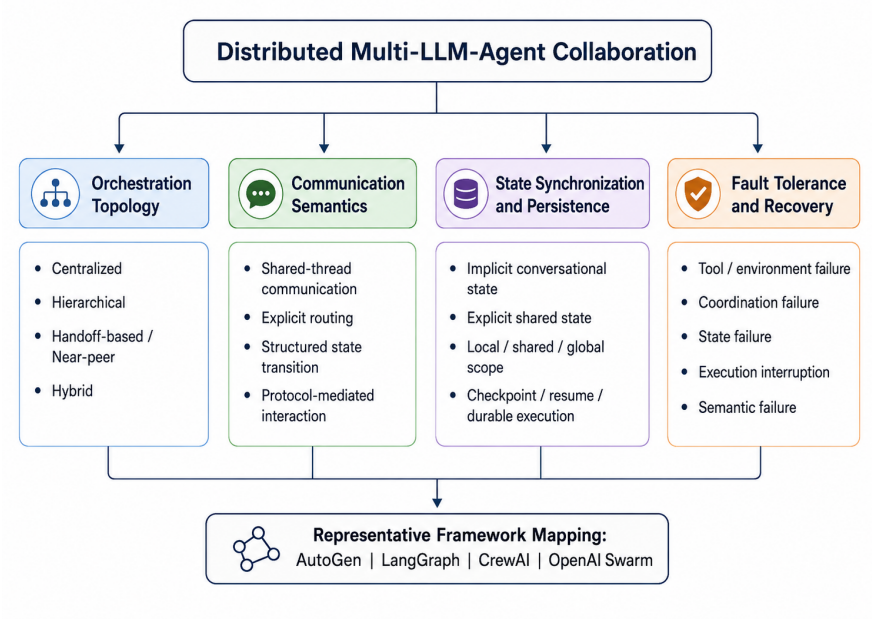

Fig. 2. A taxonomy of distributed multi-LLM-agent collaboration.

- **Communication:** Do agents interact through shared context, explicit routing, structured state transitions, or external protocols?
- **State:** Is execution state implicit or explicit, local or shared, ephemeral or persistent, resumable or stateless?
- **Fault tolerance:** What kinds of failures are visible, and what recovery strategies are supported by the framework design?

As illustrated in Fig. 2, the proposed taxonomy organizes current frameworks along four key dimensions: orchestration topology, communication semantics, state synchronization and persistence, and fault tolerance and recovery.

The purpose of this taxonomy is not merely to classify existing frameworks, but to provide a common analytical vocabulary for comparing systems that are otherwise described using different abstractions. In particular, the taxonomy highlights that coordination structure, communication mechanism, persistence model, and recovery support should be analyzed together rather than in isolation. This perspective will guide the framework analysis in Section 4 and the comparative discussion in Section 5. In the next section, this taxonomy is applied to four representative frameworks—AutoGen, LangGraph, CrewAI, and OpenAI Swarm—to examine how current systems differ in control, communication, persistence, and resilience [13–16].

4 Representative Frameworks and Public Evidence

This section applies the taxonomy from Section 3 to four representative frameworks: AutoGen, LangGraph, CrewAI, and OpenAI Swarm. The purpose is not to reproduce framework documentation, but to extract the architectural choices that matter for distributed execution: who controls progress, how messages or state are propagated, where persistence is maintained, and what recovery mechanisms can be supported. The section then connects these architectural observations with benchmark evidence, public serving reports, and protocol-oriented developments.

4.1 Compressed Framework Profiles

AutoGen AutoGen represents a conversation-centric approach to multi-agent coordination. Its core abstraction is the conversable agent, and its practical coordination patterns include two-agent chat, sequential chat, group chat, and nested chat [13, 17, 18]. In group-chat settings, a `GroupChatManager` can mediate speaker selection and message exchange, making AutoGen flexible for rapid prototyping and human-in-the-loop collaboration [19]. Architecturally, AutoGen therefore combines shared conversational context with light manager-mediated control.

The main trade-off is that communication and state are closely coupled. Message history serves as interaction log, implicit memory, and coordination substrate at the same time. This design is expressive, but long conversations may suffer from context growth, stale assumptions, and weaker separation between reasoning traces and durable workflow state. AutoGen supports resuming group chats from prior messages [20], but compared with explicit state-machine frameworks, its persistence and recovery semantics remain relatively lightweight.

LangGraph LangGraph is the clearest example of graph- and state-centric orchestration. It models execution through state, nodes, and edges, so coordination is expressed as explicit transitions rather than emergent conversational flow [14, 21]. This makes it suitable for long-running workflows, conditional routing, and applications where execution boundaries must be inspectable.

Its main strength is durable execution. LangGraph can persist execution progress and resume workflows from stored state, while its memory mechanisms distinguish thread-scoped and longer-term state [22, 23]. These features make failures more localizable: errors can often be associated with a node, transition, or state update rather than with an entire unstructured conversation. The cost is higher up-front design effort, because developers must define state schemas and graph transitions more explicitly.

