PANDA: A DECENTRALIZED ARCHITECTURE WITH FLEXIBLE ORCHESTRATION FOR SCALABLE, FAULT-TOLERANT MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing architectures for LLM-based multi-agent systems (MAS) cannot reliably and efficiently solve multi-step tasks at scale: they struggle to support large numbers of agents and concurrent tasks, tolerate failures, govern agent interactions, and accommodate the diverse planning and execution patterns different tasks require. We present **PANDA**, a decentralized architecture that connects a large collective of heterogeneous, independently administered agents, letting them discover each other's capabilities and self-organize into small specialized teams per task. **PANDA** scales by decoupling collective communication from team communication, allowing agents to participate in multiple teams simultaneously, load-balancing tasks across the collective, and scheduling concurrent work within each agent. **PANDA** further separates the underlying architecture from the orchestration strategy, supporting three planning and execution patterns (star, chain, and mesh) that can be selected according to the structure and requirements of each task. **PANDA** detects infrastructure and orchestration failures and recovers affected tasks by dynamically replanning around failed components. Finally, to provide governance without a centralized service that would limit scalability, **PANDA** uses a web-of-trust model to constrain agent interactions to established trust relationships. We evaluate **PANDA** on the HotPotQA benchmark, demonstrating that it scales to thousands of agents, assembles teams in milliseconds, matches state-of-the-art accuracy at up to $8\times$ the efficiency, and sustains 100% task completion under faults where existing systems fail.

1 INTRODUCTION

LLM-based agents are increasingly specialized through tools, knowledge, and skills for particular tasks. However, real-world problems often involve multiple steps spanning different domains and therefore require capabilities beyond those of any single specialized agent. Multi-agent systems (MAS) enable agents with complementary capabilities to divide work, exchange intermediate results, and collaborate, making them a promising paradigm for solving complex, multi-step tasks.

Realizing this potential requires MAS to support large, dynamic collectives in which many agents may concurrently collaborate on different tasks. We identify four core requirements for such systems: (1) scalability, (2) fault tolerance, (3) flexible orchestration, and (4) governance. Scalability requires supporting growth in the number of agents, concurrent tasks, and interactions while maintaining task utility and controlling the communication and token costs incurred to achieve it. Fault tolerance requires tasks to continue despite failures of agents or the underlying infrastructure. Flexible orchestration requires supporting multiple planning and execution models to accommodate different tasks and teams. Lastly, governance requires establishing trust and constraining interactions to trusted agents as collectives extend beyond a single administration.

MAS are difficult to scale because centralized agent registries, orchestration, and governance providers become bottlenecks as the number of agents and tasks grows. At the same time, agent discovery, coordination, communication, and shared-state management generate increasing latency and token costs. At scale, component failures become more likely, further increasing recovery overhead, causing system costs to grow faster than task utility.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison of **PANDA** with existing multi-agent systems. ✓ = supported, ∼ = partially supported, ✗ = not supported. Flexible refers to flexible orchestration patterns.

System	Scalable	Fault Tolerant	Flexible	Governance	Decentralized
Magentic-One (Fourney et al., 2024)	✗	✗	✗	✗	✗
Internet of Agents (Chen et al., 2025)	∼	✗	∼	✗	✗
Symphony (Wang et al., 2025)	∼	∼	✗	✗	∼
MACNET (Qian et al., 2025)	∼	✗	∼	✗	✓
AgentNet (Yang et al., 2026)	∼	∼	✗	✗	✓
PANDA (our solution)	✓	✓	✓	✓	✓

Limitations of previous work. First, current MAS architectures have limited scalability. Centralized designs such as Magentic-One (Fourney et al., 2024) concentrate planning and coordination in a single component, creating a bottleneck as the number of agents and tasks increases. Decentralized alternatives also face scalability limitations: AgentNet (Yang et al., 2026) requires every agent to initially know all other agents and then prunes connections over time, and its decentralized planner incur high agent-selection costs when choosing among thousands of candidates at each step. Symphony Wang et al. (2025) and Internet of Agents (Chen et al., 2025) are also limited by centralized communications hubs.

Second, existing governance solutions rely on centralized authorities, limiting their scalability and applicability to decentralized deployments. SAGA (Syros et al., 2026) and MAGIQ (Avizeh et al., 2026) rely on a provider to establish agent identities and enforce policies, and Laws et al. (2026) extends the provider to be resilient in a distributed trust setting, but the architecture remains logically centralized. As the number of agents and interactions grows, the authority becomes a bottleneck and a single point of failure. Moreover, such an authority may not exist in decentralized settings, where no single party can be trusted to authenticate, authorize, or vouch for all participating agents.

Third, existing MAS architectures provide limited orchestration flexibility. Magentic-One and MetaGPT (Hong et al., 2024), impose centralized planning mechanisms, limiting their ability to support different planning and execution models across tasks. Internet of Agents (Chen et al., 2025) supports more flexible coordination, but routes interactions through a centralized server, limiting its scalability. MACNET (Qian et al., 2025) supports different orchestration patterns but requires agents to be manually organized before execution, preventing runtime flexibility.

Finally, existing MAS provide only limited fault tolerance. Prior work handles model and communication errors through retries, fallback responses, redundancy, or consensus (Gao et al., 2024; Yang et al., 2026; Zheng et al., 2026). However, these approaches do not provide comprehensive recovery from infrastructure and orchestration failures, including preserving execution state, reassigning unfinished work, and continuing partially completed workflows. Consequently, failures can still disrupt execution or require tasks to restart at higher token and communication cost.

Our contribution. We present **PANDA** (**P**lanning **A**gents in a **N**etworked **D**ecentralized **A**rchitecture), a decentralized architecture that connects a large collective of heterogeneous, independently administered agents, letting them discover each other's capabilities and self-organize into small specialized teams per task. **PANDA** scales by decoupling collective communication from team communication, allowing agents to participate in multiple teams simultaneously, load-balancing tasks across the collective, and scheduling concurrent work within each agent. **PANDA** further separates the underlying architecture from the orchestration strategy, supporting three planning and execution patterns (star, chain, and mesh) that can be selected according to the structure and requirements of each task. **PANDA** detects infrastructure and orchestration failures and recovers affected tasks by dynamically replanning around failed components. Finally, to provide governance without a centralized service that would limit scalability, **PANDA** uses a web-of-trust model to constrain agent interactions to established trust relationships. Although decentralized trust and reputation models have been studied in traditional MAS, we are unaware of any LLM-based MAS architecture that uses a web of trust for decentralized governance. We summarize the contributions of **PANDA** over previous work in Table 1.

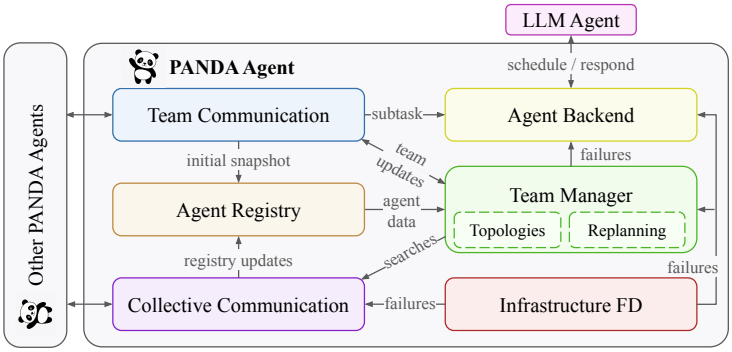


Figure 1: Internal design of a **PANDA** agent. *Team Communication* handles incoming and outgoing messages for any teams the agent is a member of. The *Agent Registry* maintains state about other agents in the collective, including their capabilities and governance information. *Collective Communication* manages propagation and reception of system-wide messages such as agent joins and leaves. The *Team Manager* assembles teams and maintains state about currently active ones. It also possesses two subunits that handle the logic for the different planning and execution topologies and replanning when failures occur. The *Infrastructure Failure Detector (FD)* detects failures and relays them both internally and to the rest of the collective. The *Agent Backend* is our wrapper around any custom or existing *LLM Agent* that maintains stateful information relevant to **PANDA** about the agent. In turn, the agent must expose a list of capabilities C and implement a function $f(c \in C, \text{request}) \mapsto \text{respose}$.