CrewAI CrewAI combines agent collaboration with workflow-oriented process control. Its *Crews* organize agents and tasks, while *Flows* provide structured orchestration through event-style primitives such as `@start`, `@listen`, and

Table 1. Compact comparison of representative frameworks.

Framework	Topology	State / Persistence	Key architectural trait
AutoGen	Mixed, conversation-centric	Implicit chat state; partial resume	Flexible collaboration through reusable conversation patterns
LangGraph	Graph-based, state-centric	Explicit shared state; durable execution	Strong control for long-running and resilient workflows
CrewAI	Sequential / hierarchical flows	Persisted flow state; unified memory	Workflow-oriented orchestration with engineering focus
OpenAI Swarm	Handoff-driven, lightweight	Stateless between calls; external persistence	Minimal and transparent coordination baseline

`@router` [15, 24]. CrewAI also supports sequential and hierarchical processes, where a manager model or manager agent can supervise delegation and validation [25]. This makes it particularly relevant for enterprise-style workflows where task ownership and supervision must be visible.

CrewAI's architectural value lies in bridging multi-agent collaboration and workflow engineering. Flow persistence and memory support make it more operationally structured than purely conversational systems [24, 26]. At the same time, its abstractions are less low-level than a fully explicit graph runtime. As a result, CrewAI is well suited to process-oriented orchestration, but its recovery properties still depend on how developers structure flows, validation checkpoints, and persistent state.

OpenAI Swarm OpenAI Swarm is a deliberately lightweight orchestration baseline centered on agents and handoffs [16]. An active agent can call functions, update context variables, and transfer control to another agent. This design makes delegation explicit and easy to inspect, and it is useful as a minimal reference point for handoff-driven coordination.

The same minimalism also limits built-in persistence. The repository describes Swarm as stateless between calls, so long-running state, checkpointing, and durable recovery must be supplied by the surrounding application [16]. Function-call errors can be appended back into the interaction so that the agent can respond, but advanced recovery mechanisms such as checkpoint restore or workflow replay are not native runtime features. Swarm is therefore best understood as a transparent handoff model rather than as a full production-grade distributed execution system.

Table 2. Data-source matrix used in this paper.

Type	Source	Evidence	Use in this paper
Benchmark	AgentBench, GAIA, WebArena	task difficulty, long-horizon behavior	Shows where coordination, tool use, and stability remain difficult
Framework docs	AutoGen, LangGraph, CrewAI, Swarm	orchestration, memory, durability, handoff logic	Supports architectural analysis of control, state, and recovery
System report	vLLM, SGLang, LMDeploy, Elastic EP	throughput, TTFT, TPOT, recovery behavior	Supports public-evidence analysis and figure design
Protocol docs	MCP, A2A, protocol surveys	interface structure, capability exposure, cross-agent communication	Supports the discussion of protocol evolution and interoperability boundaries

4.2 Benchmark, Report, and Protocol Sources

To avoid mixing incomparable evidence, this paper separates framework documentation, agent benchmarks, public system reports, and protocol documents. Framework documentation is used to identify architectural mechanisms such as orchestration patterns, state persistence, and recovery support. Agent benchmarks reveal where long-horizon and tool-augmented tasks remain difficult. Public system reports expose serving-side metrics such as throughput, TTFT, TPOT, memory behavior, and recovery under partial failure. Protocol documents show how the ecosystem is moving from single-framework execution toward cross-system interoperability.

Table 2 clarifies the methodology of this paper. Framework documents, benchmarks, public serving reports, and protocol specifications operate at different layers of the stack. They should therefore be used together but not conflated. In particular, public reports are useful for extracting traceable evidence, but they should not be interpreted as direct apples-to-apples comparisons unless hardware, model size, prompt length, concurrency, and runtime configuration are controlled.

4.3 Data-Driven Evidence from Public Reports

Public evidence connects the architectural analysis above with measurable system behavior. This is useful because the practical question is not only whether agents can solve tasks, but also whether their orchestration choices remain observable, persistent, efficient, and recoverable under deployment constraints.

Serving-Efficiency Trends Official system reports suggest that serving efficiency has become a major axis of public evidence around LLM systems. The vLLM team reported in its 2024 performance update that v0.6.0 delivered 2.7×

Fig. 3. Public system evidence across serving efficiency, long-context scaling, concurrency, and partial-failure recovery. Colored bubbles summarize reported metrics, while the rightmost column highlights the corresponding systems-level implications.

higher throughput and 5× faster TPOT on a Llama 3 8B workload, as well as 1.8× higher throughput and 2× lower TPOT on a Llama 70B workload under the reported settings [27]. LMSYS reported that SGLang could achieve up to 3.1× higher throughput than vLLM on Llama-70B in its July 2024 serving post, while remaining competitive with TensorRT-LLM under the evaluated configurations [28]. These reports do not directly evaluate AutoGen, LangGraph, CrewAI, or Swarm, but they are relevant because distributed multi-agent systems ultimately depend on the surrounding serving stack.

Cross-System Metric Evidence To connect the architectural analysis with deployment-level evidence, Fig. 3 consolidates representative public reports across serving efficiency, token-level latency, long-context scalability, concurrency, and partial-failure recovery. Each row corresponds to a specific public report or deployment setting, while the metric columns show the dimensions explicitly discussed in that source. The rightmost column summarizes the main systems-level implication of each setting.

The first two rows summarize the vLLM v0.6 performance update. For the reported Llama 3 8B setting, the update showed a 2.7× throughput improvement and a 5× improvement in TPOT. For the larger 70B setting, the corresponding gains were 1.8× in throughput and approximately 2× in TPOT. The report also exposed an important systems observation: API-server processing and scheduling accounted for a substantial share of the execution path, showing that serving performance depends on more than GPU computation alone [27].

The SGLang Runtime row extends the evidence toward broader model and online-serving settings. Its public report covered Llama models ranging from 8B

to 405B, multiple GPU configurations, and both offline and online workloads. The reported evidence included up to 3.1× throughput improvement over the selected vLLM baseline, approximately 5,000 tokens per second in a short-input setting, and online request rates above 10 requests per second for the reported 8B configuration [28].

The long-context row shows how public reporting has expanded beyond conventional throughput. Under the reported 128K-input and 8K-output setting, the SGLang deployment reported 226.2 TPS/GPU, an 8.6-second best TTFT, and support for 288 concurrent requests. The associated systems design combined prefill–decode disaggregation, wide expert parallelism, multi-token prediction, and KV-aware routing, illustrating how long-context performance increasingly depends on orchestration across the serving runtime [30].

The final row focuses on partial-failure recovery. Elastic EP was evaluated under failures ranging from one to sixteen ranks. The public report showed that remaining throughput decreased as the number of failed ranks increased, while service interruption remained approximately 6.2–6.8 seconds. This represented an approximately 90% reduction compared with the reported full-restart time. The result highlights the role of expert redundancy, token rerouting, and localized recovery in resilient distributed execution [31].

Taken together, Fig. 3 shows that public system evidence has expanded from throughput and token latency toward long-context scaling, runtime coordination, concurrency, and failure recovery. LMDeploy's benchmark documentation provides a related metric vocabulary, including throughput, TTFT, token latency, percentile latency, and GPU memory usage [29].

Long-Horizon and Recovery-Aware Evidence Benchmark papers such as AgentBench, GAIA, and WebArena suggest that long-horizon, tool-augmented, and interactive environments remain difficult for current agent systems [9–12]. These results do not directly compare the four frameworks, but they explain why persistence, routing discipline, and recovery support matter: long tasks place sustained pressure on action loops, state continuity, tool invocation, and coordination under uncertainty.

More recent official reports show that public evidence is no longer limited to raw throughput. LMSYS reported in 2026 that SGLang's long-context serving on GB300 NVL72 reached 226.2 TPS/GPU at the reported 128K/8K setting and also reduced TTFT for 128K prefill under the tested configuration [30]. LMSYS also reported partial-failure evidence for SGLang Elastic EP, stating that service interruption remained under 10 seconds and represented about a 90% reduction compared with the 2–3 minutes typically required for a full restart [31]. These reports motivate a recovery-aware view of distributed agent systems: once multi-agent workflows depend on long execution traces, external tools, and serving infrastructure, fault tolerance becomes part of orchestration design rather than a purely backend concern.

Figure 4 illustrates this recovery-oriented perspective. Before the injected failure, all strategies maintain a relatively stable task success rate. After the failure

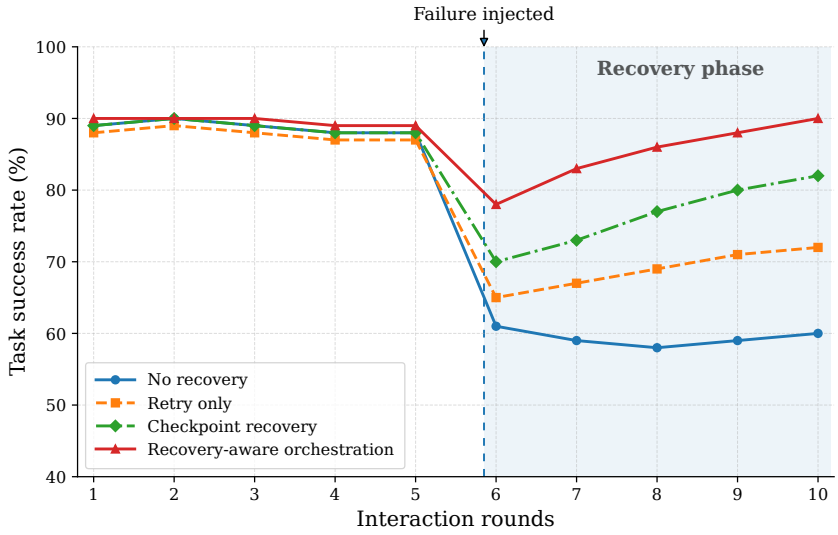

Fig. 4. Conceptual illustration of recovery trends under different fault-tolerance strategies after an injected failure.