Results. We evaluate **PANDA** on the HotPotQA benchmark (Yang et al., 2018) and demonstrate three key properties across all topologies. (1) **PANDA** scales efficiently to large collectives under heavy workloads, assembling teams in milliseconds with per-assembly cost that remains constant even at 10^6 agents, and solving tasks up to $8\times$ faster than baselines while achieving comparable accuracy. (2) **PANDA** is resilient to both infrastructure and orchestration faults, maintaining high accuracy and 100% task completion where existing systems fail. (3) **PANDA**'s decentralized governance establishes tunable trust across the collective, letting agents configure how far trust propagates while maintaining $> 99\%$ precision under churn.

2 PANDA Architecture

In this section we present the architecture of our system, **PANDA**. We first present a high-level overview and then provide details about the main components.

2.1 Design Goals and Overview

PANDA is a MAS designed around four central goals. **(1) Scalability:** **PANDA** must handle massive numbers of agents and concurrent tasks, with per-task cost that grows sub-linearly with respect to the collective so that adding agents expands capability without degrading throughput. **(2) Governance:** Agents should be able to establish trusted connections and control who they interact with within the collective. **(3) Flexible planning and execution patterns:** Rather than committing to a single topology, **PANDA** must support heterogeneous agents and arbitrary communication patterns so that each task can be solved with the structure best suited to it. **(4) Fault tolerance:** Agents may crash, become unreachable, return malformed outputs, or infrastructure links may drop; **PANDA** must detect these conditions and complete tasks reliably despite them, without operator intervention.

To provide scalability and eliminate single points of failure, **PANDA** adopts a decentralized (peer-to-peer) architecture. **PANDA** provides the infrastructure and abstractions that allow independently created agents to discover other agents, establish teams, and coordinate to solve complex tasks. Figure 1 visualizes the internals of a single **PANDA** agent. **PANDA** supports any LLM agent, as long as it implements the function required by Agent Backend module. This module manages **PANDA**-specific state for the agent and handles local subtask scheduling, since an agent can be part of multiple teams working on separate tasks concurrently. Communication takes place over two separate services,

a collective communication service, optimized for scalability, and a team communication service, optimized for reliability. More details about scalability mechanisms are presented in Section 2.2.

There is no central authority or central repository in **PANDA**. Instead, each **PANDA** agent maintains a local registry tracking a subset of other agents, their capabilities, how to reach them, and the information required for governance (public key certificate). The registry is initialized when an agent joins the collective through a current member, and updated as the collective state changes. To enable sublinear capability lookups, each registry maintains an inverted index mapping capabilities to the agents that provide them. Agents establish trust using a self-governance mechanism based on web-of-trust and described in detail in Section 2.3.

PANDA decouples the communication topology from the orchestration logic, to ensure efficient task planning and execution. It supports three (star, chain, and mesh) topologies and the *Team Manager* allows the selection of the appropriate orchestration strategy handling the logic for planning and execution. Details about the different orchestration mechanisms are presented in Section 2.4.

PANDA provides support for infrastructure and orchestration failure detection and recovery. Each agent contains an Infrastructure Failure Detector (FD) module that detects failures and relays them both internally and to the rest of the collective, along with mechanisms for swapping out crashed or misconfigured agents. Failure detection and replanning strategies are described in Section 2.5.

2.2 SCALABILITY

PANDA targets scalability through multiple mechanisms: (1) decentralized registries and governance; (2) highly-efficient communication, scheduling, and load balancing services; (3) flexible orchestration beyond centralized approaches; (4) fault-tolerant team assembly, task planning, and execution. Below we present the system level scalability mechanisms, and describe the governance, flexible orchestration, and fault tolerance mechanisms in the following subsections.

Communication services. Maintaining a full mesh of reliable connections between agents is intractable at scale. **PANDA** decouples communication into two services, a collective communication service optimized for scalability and relying on gossip protocols (our implementation uses GossipSub) (Vyzovitis et al., 2020) and a team communication service that is instantiated on the fly for each team and is designed for reliability (our implementation uses TCP).

Scheduling. To minimize idle time, **PANDA** agents can belong to multiple teams simultaneously. To facilitate this, we introduce a scheduling mechanism inspired by a classical CPU. Instead of immediately executing assigned subtasks, agents enqueue their work, and an internal scheduling policy dispatches subtasks as the underlying agent becomes available. We support first-in-first-out (FIFO) and lottery scheduling, but more sophisticated policies can be added. This is distinct from LLM serving (Luo et al., 2025; Shahout et al., 2025) as it operates at the agent level.

Load Balancing. To prevent any single agent from being overloaded, we employ a load-balancing mechanism adapted from Ousterhout et al. (2013). During team assembly, for each capability c with redundancy $r(c)$, we poll $d \cdot r(c)$ agents with capability c and ask each how many teams they currently belongs to; the $r(c)$ least loaded of the group joins the new team. Ousterhout et al. (2013) demonstrates that a probe ratio $d = 2$ performs best under high cluster load with over $2.5\times$ faster median response time compared to random sampling ($d = 1$). When $d = 2$ and $r(c) = 1$, this reduces to Mitzenmacher (2002)'s Power of Two Choices.

2.3 WEB-OF-TRUST GOVERNANCE

PANDA's governance is inspired by the web of trust (WoT) introduced by PGP (Zimmermann, 1995). WoT has been used in settings such as gating Debian Developer status (Debian Project), but to our knowledge **PANDA** is the first to apply it as a concrete governance mechanism for LLM-based MAS.

We formalize our governance as follows. Each agent A_i holds a keypair (pk_i, sk_i) from a signature scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. When an agent A_i wants to join the collective, they must be certified by one or more existing member of the collective after providing out of band proof of their identity.[1] When an existing member A_j certifies a new agent A_i, A_j issues a certificate ($\mathrm{cert}_{j\to i}$) binding A_i's

[1] For instance, via a dedicated key-signing party or an existing real-world connection.

key and identity together with its advertised capability set c_i and issuance timestamp ts:

$$\mathrm{cert}_{j\to i} = \big(pk_i,\ \mathrm{id}_i,\ c_i,\ ts,\ \sigma_{j\to i}\big), \qquad \sigma_{j\to i} = \mathsf{Sign}_{sk_j}\big(pk_i \,\|\, \mathrm{id}_i \,\|\, c_i \,\|\, ts\big). \tag{1}$$

A_i will also often certify A_j assuming the out of band proof was exchanged mutually. Certificates are gossiped alongside capability announcements, so each agent's local registry contains a partial view of the certification graph, with a directed edge $A_j \to A_i$ for every known certificate $\mathrm{cert}_{j\to i}$. The collective is bootstrapped by one or more trusted *genesis* nodes: a single genesis node self-certifies, or multiple genesis nodes mutually certify one another, establishing the initial trust base.

Trust is anchored at a specific agent and extended by depth-bounded reachability over the certification graph. A_i's trusted set contains every peer reachable from its anchor via a chain of at most n certifications. **PANDA** supports two anchor modes. Under the *self* anchor, agent A_i anchors on itself. Under the *genesis* anchor, A_i uses a genesis node as its anchor. In both cases we define the trusted set for agent A_i, denoted $\mathcal{T}_i$, recursively as:

$$\mathcal{T}_i^{(0)} = \{A_i^{\mathrm{anchor}}\}, \qquad \mathcal{T}_i^{(k)} = \mathcal{T}_i^{(k-1)} \cup \big\{A_x : \exists\, A_y \in \mathcal{T}_i^{(k-1)} \text{ with valid } \mathrm{cert}_{y\to x}\big\}, \tag{2}$$

and let $\mathcal{T}_i = \mathcal{T}_i^{(n)}$ for a configurable depth $n \in \mathbb{N} \cup \{\infty\}$. Each agent chooses its anchor mode and depth locally, tailoring trust to its own requirements.

2.4 Flexible Orchestration

Team assembly. Solving multi-agent tasks efficiently requires quickly determining which agents will contribute to a task – a process we call *team assembly*. A task is submitted to an agent designated the *entrypoint*, which assembles a team based on a set of required capabilities $\mathcal{C}$. This set is either provided by the user alongside the task or initialized dynamically by the entrypoint upon receiving the request, consulting its local registry to determine which capabilities are available. Once a team is set, an overlay network is initialized over the members corresponding to the specified communication pattern, allowing for fast, reliable communication between the team members. This approach is distinct from DyLAN (Liu et al., 2024), which relies on dense inter-agent communication that is intractable at scale. We formalize the assembly protocol in Appendix A.1.

Planning and execution. **PANDA** supports multiple execution topologies within a team, and we implement three that span the design space of state-of-the-art MAS: star, chain, and mesh. Each is equipped with a planning strategy that produces a plan $\mathcal{P}$ as a directed acyclic graph (DAG) of subtasks (Besta et al., 2024), where each vertex is bound to a required capability and an agent drawn from the assembled team. In the *star* approach, an orchestrator creates the plan and dispatches dependency-free subtasks in parallel, synthesizing intermediate results as subtasks complete. In the *chain* approach, agents form a sequential chain in which each communicates only with its predecessor and successor. A random initiator produces a linear plan and hands it to the first agent, each subsequent agent executes its subtask and forwards the plan onward. In the *mesh* approach, a subset of *collaborators* covering every capability jointly agrees on $\mathcal{P}$ via a propose–rank–revise loop: each collaborator proposes a plan, all score every proposal via ranked-choice voting, the lowest-scoring proposals are pruned, and surviving authors revise; the loop terminates when a single plan remains (Lee et al., 2026). Once a consensus plan has been reached, subtasks with redundant capabilities produce their output via multi-agent debate (Du et al., 2024) and completed outputs are broadcast to the team. A supermajority vote after each subtask can trigger replanning under the same propose–rank–revise loop. We depict each topology in Figure 2 and present more details in Appendix B.

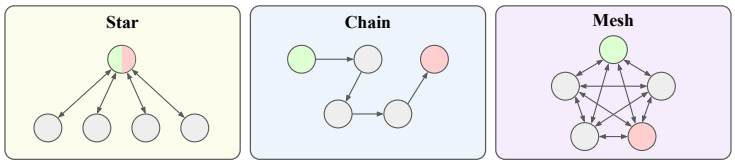


Figure 2: An example of each topology. Green indicates where input is passed to the team, and red denotes who provides the output. In the star, I/O flows through the orchestrator. In the chain, input enters the first agent and exits the last. In the mesh, I/O can begin and end at any teammate.

2.5 FAULT TOLERANT ORCHESTRATION

PANDA provides two types of mechanisms for dealing with faults across the lifecycle of a task: *proactive mechanisms* in the form of provisioning backup agents and redundant execution, and *reactive mechanisms* in the form of failure detection and replanning. **PANDA** has dedicated detection mechanisms for both infrastructure and orchestration failures. Infrastructure failures cover faults external to the agents, such as crashes and network partitions. Orchestration failures cover faults in how the agents themselves plan, organize, and execute.

Proactive mechanisms: Agent provisioning. **PANDA** provisions for redundancy to mitigate failures. The team assembly protocol accommodates fault-tolerance by proactively recruiting backup agents for each capability needed to solve the task. Each capability $c \in \mathcal{C}$ can be annotated with a redundancy $r(c) \in \mathbb{Z}^+$ that denotes how many agents with capability c to recruit.

Proactive mechanisms: Redundant execution. All orchestration strategies support a redundancy extension: a subtask v is dispatched to $n \leq r(\text{cap}(v))$ agents in parallel, and the first successful response is accepted and the rest are canceled, tolerating up to $n - 1$ failures without needing to recruit a new agent.

Reactive mechanisms: Infrastructure-level failures. To detect infrastructure-level failures, we adopt the scalable failure-detector model of Gupta et al. (2001). Each agent periodically pings another and if no response arrives, it asks several other agents to ping the suspected agent and relay the results. If all such pings fail, the agent is declared dead after a timeout and an eviction message is sent to the collective using the collective communication service. False positives are possible; however, if an agent learns of its own removal, it simply rebroadcasts itself, canceling the eviction. The expected time to detect (Δ_{det}) a failure is probabilistic, given by $\Delta_{\text{det}} = T \cdot \frac{e^q}{e^q - 1}$ where T is the protocol period and q is one minus the per-channel message loss probability. Because we probe over TCP, delivery is reliable ($q = 1$), and the expression simplifies to $T \cdot \frac{e}{e-1}$. We can tune T to balance detection speed and network traffic.

Agents are also proactive about confirming they are still considered alive. If an agent receives no messages within a specified timeout, it pings k peers to ask whether they still consider it alive. If they respond negatively, the agent rebroadcasts its capabilities to the collective; if it was a false alarm, the agent resets its timeout and continues.

Reactive mechanisms: Orchestration failures and replanning. Structuring plans as DAGs of typed subtasks enables efficient recovery by replacing the agent assigned to a failed vertex rather than restarting the entire task. When an agent fails, the system invokes the swap protocol that assigns the affected subtask to another compatible agent already in the team, when redundancy permits, or recruits a new agent otherwise. We formalize the swap protocol in Appendix A.2.

For the star orchestration, worker failures are detected and handled by the orchestrator. After each subtask, the orchestrator may revise the plan – holding completed subtasks fixed – to account for any failures encountered, invoking the swap protocol if necessary. Orchestrator failure can be monitored by the entrypoint, and improved by checkpointing the plan and its execution state to a peer for warm restarts if an orchestrator crashes.

For the chain orchestration, once agent A_i forwards to its successor A_{i+1}, it sends `DONE` to its predecessor A_{i-1}; if A_{i-1} detects A_i failure before receiving `DONE`, it invokes the swap protocol and reissues the task to A'_i. Reissues carry a monotonic generation number, so duplicate chains caused by agents that continues after being swapped are killed when detected. Replanning is also local: after an agent receives a subtask, it may rewrite the remaining plan before executing.

For the mesh orchestration, planning-time failures drop the affected collaborator from contention or trigger a swap; execution-time failures either drop the failed agent from an ongoing debate or trigger swap when the capability has no redundancy.

3 EXPERIMENTS

We evaluate **PANDA** across three properties: its ability to scale to large collectives under heavy workloads (Section 3.1), its resilience to infrastructure and orchestration failures (Section 3.2), and the effectiveness of its decentralized governance in establishing trust across the collective (Section 3.3).

Baselines. We compare **PANDA** against Magentic-One, which uses a star topology; AgentNet, which uses a chain; and Internet of Agents (IoA), which uses a recursive-star (tree) topology.

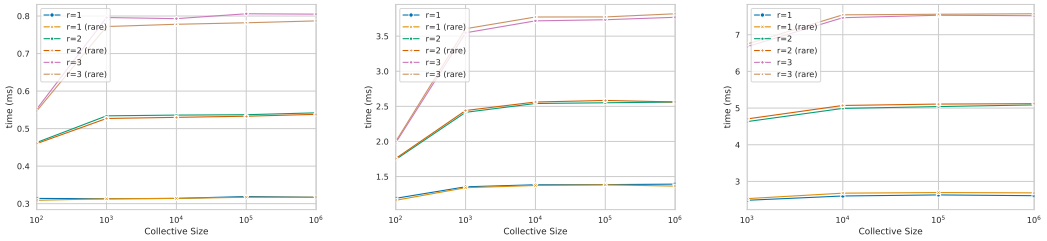
Benchmark. We evaluate on the HotPotQA benchmark in the fullwiki setting, a Wikipedia-based question-answering dataset in which each question requires reasoning across two pages. Questions take one of two forms: *bridge*, where information from one page must be gathered before the final answer can be located on a second page, and *comparison*, where information from two pages must be contrasted. To ensure that agents work from the dataset's original snapshot rather than more recent content, they retrieve pages through an MCP server that hosts the Wikipedia version used to curate the benchmark. Agents are explicitly instructed to answer only from evidence returned by the MCP server, and empirically when the server is offline they correctly fail rather than hallucinate.

Configuration. Agents use `GPT-5.4-mini` as the underlying LLM. Each system runs with 10 worker agents under two configurations: *complete knowledge*, where all agents access the full Wikipedia snapshot, and *partial knowledge*, where they are split evenly across two disjoint halves.