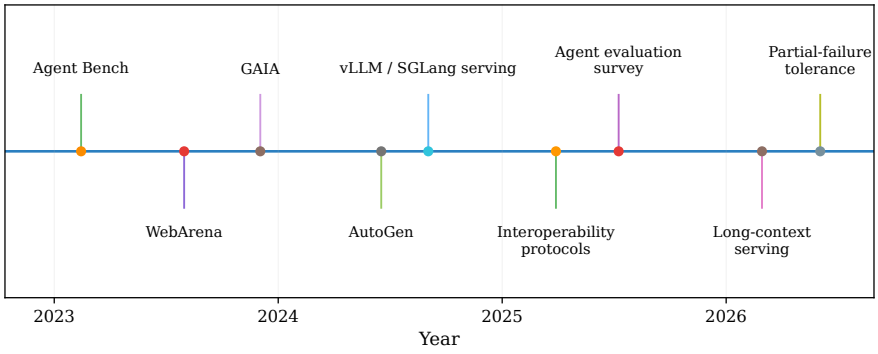

Fig. 5. Evolution of public evidence from benchmark difficulty and serving efficiency toward long-context and recovery-aware reporting.

point, the baseline without recovery degrades sharply and remains at a low success level. Retry-only recovery provides limited improvement, checkpoint-based recovery restores part of the lost performance, and recovery-aware orchestration shows a faster and more stable recovery trend. Since this figure is conceptual, it should be read together with the public recovery evidence discussed above rather than as a direct benchmark comparison among specific frameworks.

Figure 5 captures the broader evidence trajectory. The public landscape has expanded from early benchmark difficulty toward serving efficiency, interoperability, long-context scalability, and partial-failure recovery. This shift supports

the central argument of this paper: distributed multi-agent systems should be evaluated not only by task success, but also by their ability to maintain state, coordinate tools, and recover from failures in realistic deployment environments.

4.4 Section Summary

The framework profiles above show four different architectural philosophies: AutoGen prioritizes conversation-centric flexibility; LangGraph emphasizes explicit graph-based state orchestration and durable execution; CrewAI combines workflow hierarchy with persisted process control; and Swarm reduces coordination to lightweight handoffs with externalized state. The public-evidence analysis then shows that the relevant systems questions are not purely conceptual. Public reports increasingly expose concurrency, long-context pressure, latency distribution, memory behavior, and recovery under failure as practical dimensions. Fig. 3–5 therefore serve different roles: Fig. 3 organizes heterogeneous metric evidence, Fig. 4 illustrates recovery behavior after an injected failure, and Fig. 5 summarizes the evolution of public evidence. Together, these observations motivate the cross-framework synthesis in the next section.

5 Comparative Analysis

Building on the taxonomy in Section 3 and the framework profiles in Section 4, this section develops a cross-framework synthesis from a distributed-systems perspective. Rather than repeating individual framework descriptions, it compares the frameworks along four deployment-relevant dimensions: orchestration control, communication and state propagation, recovery and observability, and production concerns. Table 3 summarizes the comparison.

5.1 Control Topology: From Conversation to Explicit Workflow

The first major distinction concerns how strongly each framework makes control explicit. AutoGen favors flexible conversational coordination. Its reusable conversation patterns and group-chat management make it effective for exploratory collaboration, but the execution path can be less explicit than in graph-based systems [18, 19]. LangGraph places control in graph structure, node transitions, and shared state updates, making it more deterministic and easier to inspect [21, 14]. CrewAI emphasizes supervised process hierarchy through sequential and hierarchical execution, which is useful when task decomposition and validation need to be visible [25]. Swarm reduces control to active agents and explicit handoffs, offering transparency with minimal runtime assumptions [16].

These designs should not be reduced to a simple centralized/decentralized binary. In practice, most current frameworks distribute work without fully distributing authority. A workflow may involve multiple agents and remote tools while still depending on a scheduler, manager, graph runtime, or application loop. The more useful comparison is therefore a spectrum: AutoGen emphasizes

Table 3. Cross-framework synthesis of orchestration, communication, persistence, interoperability, and recovery.

Framework	Orchestration	Communication	Persistence	Interop.	Recovery capability
AutoGen	Mixed	Shared chat context; manager-mediated routing	Partial resume from history	Limited native cross-platform support	Flexible but weaker structured recovery in long threads
LangGraph	Graph-based	Explicit node transitions and shared state updates	Durable execution	Better tool/runtime integration alignment	Strong observability and checkpoint-like resume
CrewAI	Sequential / hierarchical	Task- and flow-oriented routing with manager supervision	Persisted flows and unified memory	Moderate interoperability via workflow integration	Good practical recovery basis through persisted flows
OpenAI Swarm	Handoff-driven	Active-agent messaging and explicit handoffs	Externalized to application layer	Lightweight but limited native interoperability semantics	Transparent local loop, weak durable recovery