Metrics. We report four metrics: exact match (EM), F1, completion, and runtime. EM and F1 follow Yang et al. (2018): EM is a post-normalization exact match between the expected and provided answers, and F1 is a per-word F1 giving partial credit for partially correct answers. Completion measures whether any answer was produced, regardless of correctness. We report runtime (RT) in several forms, all in wall-clock seconds: *total* is the time for all tasks to complete, while *completed*, *failed*, *correct*, and *incorrect* are the average per-task times over tasks in each respective category.

3.1 PANDA SCALABILITY

Team Assembly. To show that **PANDA** generalizes to thousands of agents, we measure the cost of team assembly as a function of collective size. Once assembly completes and the team overlay is established, subsequent work is confined to the team and does not scale with the collective. We construct a collective of n fully connected peers, each advertising 5 capabilities sampled uniformly from a pool of 100. Figure 3 reports median time to complete a team assembly request across $c = 5, 25, 50$ requested capabilities with redundancies $r = 1, 2, 3$. For each configuration we include a *rare* variant in which one of the required capability has only five providers in the collective.



(a) $c = 5$ (b) $c = 25$ (c) $c = 50$

Figure 3: Team assembly times for number of capabilities $c = 5$, 25, and 50 across redundancies $r = 1$, 2, 3. Rare refers to when only 5 agents with a given needed capability exist. At $c = 50$, 100 agents are not sufficient for staffing the needed capabilities.

We observe several key patterns. First, past a critical collective size – at which sufficient redundancy across all capabilities is reached – assembly time is constant with respect to the collective size. The initial rise reflects a smaller candidate pool, which lets assembly commit quickly; however, once coverage is sufficient, our load balancer probes a capped set of candidates regardless of n, giving constant scaling thereafter. The *rare* variant is slightly faster for the same reason: scarce capabilities constrain the candidate set earlier. Assembly time does scale with team size, but remains on the order of milliseconds, negligible relative to the runtimes in Table 2. Finally, team assembly incurs a fixed network overhead equal to the maximum RTT between the assembling agent and its probed candidates, which varies with agent co-location but is likewise independent of collective size.

Load Balancing and Scheduling. We run 200 randomly selected tasks at a concurrency of five, yielding a 1:1 ratio of tasks to workers per task to simulate a loaded collective. We configure **PANDA**

with a probe ratio $d = 2$ and a FIFO scheduling policy. To make the comparison fair with respect to concurrency, each baseline is fronted with a server that accepts and distributes concurrent requests across its underlying architecture. We provide results in Table 2.

Table 2: Performance on HotPotQA benchmark without failures.

	Complete Knowledge			Partial Knowledge		
System	EM (%)	F1 (%)	Total RT (s)	EM (%)	F1 (%)	Total RT (s)
Magentic-One	42.5	48.52	851.1	25.5	29.99	1181.7
AgentNet	67.5	77.33	840.4	43.5	51.37	1588.4
Internet of Agents	**72.5**	**81.26**	2182.1	**62.0**	**68.07**	3949.1
PANDA-STAR	66.5	74.11	<u>464.5</u>	45.5	54.60	<u>594.9</u>
PANDA-CHAIN	64.5	75.54	**462.5**	47.0	53.18	**452.9**
PANDA-MESH	<u>70.0</u>	<u>79.39</u>	714.0	<u>56.0</u>	<u>65.90</u>	1038.4

PANDA significantly speeds up performance with respect to our baselines. **PANDA-STAR** and **PANDA-CHAIN** are roughly $2\times$ faster than Magentic-One and AgentNet improving EM and F1 over Magentic-One and comparable with AgentNet. IoA records higher accuracy than **PANDA-STAR** and **PANDA-CHAIN** but incurs 5-$8\times$ runtime. **PANDA-MESH** is faster than Magentic-One and AgentNet and achieves better accuracy metrics especially in the partial knowledge case. **PANDA-MESH** essentially matches IoA in terms of EM and F1 but with a 3-$4\times$ faster runtime. Overall we demonstrate that **PANDA** can handle a high throughput of tasks while maintaining state-of-the-art EM and F1.

3.2 Task Completion with Failures

Infrastructure Failures. For each task, we ensure that at least one agent crashes during execution. Since the number of steps a task will take is not known, we set a high per-step crash probability of $p = 0.5$, with a backstop guaranteeing at least one crash per task. We present the results in Table 3.

Table 3: Performance on HotPotQA under infrastructure failures.

	Complete Knowledge				Partial Knowledge			
System	Completion (%)	F1 (%)	Comp. RT (s)	Failed RT (s)	Completion (%)	F1 (%)	Comp. RT (s)	Failed RT (s)
Magentic-One	0.00	0.00	–	<u>8.14</u>	0.00	0.00	–	**8.92**
AgentNet	0.00	0.00	–	**5.08**	0.00	0.00	–	<u>15.91</u>
Internet of Agents	0.00	0.00	–	100.63	0.00	0.00	–	122.63
PANDA-STAR	**100.0**	**72.33**	<u>28.04</u>	–	**100.0**	**68.10**	39.42	–
PANDA-CHAIN	**100.0**	68.23	**27.29**	–	**100.0**	<u>60.43</u>	<u>32.23</u>	–
PANDA-MESH	**100.0**	<u>71.80</u>	61.57	–	**100.0**	57.13	**31.87**	–

None of the baselines complete any task under agent crashes. In contrast, all three **PANDA** topologies sustain high accuracy and reasonable per-task runtimes. We note that Magentic-One and AgentNet fail quickly, whereas IoA has no mechanism to detect a crash until a subtask times out, inflating its failure latency. We define a failed task as a task that did not complete.

Orchestration Failures. For each task, one agent is faulty at the start of execution, always returning an unrelated search result. In AgentNet, IoA, and **PANDA**, healthy replacements exist in the registry and can be discovered; Magentic-One has no mechanism to swap a faulty agent and is stuck with the misconfigured one. A task is *correct* if F1 > 0, and *incorrect* if F1 $= 0$. We present the results in Table 4.

Under faults, **PANDA** outperforms the baselines across all metrics. Its dedicated swap protocol replaces the faulty agent with a healthy one, allowing the task to complete correctly and efficiently. Complete knowledge provides redundancy that improves EM and F1 for Magentic-One and IoA; however, for AgentNet, the added confidence in capability causes the faulty agent to return incorrect answers with higher probability.

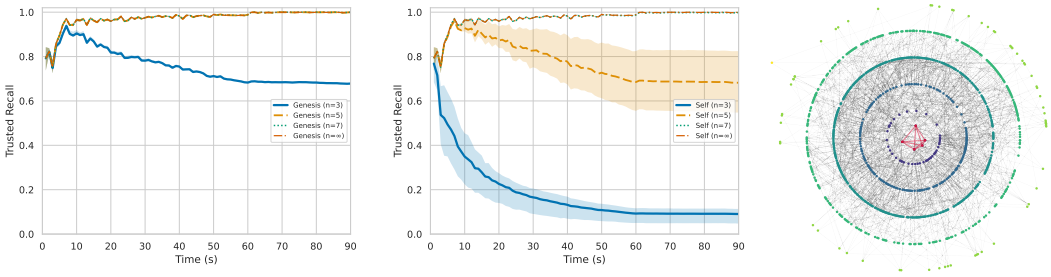
Table 4: Performance on HotPotQA under orchestration failures.

	Complete Knowledge				Partial Knowledge			
System	EM (%)	F1 (%)	Correct RT (s)	Incorrect RT (s)	EM (%)	F1 (%)	Correct RT (s)	Incorrect RT (s)
Magentic-One	45.00	52.31	28.24	59.58	6.50	6.83	18.01	40.07
AgentNet	1.00	1.56	47.90	24.57	22.50	27.51	72.90	50.59
Internet of Agents	24.50	27.34	131.40	192.78	11.00	12.51	57.87	278.40
PANDA-STAR	64.00	71.19	**13.93**	23.69	49.50	59.05	**15.91**	29.42
PANDA-CHAIN	58.50	69.33	19.69	**23.66**	53.50	62.89	22.73	25.32
PANDA-MESH	**69.50**	**78.86**	15.73	26.59	**54.00**	**66.14**	22.70	**24.51**

3.3 PANDA GOVERNANCE

We study how trust propagates in **PANDA**. Our experiment proceeds in two stages. In the *join* stage (60 s), 1000 agents enter the collective, each collecting $t = 2$ certificates from existing agents that verify and vouch for its identity. Joins are uniformly distributed across the stage, and certifiers are drawn uniformly at random from the agents active in the collective. In the *churn* stage (30 s), agents join and leave the collective, each at a rate of one per second following a Poisson process.