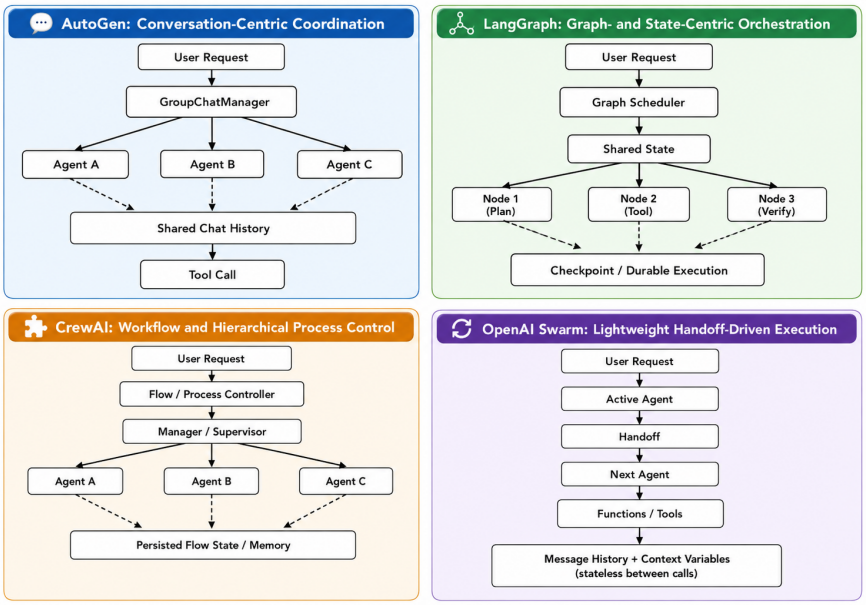

Fig. 6. Comparative architectural styles of representative multi-agent frameworks.

conversational flexibility, LangGraph emphasizes explicit workflow control, CrewAI emphasizes hierarchical process discipline, and Swarm emphasizes minimal handoff-based delegation.

Figure 6 visualizes this spectrum. The figure highlights that architectural differences are not merely differences in API syntax: they determine where routing logic lives, how agents share information, and whether state is implicit, explicit, persistent, or externalized.

5.2 Communication, State Synchronization, and Persistence

The second major distinction concerns how information and state move through a workflow. AutoGen uses message history as both communication substrate and implicit state [18, 19]. This increases flexibility, but can also produce context growth and ambiguity. LangGraph separates communication from free-form conversation more strongly by representing progress through state updates and graph transitions [21]. This improves inspectability and makes recovery boundaries clearer. CrewAI sits between these two extremes by combining collaborative agents with workflow-oriented Flows, routing, and persisted state [24, 15]. Swarm keeps communication minimal: the active agent, message list, context variables, and handoff target determine execution continuity [16].

Persistence follows the same pattern. LangGraph provides the strongest built-in durable execution support [22]. CrewAI provides practical persistence through Flows and memory abstractions [24, 26]. AutoGen offers lighter-weight resumability through message reuse and group-chat continuation [20]. Swarm explicitly externalizes persistence to the calling application [16]. Therefore, the key trade-off is not simply whether a framework supports memory, but whether persistence is a native execution property or an application-level responsibility.

5.3 Fault Tolerance, Recovery, and Observability

Fault tolerance is where the architectural differences become most visible. LangGraph has the strongest recovery-oriented structure because node boundaries and durable execution make interruption and resumption explicit [22]. CrewAI also provides a useful recovery basis through manager-level validation and persisted Flow state [25, 24]. AutoGen can be made robust by adding validators, human checkpoints, or specialized correction agents, but failure localization is harder when reasoning, memory, and coordination are all interleaved in a shared conversation [18]. Swarm is transparent at the local loop level and can pass function-call errors back into the interaction, but it lacks native checkpointing or long-running workflow recovery [16].

Observability follows a similar pattern. Explicit workflow boundaries make it easier to trace which component acted, what state changed, and where a failure occurred. Thus, LangGraph and CrewAI are generally better aligned with production tracing and debugging. AutoGen provides strong interactive flexibility, but large shared threads can obscure root-cause analysis. Swarm is easy to inspect locally, but cross-run observability depends on external logging and state management. Overall, frameworks that make control and state explicit tend to support stronger observability and recovery, while frameworks that emphasize conversational flexibility tend to trade structural guarantees for adaptability.

5.4 Deployment Concerns: Monitoring, Cost, Scalability, and Governance

A deployment-oriented analysis must go beyond whether a framework can complete a task once. Production multi-agent systems must remain monitorable, cost-aware, scalable, and governable under repeated execution. Four concerns are especially important.