We evaluate the *self* and *genesis* anchors at depths $n = 3, 5, 7, \infty$ and measure *trusted recall*: the fraction of alive agents that each agent trusts. We visualize the underlying certificate graph, in which each node is an agent and each edge (A_i, A_j) indicates that A_i and A_j have mutually signed each other's certificates. Results are shown in Figure 4; (see Appendix D for additional results). All configurations achieve trusted precision, the fraction of trusted peers that are actually alive, $> 99\%$.



(a) Genesis Anchor ($t = 2$) (b) Self Anchor ($t = 2$) (c) Certificate Graph ($t = 2$)

Figure 4: Average trusted recall in a collective where each agent requires two certificates to join with five genesis agents. Shaded regions show the IQR of trusted recall across agents.

Under the genesis anchor, coverage is strong: even at $n = 3$, trusted recall reaches nearly 70%. Genesis anchors have essentially no IQR, since all genesis nodes are mutually connected. Self anchoring is weaker at lower n but still recovers a trusted set of nearly 100 agents at $n = 3$. Overall, trusted set size remains stable under churn, and even restrictive trust schemes build substantial trusted sets.

4 CONCLUSION

In this work, we presented **PANDA**, a decentralized architecture for large-scale LLM-based multi-agent systems, addressing four requirements that existing architectures satisfy either partially or not at all: scalability, fault tolerance, flexible orchestration, and decentralized governance. **PANDA** distributes agents and tasks across the infrastructure, preserves task execution despite orchestration and infrastructure failures, and supports multiple planning and execution models. Its web-of-trust governance model enables agents from different administrative domains to establish trust without relying on a single central authority. Our evaluation demonstrates that **PANDA** scales efficiency while maintaining task utility, recovering from failures, and supporting diverse orchestration patterns, providing a foundation for building open, resilient, and governable multi-agent ecosystems.

AI USE STATEMENT

In this work, we used generative AI tools to help implement methods and formulate mathematical claims. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, clean and reformat datasets, or interpret results. Generating synthetic data sets, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, assisting with translation, and supporting qualitative and thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to create or edit software code, summarize or analyze existing literature, discover research topics or identify gaps, brainstorm, edit a research paper to improve readability, identify relevant literature, and propose a title or keywords for a research paper. We have reviewed all AI-assisted work. We manually reviewed all LLM-generated code and mathematical claims. All papers cited were manually verified and read by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Our code, along with how to download the data we used, is available at `https://anonymous.4open.science/r/panda-anon/`. Section 2 describes our system in detail, and Appendix B precisely describes each orchestration topology. The HotPotQA dataset is publically available on HuggingFace at `https://huggingface.co/datasets/hotpotqa/hotpot_qa`.

REFERENCES

Sepideh Avizeh, Tushin Mallick, Alina Oprea, Cristina Nita-Rotaru, and Reihaneh Safavi-Naini. MAGIQ: A post-quantum multi-agentic AI governance system with provable security. *arXiv preprint arXiv:2605.06933*, 2026.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, J Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Thirty-Eight AAAI Conference on Artificial Intelligence, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, Fourteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI OOK-Press, 2024.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026.

Weize Chen, Ziming You, Ran Li, Chen Qian, Chenyang Zhao, Cheng Yang, Ruobing Xie, Zhiyuan Liu, Maosong Sun, et al. Internet of agents: Weaving a web of heterogeneous agents for collaborative intelligence. In *International Conference on Learning Representations*, volume 2025, pp. 36374–36411, 2025.

Debian Project. Debian new members process: Identification. `https://www.debian.org/devel/join/nm-step1`. Accessed: 2026-09-24.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 11733–11763, 2024.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, et al. Magentic-one: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*, 2024.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. Agentscope: A flexible yet robust multi-agent platform, 2024. URL `https://arxiv.org/abs/2402.14034`.

Sandro Rodriguez Garzon, Awid Vaziry, Enis Mert Kuzu, Dennis Enrique Gehrmann, Buse Varkan, Alexander Gaballa, and Axel Küpper. Ai agents with decentralized identifiers and verifiable credentials. *arXiv preprint arXiv:2511.02841*, 2025.

Indranil Gupta, Tushar D Chandra, and Germán S Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pp. 170–179, 2001.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Ziyu Ye, Bowei Xia, Tao Sun, Zhaoxuan Jin, Yingru Li, Zeyu Zhang, et al. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. *Advances in Neural Information Processing Systems*, 38:50859–50906, 2026.

Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, pp. 85–103. Springer, 1972.

Matthew D Laws, Alina Oprea, and Cristina Nita-Rotaru. Attacks and mitigations for distributed governance of agentic ai under byzantine adversaries. *arXiv preprint arXiv:2605.12364*, 2026.

Haejoon Lee, Vincent-Daniel Yun, Hyeonho Oh, Dimitra Panagou, and Sai Praneeth Karimireddy. Robust multi-agent llms under byzantine faults. *arXiv preprint arXiv:2605.09076*, 2026.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic llm-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for llm agents as general programs. *CoRR*, abs/2502.13965, February 2025. URL https://doi.org/10.48550/arXiv.2502.13965.

Yosi Mass and Onn Shehory. Distributed trust in open multi-agent systems. In Rino Falcone, Munindar Singh, and Yao-Hua Tan (eds.), *Trust in Cyber-societies*, pp. 159–174, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45547-9.

Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE transactions on parallel and distributed systems*, 12(10):1094–1104, 2002.

Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 69–84, 2013.

Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

Ramesh Raskar, Pradyumna Chari, John Zinky, Mahesh Lambe, Jared James Grogan, Sichao Wang, Rajesh Ranjan, Rekha Singhal, Shailja Gupta, Robert Lincourt, et al. Beyond dns: Unlocking the internet of ai agents via the nanda index and verified agentfacts. *arXiv preprint arXiv:2507.14263*, 2025.

Ronald L Rivest and Butler Lampson. Sdsi-a simple distributed security infrastructure. 1996.

Rana Shahout, Cong Liang, Shiji Xin, Qianru Lao, Yong Cui, Minlan Yu, and Michael Mitzenmacher. Fast inference for augmented large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=uNqTxj5brQ.

Georgios Syros, Anshuman Suri, Jacob Ginesin, Cristina Nita-Rotaru, and Alina Oprea. SAGA: A security architecture for governing ai agentic systems. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS, 2026.

Jen tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Michael Lyu, and Maarten Sap. On the resilience of LLM-based multi-agent collaboration with faulty agents. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=bkiM54QftZ.

Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks. *arXiv preprint arXiv:2007.02754*, 2020.

Ji Wang, Kashing Chen, Xinyuan Song, Ke Zhang, Lynn Ai, Eric Yang, and Bill Shi. Symphony: A decentralized multi-agent framework for scalable collective intelligence. *arXiv preprint arXiv:2508.20019*, 2025.

Zile Wang, Qianli Liu, Kaibin Guo, Haodong Wang, Jian Lin, Zicong Hong, and Song Guo. Ppai: Enabling personalized llm agent interoperability for collaborative edge intelligence. In *IEEE INFOCOM 2026-IEEE Conference on Computer Communications*, pp. 1–10. IEEE, 2026.

Yingxuan Yang, Huacan Chai, Shuai Shao, Yuanyi Song, Siyuan Qi, Renting Rui, and Weinan Zhang. Agentnet: Decentralized evolutionary coordination for llm-based multi-agent systems. *Advances in Neural Information Processing Systems*, 38:107309–107336, 2026.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

Geunbin Yu. Adaptorch: Task-adaptive multi-agent orchestration in the era of llm performance convergence. *arXiv preprint arXiv:2602.16873*, 2026.