First, *monitoring and traceability* are essential for long-horizon workflows. Failures may arise from tool errors, stale context, repeated handoff loops, invalid state updates, or latency accumulation. Frameworks with explicit state and workflow boundaries make such failures easier to localize. Second, *token and latency overhead* can dominate runtime cost. Multi-agent collaboration introduces routing messages, validation prompts, summaries, memory retrievals, and tool feedback loops. Without controlled context propagation, an apparently modular design can become expensive and slow. Third, *scalability bottlenecks* appear when centralized managers, schedulers, or shared-memory components must coordinate many agents or tools. Strong central control improves consistency but may limit throughput; looser handoff models reduce some bottlenecks but complicate debugging and recovery. Fourth, *governance and safety* matter because multi-agent systems can trigger external tools, exchange sensitive context, and amplify erroneous assumptions across agents. Audit trails, permission boundaries, human checkpoints, and tool-level policy enforcement are therefore deployment concerns rather than optional features.

These concerns reinforce the central claim of this paper: orchestration architecture should be evaluated together with runtime and operational constraints. A lightweight framework may be ideal for prototyping, but it may require substantial external infrastructure for tracing, persistence, and policy control. A more structured framework may require more design work, but can reduce operational risk in long-running or failure-prone deployments. Therefore, deployment suitability depends on the required balance among flexibility, cost, observability, persistence, and recovery.

5.5 Design Implications

Three design implications follow. First, orchestration topology should be treated as a first-class architectural decision rather than as a by-product of framework choice. Second, persistence and recovery should be designed into the orchestration layer whenever workflows are long-running, tool-heavy, or user-facing. Third, interoperability protocols can reduce integration friction, but they do not replace runtime semantics for state, tracing, and recovery. These implications motivate the discussion in the next section, where the paper turns from framework comparison to the broader evolution of distributed multi-agent ecosystems.

6 Discussion

The preceding sections show that current multi-agent LLM systems already exhibit substantial diversity in orchestration style, communication structure, per-

sistence model, and recovery support. This section discusses three broader implications: the limited sense in which current systems are distributed, the difficulty of fair comparison, and the evolution from framework-specific design toward protocolized ecosystems.

6.1 Are Current Multi-Agent Frameworks Truly Distributed?

In a loose engineering sense, many current multi-agent systems are distributed because they invoke external tools, call remote services, interact with heterogeneous runtimes, or coordinate multiple workers. In a stricter systems sense, however, most representative frameworks are better understood as *distributed coordination layers* rather than fully decentralized runtimes.

The reason is that control remains highly structured. AutoGen depends on shared chat context, conversation patterns, and manager-like coordination [18, 19]. LangGraph explicitly models execution through graph structure, shared state, and durable execution [21, 22]. CrewAI emphasizes flows, processes, and manager-led coordination [25, 24]. Swarm appears more decentralized because of handoffs, but its execution still depends on an application-managed loop and externalized persistence [16]. The current ecosystem therefore distributes work more than it distributes authority.

This distinction matters. A workflow may involve many agents and services while still depending on a single scheduler, state representation, or execution loop. A stronger notion of distributed multi-agent execution would require clearer partial autonomy, explicit coordination semantics across services, local failure containment, and defined rules for state propagation and recovery. Current frameworks are moving in this direction, but they have not yet converged on a mature distributed execution model.

6.2 Why Fair Comparison Remains Difficult

Fair comparison is difficult because available evidence operates at different layers. Framework documentation describes mechanisms; agent benchmarks reveal task difficulty; serving reports expose throughput, latency, memory, and recovery behavior; protocol documents describe interface boundaries. These sources are useful, but they do not answer the same question. A framework that appears strong architecturally may be evaluated with a different model, serving stack, prompt budget, hardware platform, or tool set than another framework.

This problem resembles broader benchmarking challenges in systems evaluation. MLPerf Inference, for example, defines scenario-specific metrics, latency constraints, and consistency requirements for the system under test [32, 33]. By contrast, much public evidence in the agent ecosystem is engineering-oriented rather than benchmark-standardized. It is valuable for extracting signals, but it is not equivalent to a controlled comparison.

For multi-agent systems, fair evaluation should be layered. At the model layer, comparisons should control for model family, precision, and tokenizer behavior. At the serving layer, they should control for hardware, batching, con-

currency, and context length. At the agent layer, they should control for tools, prompt budgets, planning strategy, memory design, and stopping criteria. At the workflow layer, they should control for orchestration topology and recovery policy. Most current evidence does not normalize all of these layers at once. Therefore, the defensible conclusion is comparative but cautious: public evidence can support architectural insight, but not universal ranking claims.

6.3 Protocol Evolution and Interoperability Boundaries

The ecosystem is also moving from framework-specific execution toward protocolized interoperability. MCP is a clear signal of this shift: its specification defines a host/client/server architecture, JSON-RPC message format, stateful connections, capability negotiation, and structured exposure of resources, prompts, and tools [5]. A2A similarly aims to let agents built by different vendors and frameworks communicate and coordinate actions across heterogeneous enterprise environments [6, 7]. Recent protocol-oriented work further treats MCP, ACP, A2A, and ANP as part of a broader interoperability layer for agent systems [8].

This evolution changes the unit of analysis. Earlier multi-agent systems could keep most communication assumptions inside a single framework runtime. Protocol-oriented systems instead push agents, tools, and services across framework boundaries, organizational domains, and execution environments. In such settings, the key question is no longer only whether components can connect, but whether they can exchange structured capability descriptions, preserve execution intent, support tracing, and recover across system boundaries.