Sukwon Yun, Jie Peng, Pingzhi Li, Wendong Fan, Jie Chen, James Y Zou, Guohao Li, and Tianlong Chen. Graph-of-agents: A graph-based framework for multi-agent llm collaboration. In *International Conference on Learning Representations*, volume 2026, pp. 19745–19760, 2026.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for llm-based multi-agent systems. In *International Conference on Learning Representations*, volume 2025, pp. 75389–75428, 2025a.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=LpE54NUnmO.

Shengli Zhang, Deen Ma, Zibin Lin, and Taotao Wang. Distributed general-purpose agent networks: Architecture, key mechanisms, and prototypes. *arXiv preprint arXiv:2606.17368*, 2026.

Lifan Zheng, Jiawei Chen, Qinghong Yin, Jingyuan Zhang, Xinyi Zeng, and Yu Tian. Rethinking the reliability of multi-agent systems: a perspective from byzantine fault tolerance. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence and Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence and Sixteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'26/IAAI'26/EAAI'26. AAAI Press, 2026. ISBN 978-1-57735-906-7. doi: 10.1609/aaai.v40i41.40806. URL https://doi.org/10.1609/aaai.v40i41.40806.

Siyuan Zhuang, Stephanie Wang, Eric Liang, Yi Cheng, and Ion Stoica. {ExoFlow}: A universal workflow system for {Exactly-Once}{DAGs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 269–286, 2023.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=uTC9AFXIhg.

Philip R Zimmermann. *The official PGP user's guide*. MIT press, 1995.

A Team Assembly and Repair

Team formation and maintenance are fundamental components of **PANDA**. In this appendix we formalize initial team assembly and team repair via the `swap` protocol.

A.1 Assembly Protocol

Determining whether an agent can fulfill a capability relies on a matching primitive $\texttt{match}(c, a) \in \{0, 1\}$, which can be realized in various ways, including direct string comparison or thresholded semantic scoring. Given a set of capabilities $\mathcal{C}$, a redundancy $r(c) \in \mathbb{Z}^+$ and a trusted set of agents $\mathcal{A}$, we construct a matching $\phi : \mathcal{C} \to 2^{\mathcal{A}}$ of agents to capabilities, assigning each capability c a set $\phi(c) \subseteq \mathcal{A}$ where $|\phi(c)| = r(c)$ agents. We require ϕ to be *valid*: every assigned agent must possess the corresponding capability, i.e., $\forall a \in \phi(c) : \texttt{match}(c, a) = 1$. Among valid matchings, we consider two strategies. A *dedicated* matching assigns each capability its own distinct pool, with no agent shared across capabilities:

$$\phi(c_i) \cap \phi(c_j) = \emptyset \quad \forall\, c_i \neq c_j \in \mathcal{C}. \tag{3}$$

This maximizes parallelism at the cost of recruiting $\sum_{c \in \mathcal{C}} r(c)$ agents. A *consolidated* matching instead minimizes the total number of distinct agents recruited:

$$\min_{\phi} \Big| \bigcup_{c \in \mathcal{C}} \phi(c) \Big|. \tag{4}$$

Computing a minimal consolidated matching is an instance of set cover, a known NP-hard problem (Karp, 1972); we therefore opt for an approximate greedy solution instead.

The assembly procedure proceeds as follows. The entrypoint, the agent to which the request was sent, first consults its local registry. If it can assemble a valid matching ϕ from registered agents, team assembly completes locally. Otherwise, the entrypoint broadcasts a request for the unmatched capabilities, and any trusted agent able to fulfill one may respond and join the team.

A.2 Swap Protocol

When an agent a_i that serves some capability c crashes or is declared failed, we must remove it from the team and replace its capabilities. We refer to this as the `swap` protocol, denoted $\texttt{swap}(a_i, c)$. If, for some capability c, sufficient redundancy $r(c)$ remains such that $|\phi(c) \setminus \{a_i\}| > 0$, we can rely on the other agents $a_j \in \phi(c) \setminus \{a_i\}$ covering that capability: any subtasks previously assigned to a_i are reassigned to some a_j, and a_i is evicted from the team: $\phi(c) \leftarrow \phi(c) \setminus \{a_i\}$ so no further subtasks are assigned to it.

If $|\phi(c)|$ ever reaches zero, a replacement must be found. Similar to team assembly, the agent that detected the fault first searches its registry; if successful, it adds the found agent a_k to the team updating $\phi(c) \leftarrow \{a_k\}$. Otherwise, it broadcasts the request and awaits a response, updating $\phi(c)$ when one arrives.

B Fault-Tolerant Orchestration Details

PANDA's architecture naturally accommodates a variety of execution patterns. In this appendix, we explore in detail three general execution topologies into which **PANDA** teams can be shaped, and under which the majority of state-of-the-art multi-agent systems fall. For each topology, we design a planning strategy that produces a structured plan, decomposing a task into a graph of subtasks, each corresponding to a single capability. We then extend each design to be fault-tolerant, where the step-based plan facilitates efficient recovery.

Table 5: Summary of notation.

Task & Team	
$\mathcal{T}$	User task
$\mathcal{C}$	Required capabilities
$\mathcal{A}'$	Team of agents
$r(c)$	Redundancy: agents recruited per capability
$\phi(c)$	Pool of $r(c)$ agents matched to capability c
Plan	
$\mathcal{P}$	Plan (DAG of subtasks)
$\mathcal{V}, \mathcal{E}$	Subtasks and dependencies
$\mathrm{cap}(v)$	Capability required by subtask v
$\delta(v)$	Agent assigned to subtask v
State & Recovery	
$\mathcal{S}$	Global task state $(\mathcal{P}, \mathcal{D}, \mathcal{O}, \mathcal{F})$
$\mathcal{D}$	Completed subtasks
$\mathcal{O}$	Outputs of completed subtasks
$\mathcal{F}$	In-flight subtasks

B.1 Problem Formulation

In each topology, the goal is to solve a user task $\mathcal{T}$ given the tuple $(\mathcal{T}, \mathcal{C}, r, \mathcal{A}', \phi)$, where $\mathcal{C}$ is the set of required capabilities, $r : \mathcal{C} \to \mathbb{Z}^+$ is the redundancy mapping, $\mathcal{A}'$ is the team of agents, and $\phi : \mathcal{C} \to 2^{\mathcal{A}'}$ is the matching assigning each capability c to a pool of $|\phi(c)| = r(c)$ capable agents (Appendix A.1). To solve $\mathcal{T}$, we first generate a plan $\mathcal{P}$, represented as a directed acyclic graph (DAG) of subtasks, adapting graph-structured reasoning to a multi-agent setting (Besta et al., 2024). The vertices $\mathcal{V}$ represent subtasks and the edges $\mathcal{E}$ represent dependencies between them. Each vertex $v_i \in \mathcal{V}$ is assigned a capability $\mathrm{cap}(v_i) \in \mathcal{C}$ and a capable agent $\delta(v_i) \in \phi(\mathrm{cap}(v_i))$:

$$\mathrm{cap} : \mathcal{V} \to \mathcal{C}, \qquad \delta : \mathcal{V} \to \mathcal{A}', \qquad \delta(v_i) \in \phi(\mathrm{cap}(v_i)). \tag{5}$$

Once the plan is generated, execution proceeds according to the specified topology. Across all topologies, if an agent fails, we invoke the `swap`(a, c), where a crashed agent a is replaced by another agent a' that fulfills the same capability c. If redundancy permits ($r(c) > 1$), a replacement is available immediately; otherwise, a new agent is added dynamically by searching for c via the `swap` protocol (Appendix A.2).

B.2 Star Topology

Design. The star topology is common in multi-agent systems, where all planning is handled by an orchestrator at the center of the star (Fourney et al., 2024; Hu et al., 2026). In **PANDA-STAR**, the orchestrator receives the task, $\mathcal{T}$, and generates the plan, $\mathcal{P}$. It begins by executing all dependency-free subtasks in parallel. As dependencies are resolved, the orchestrator optionally synthesizes intermediate results and dispatches each newly enabled subtask to its assigned agent, along with the relevant context from completed dependencies. The orchestrator maintains the global task state $\mathcal{S} = (\mathcal{P}, \mathcal{D}, \mathcal{O}, \mathcal{F})$, where $\mathcal{P}$ is the plan, $\mathcal{D} \subseteq \mathcal{V}$ is the set of completed subtasks, $\mathcal{O}$ are the saved outputs, and $\mathcal{F} \subseteq \mathcal{V}$ are the in-flight subtasks. Once all subtasks are fulfilled, the result is returned to the user.

Replanning. In order to remain tolerant to failures, the orchestrator can replan. After each subtask completes, the orchestrator considers the new output together with the current state $\mathcal{S}$ and decides whether to replan. If so, it generates a revised plan $\mathcal{P}'$ while holding the already-completed subtasks $\mathcal{D}$ fixed. It then reconciles the in-flight subtasks $\mathcal{F}$ – any no longer required under $\mathcal{P}'$ are canceled, while the rest continue running.

Fault Tolerance. There are two components of the star topology that can fail, the orchestrator or the workers. Worker infrastructure failures are straightforward: the orchestrator uses **PANDA**'s

failure detector to detect a crashed agent $a \in \phi(c)$ and swap it with another agent a' using the `swap` protocol. If the crashed agent was actively executing a subtask v_i ($\delta(v_i) = a$), the orchestrator reissues it, updating $\delta(v_i) = a'$; otherwise, execution proceeds as normal with a fully restored team. Orchestration failures, such as repeated bad responses, are reasoned about after each subtask completes. An orchestrator can elect to invoke the swap protocol for an agent that repeatedly fails to give a reasonable answer.

We next consider failure of the orchestrator itself. The simplest response is to restart the task from scratch, however, this discards all completed work. Instead, the orchestrator, a_{orch}, can checkpoint its execution state $\mathcal{S}$ to one or more peers as it progresses, so that upon failure a replacement orchestrator can resume from the latest checkpoint rather than from the beginning. The entrypoint or a neighboring orchestrator are natural candidates for holding these checkpoints. Whichever peer holds the checkpoint detects the orchestrator's failure and restarts the task from the checkpointed state at a new orchestrator, $o' = \texttt{swap}(a_{\text{orch}}, \text{“orchestration”})$.

Redundant Execution. For even stronger fault tolerance, at the cost of additional computation, a subtask v can be executed redundantly whenever its capability pool contains spare agents ($r(\text{cap}(v)) > 1$). Rather than assigning v to a single agent, we assign it to $n \leq r(\text{cap}(v))$ agents drawn from $\phi(\text{cap}(v))$, accept the first response, and cancel the remaining executions. Because the subtask completes as soon as any one agent succeeds, it tolerates up to $n - 1$ failures without triggering a swap or incurring recovery latency.

B.3 Chain Topology

Design. The chain topology consists of agents that communicate without a centralized point of control. Agents are arranged in a dynamic chain in which each agent communicates only with its immediate predecessor and successor. **PANDA** applies dynamic, decentralized planning to this topology. A randomly chosen agent a_0 first generates a preliminary plan $\mathcal{P}$. Without a central point of control, coordinating concurrent subtasks quickly becomes intractable, so $\mathcal{P}$ is restricted to a sequential plan $v_1, \ldots, v_n$ – which our DAG formulation handles gracefully, as a sequence is trivially a DAG. a_0 then forwards $\mathcal{P}$, along with the task, to $\delta(v_1)$, the agent assigned to its first subtask. This repeats down the chain: each agent $a = \delta(v_i)$ fulfills its subtask v_i and forwards the plan to $\delta(v_{i+1})$. Once all subtasks are complete, the final agent returns the result.

Replanning. Upon receiving a subtask, an agent $a = \delta(v_i)$ for the pending subtask v_i evaluates it. If unsatisfactory, a may trigger a *replan*, producing a revision $\mathcal{P}' = v'_i, \ldots, v'_m$ in place of the remaining plan. If the new pending subtask is its own ($\delta(v'_i) = a$), a executes v'_i and forwards the plan to $\delta(v'_{i+1})$; otherwise, a forwards the revised plan to $\delta(v'_i)$ to fulfill. The number of replans is capped to guarantee termination. This decentralized replanning lets agents progressively rewrite the plan to better suit the team's capabilities.

Fault Tolerance. Consider three consecutive agents in the chain, a_{i-1}, a_i, a_{i+1}, assigned to subtasks v_{i-1}, v_i, v_{i+1} respectively. When a_{i-1} completes v_{i-1}, it forwards the plan and its output to a_i but retains a copy of what it sent. Orchestration failures, such as misconfigured agents can be detected and swapped during the replanning step. The challenge is detecting and recovering from infrastructure failures of a_i without global coordination.

We introduce a `DONE` acknowledgment for this purpose. Once a_i has fulfilled v_i and forwarded its result to a_{i+1}, it sends `DONE` back to a_{i-1} in the background, signaling that responsibility has passed downstream. If a_{i-1} declares a_i failed (via **PANDA**'s failure detector) before receiving this `DONE`, it invokes $\texttt{swap}(a_i, \text{cap}(v_i))$ to obtain a replacement a'_i and reissues v_i – using its retained copy of the handoff state – to a'_i. Because each agent retains its handoff until acknowledged, recovery requires only local state at the immediate predecessor.

A duplicate chain can arise in an edge case: a_i forwards to a_{i+1} but fails before sending `DONE` to a_{i-1}, causing a_{i-1} to reissue v_i even though the work has already progressed downstream. To prevent this, each reissue carries a unique, monotonically increasing *generation* number. Every downstream agent records the highest generation it has seen and terminates any message belonging to a lower generation, so the stale chain is killed while the reissued one proceeds.

Redundant Execution. As with the star, we can trade computational cost for increased fault tolerance: rather than dispatching v_i to a single successor, a_{i-1} assigns it to $n \leq r(\mathrm{cap}(v_i))$ agents from $\phi(\mathrm{cap}(v_i))$ and, upon receiving a `DONE` from any one, cancels the rest. The subtask then tolerates up to $n-1$ failures without invoking the `swap` protocol.

B.4 Mesh Topology

Design. The mesh topology consists of a team of agents arranged in a fully connected mesh, each able to communicate with all others. The team's first step is to agree on a plan $\mathcal{P}$, represented as a DAG. To reach agreement efficiently, we first select a subset of collaborators $\mathcal{K} \subseteq \mathcal{A}'$ that covers every required capability:

$$\mathcal{K} = \{\, a_c : a_c \in \phi(c) \,\}_{c \in \mathcal{C}} \tag{6}$$

Restricting planning to $\mathcal{K}$ rather than the full team keeps the process efficient while ensuring all capabilities are represented. Each collaborator $a_i \in \mathcal{K}$ generates a candidate plan $\mathcal{P}_i$. Every plan is broadcast to the other collaborators, who rank them via ranked-choice voting. The resulting scores are broadcast so that every collaborator holds the same set of scores and thus a consistent view of the rankings. Each agent then independently eliminates the bottom j proposals – arriving at the same outcome from the shared scores – and feedback is returned to the authors of the surviving plans, who revise them. This propose–rank–revise cycle repeats until a single consensus plan $\mathcal{P}$ remains.

Once a plan is agreed upon, the team executes it. The plan is first broadcast to the entire team, including redundant agents, and every subtask without dependencies is assigned. Fulfillment of a subtask v proceeds as follows. If multiple agents are assigned to its capability ($|\phi(\mathrm{cap}(v))| > 1$), they jointly produce a result through multi-agent debate (Du et al., 2024): each agent proposes an answer and critiques the others over successive rounds until they converge on an agreed output o_v. Otherwise, a single agent produces o_v directly. When v completes, o_v is broadcast to the team, and any subtask whose dependencies are now satisfied may begin. Each agent maintains the global task state $\mathcal{S} = (\mathcal{P}, \mathcal{D}, \mathcal{O}, \mathcal{F})$. Once all subtasks are fulfilled, the result is returned to the user.