However, protocolization is not a complete solution to distributed coordination. Protocols standardize interfaces, discovery, message structure, and capability exposure, but they do not automatically define state consistency, checkpoint semantics, replay behavior, delegation policy, trust management, or recovery after partial failure. MCP can expose tools and resources to an LLM application, but it does not decide how a multi-agent workflow should arbitrate among competing plans or resume after interruption. A2A can support cross-agent communication, but it does not by itself provide a universal execution model for observability and recovery.

This suggests a layered future. Protocols provide interoperability, serving systems provide scalable execution, and orchestration frameworks provide control logic, persistence, and recovery structure. Figure 7 summarizes this layered interpretation.

In such an ecosystem, the central design question shifts from "Which framework is best?" to "How should protocol, runtime, and orchestration layers be composed for a given deployment setting?" Protocols lower integration cost, but they also make runtime questions more visible. Once agents can interact across boundaries, trust, observability, state recovery, and fault containment become more important rather than less.

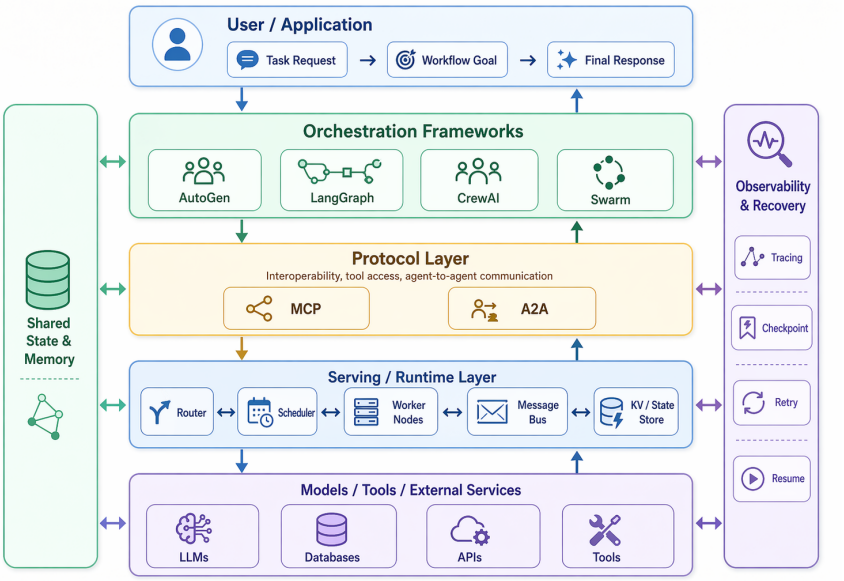

Fig. 7. A layered view of distributed multi-agent ecosystems across orchestration, protocol, runtime, and service layers.

6.4 Discussion Summary

The discussion leads to three conclusions. First, current frameworks are distributed operationally, but only partially distributed in authority and runtime semantics. Second, fair comparison remains difficult because current evidence mixes framework documentation, task benchmarks, serving reports, and protocol specifications. Third, protocol evolution is an enabling trend, but standardized interfaces do not replace the need for explicit state management, observability, and recovery-aware orchestration.

7 Conclusion

This paper examined distributed multi-LLM-agent architectures through the lens of orchestration patterns, communication protocols, and fault tolerance. Instead of treating multi-agent systems as collections of role-specialized agents alone, it analyzed them as coordination systems whose behavior depends on control topology, communication semantics, state persistence, and recovery design. The proposed taxonomy organized these concerns into four dimensions: orchestration topology, communication semantics, state synchronization and persistence, and fault tolerance and recovery.

The framework analysis showed that AutoGen, LangGraph, CrewAI, and OpenAI Swarm represent different points in the current design space. AutoGen

emphasizes conversation-centric flexibility; LangGraph emphasizes explicit stateful orchestration and durable execution; CrewAI combines workflow hierarchy with persisted process control; and Swarm provides a lightweight handoff-driven baseline with externalized persistence [13–16]. These choices directly affect observability, resumability, controllability, deployment cost, and recovery behavior.

The public-evidence analysis further showed that architectural comparison cannot be separated from deployment concerns. Benchmarks such as AgentBench, GAIA, and WebArena indicate that long-horizon and tool-augmented tasks remain difficult [9–11]. Public system reports expose throughput, token latency, long-context pressure, and recovery behavior as practical constraints [27, 28, 30, 31]. Protocol efforts such as MCP and A2A show that the ecosystem is evolving toward cross-framework interoperability, but protocols alone do not solve state consistency, observability, or recovery semantics [5–8].

Overall, the current generation of multi-agent LLM frameworks has demonstrated the value of collaborative orchestration, but it has not yet converged on a mature distributed execution model. Future progress will require stronger structured state management, recovery-aware orchestration, deployment-oriented observability, and evaluation methods that separate model quality, serving infrastructure, workflow design, and protocol interoperability. The most promising direction is therefore not a single universal framework, but a layered ecosystem in which protocols, runtimes, and orchestration frameworks work together to support interoperable, observable, and fault-tolerant multi-agent systems.

References

1. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Xu, C., Lin, Y., Zhao, W.X., Wei, Z., Wen, J.-R.: A Survey on Large Language Model Based Autonomous Agents. *Frontiers of Computer Science* (2024)
2. Tran, K.T., Zhang, X., Chung, T., et al.: Multi-Agent Collaboration Mechanisms: A Survey of LLMs. *arXiv preprint arXiv:2501.06322* (2025)
3. Yan, B., Zhang, X., Zhang, L., Zhang, L., Zhou, Z., Miao, D., Li, C.: Beyond Self-Talk: A Communication-Centric Survey of LLM-Based Multi-Agent Systems. *arXiv preprint arXiv:2502.14321* (2025)
4. Anthropic: Introducing the Model Context Protocol. `https://www.anthropic.com/news/model-context-protocol`
5. Model Context Protocol: Model Context Protocol Specification. `https://modelcontextprotocol.io/specification/2025-06-18`
6. Google Developers Blog: A New Era of Agent Interoperability: Agent2Agent Protocol. `https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/`
7. A2A Project: Agent2Agent Protocol Repository. `https://github.com/a2aproject/A2A`
8. Ehtesham, A., Singh, A., Gupta, G.K., Kumar, S.: A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP). *arXiv preprint arXiv:2505.02279* (2025)
9. Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., et al.: AgentBench: Evaluating LLMs as Agents. *arXiv preprint arXiv:2308.03688* (2023)

10. Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., Scialom, T.: GAIA: A Benchmark for General AI Assistants. *arXiv preprint arXiv:2311.12983* (2023)
11. Zhou, S., Xu, F.F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., et al.: WebArena: A Realistic Web Environment for Building Autonomous Agents. *arXiv preprint arXiv:2307.13854* (2024 version)
12. Yehudai, A., Eden, L., Li, A., Uziel, G., Zhao, Y., Bar-Haim, R., Cohan, A., Shmueli-Scheuer, M.: Survey on Evaluation of LLM-based Agents. *arXiv preprint arXiv:2503.16416* (2025)
13. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., et al.: AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In: *Proceedings of COLM* (2024)
14. LangChain: LangGraph Documentation. `https://docs.langchain.com/oss/python/langgraph/overview`
15. CrewAI: CrewAI Documentation. `https://docs.crewai.com/`
16. OpenAI: Swarm GitHub Repository. `https://github.com/openai/swarm`
17. Microsoft: Multi-agent Conversation Framework | AutoGen 0.2. `https://microsoft.github.io/autogen/0.2/docs/Use-Cases/agent_chat/`
18. Microsoft: Conversation Patterns | AutoGen 0.2. `https://microsoft.github.io/autogen/0.2/docs/tutorial/conversation-patterns/`
19. Microsoft: Group Chat | AutoGen 0.2. `https://microsoft.github.io/autogen/0.2/docs/notebooks/agentchat_groupchat/`
20. Microsoft: Resuming a GroupChat | AutoGen 0.2. `https://microsoft.github.io/autogen/0.2/docs/topics/groupchat/resuming_groupchat/`
21. LangChain: Graph API Overview - LangGraph. `https://docs.langchain.com/oss/python/langgraph/graph-api`
22. LangChain: Durable Execution - LangGraph. `https://docs.langchain.com/oss/python/langgraph/durable-execution`
23. LangChain: Memory Overview - LangGraph. `https://docs.langchain.com/oss/python/langgraph/memory`
24. CrewAI: Flows. `https://docs.crewai.com/en/concepts/flows`
25. CrewAI: Processes. `https://docs.crewai.com/en/concepts/processes`
26. CrewAI: Memory. `https://docs.crewai.com/en/concepts/memory`
27. vLLM Team: vLLM v0.6.0: 2.7x Throughput Improvement and 5x Latency Reduction. `https://vllm.ai/blog/perf-update`
28. LMSYS Org: Achieving Faster Open-Source Llama3 Serving with SGLang Runtime. `https://lmsys.org/blog/2024-07-25-sglang-llama3/`
29. LMDeploy: Profile Token Latency and Throughput. `https://lmdeploy.readthedocs.io/en/v0.2.5/benchmark/profile_generation.html`
30. LMSYS Org: Deploying DeepSeek on GB300 NVL72: Big Wins in Long-Context Serving. `https://lmsys.org/blog/2026-02-19-gb300-longctx/`
31. LMSYS Org: Elastic EP in SGLang: Achieving Partial Failure Tolerance. `https://lmsys.org/blog/2026-03-25-eep-partial-failure-tolerance/`
32. MLCommons: MLPerf Inference: Datacenter. `https://mlcommons.org/benchmarks/inference-datacenter/`
33. MLCommons: MLPerf Inference Rules. `https://github.com/mlcommons/inference_policies/blob/master/inference_rules.adoc`