Replanning. After each subtask output is fulfilled, the collaborators $\mathcal{K}$ vote on whether to replan. If a supermajority (more than $2/3$) votes in favor, each agent that voted yes proposes a revised plan $\mathcal{P}'_i$, holding the already-completed subtasks $\mathcal{D}$ fixed. The same propose–rank–revise cycle then runs over these proposals to produce the new plan $\mathcal{P}'$. Once $\mathcal{P}'$ is broadcast to the team, each agent independently cancels any in-flight subtask $v_f \in \mathcal{F}$ no longer required under $\mathcal{P}'$, while the rest continue running.

Fault Tolerance. Nodes can fail at two points: planning or execution. If a collaborator $a_f \in \mathcal{K}$ responsible for capability c_f fails during planning, we perform $\mathtt{swap}(a_f, c_f)$; if a_f's proposal is still in contention, it is removed from the running. This can never eliminate the last remaining plan, since execution begins as soon as a single plan remains. If instead the failed node was a redundant agent (not in $\mathcal{K}$), the failure can be ignored, as it does not affect the team's ability to plan.

If an agent fails during execution, there are two cases. If the failed agent belongs to a redundant pool ($|\phi(c)| > 1$) and is actively participating in the debate for a subtask v with $\mathrm{cap}(v) = c$, the remaining agents in $\phi(c)$ simply drop it from the debate and continue to produce o_v; if that pool is idle at the time of failure, the failure can be ignored. If instead the capability has no redundancy ($|\phi(c)| = 1$), we recover via $\mathtt{swap}(a, c)$.

Redundant Execution. As with the previous two topologies, the mesh topology can execute redundantly. However, because the mesh already uses all available redundancy for multi-agent debate, redundant execution is offered as an alternative rather than an additive extension. Multi-agent debate optimizes for accuracy at the cost of additional rounds of communication; redundant execution instead has all agents in $\phi(\mathrm{cap}(v))$ execute the subtask v in parallel and accepts the first response, cancelling the rest. This tolerates up to $|\phi(\mathrm{cap}(v))| - 1$ failures without triggering a swap or incurring recovery latency, trading debate's accuracy gains for lower latency.

C Formal Verification

We formally specify **PANDA**'s core protocols in TLA$^+$, modeling peer gossip, infrastructure failure detection, failure recovery, self-refutation, team assembly, and the swap protocol. Load-aware candidate selection is abstracted as nondeterministic choice, so safety holds under any selection policy. Using TLC, we verify three properties: every completed task has all required capabilities assigned (safety), no agent believes it belongs to a team it was not assigned to (safety), and every task eventually completes or fails under standard fairness assumptions (liveness).

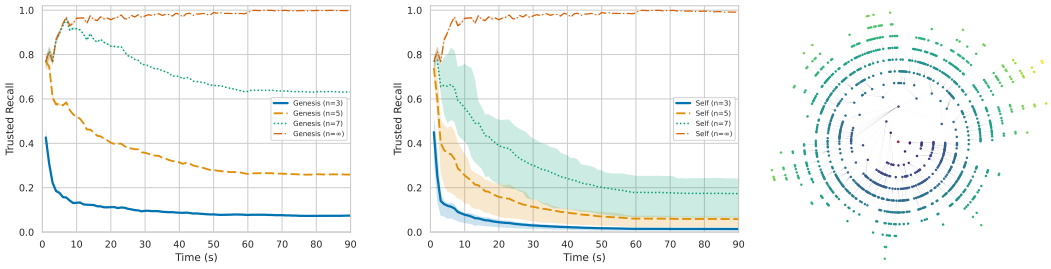
D Additional Governance Experiments

In this appendix we expand our study of how trust propagates in **PANDA**. As in Section 3.3, each experiment proceeds in two stages. In the *join* stage, agents enter the collective, each exchanging t mutual certificates with existing agents that verify its identity. In the *churn* stage, agents join and leave the collective, each at a rate of one per second following a Poisson process.

For each policy, we evaluate the *self* and *genesis* anchors at depths $n \in \{3, 5, 7, \infty\}$, measure *trusted recall*, and visualize the underlying certificate graph (as defined in Section 3.3). All configurations achieve trust precision $> 99\%$.

D.1 Varying the Number of Certifiers

We present results for $t = 1$ and $t = 3$ on a collective size of 1000 in Figures 5 and 6, respectively. At $t = 1$, the reduced certification requirement yields a significantly less connected graph, but coverage remains strong under the genesis anchor: even at $n = 3$, each agent trusts nearly 100 peers. Self anchoring is weaker but scales with depth, achieving near-perfect recall at $n = \infty$ because the trust graph is connected. While fewer connections reduce the trusted set overall, substantial trusted sets can still be accrued when the collective is sufficiently large. At $t = 3$, trusted recall improves relative to $t = 2$ (Section 3.3), though the gain is smaller than the jump from $t = 1$ to $t = 2$, suggesting diminishing returns. Together these results suggest $t = 2$ as a reasonable default – also the choice made by the Debian developers group (Debian Project). In both cases, trusted set size remains stable under churn.

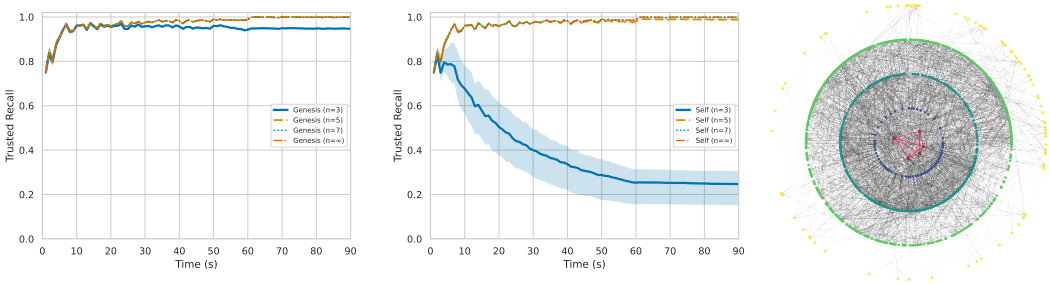


(a) Genesis Anchor ($t = 1$) (b) Self Anchor ($t = 1$) (c) Certificate Graph ($t = 1$)

Figure 5: Average trusted recall in a 1000 agent collective where each agent requires one certificate to join with one genesis agent. Shaded regions show the IQR of trusted recall across agents.

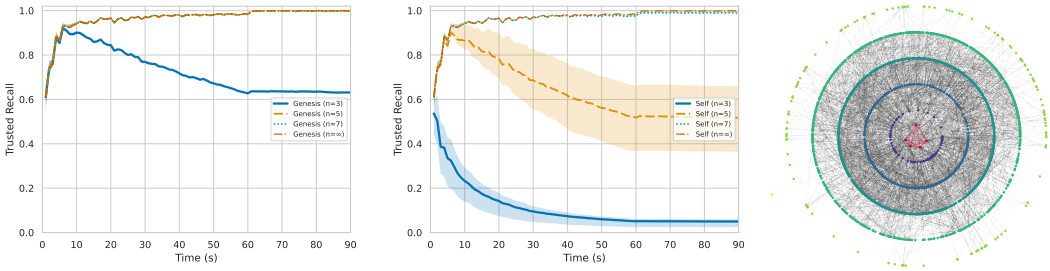
D.2 Larger Collectives

Figure 7 presents the same experiment with $t = 2$ and a collective size of 2000 agents. While trusted recall drops slightly compared to the $t = 2$, $N = 1000$ setting (Section 3.3), the absolute number of trusted agents increases, indicating that trusted sets grow with collective size. However, the drop in recall indicates that trusted set size grows sublinearly with collective size.



(a) Genesis Anchor ($t = 3$) (b) Self Anchor ($t = 3$) (c) Certificate Graph ($t = 3$)

Figure 6: Average trusted recall in a 1000 agent collective where each agent requires three certificates to join with five genesis agents. Shaded regions show the IQR of trusted recall across agents.



(a) Genesis Anchor ($t = 2$) (b) Self Anchor ($t = 2$) (c) Certificate Graph ($t = 2$)

Figure 7: Average trusted recall in a 2000 agent collective where each agent requires two certificates to join with five genesis agents. Shaded regions show the IQR of trusted recall across agents.

E Related Work

Scalable and decentralized multi-agent systems. Recent work has explored architectures for connecting heterogeneous LLM agents beyond small, fixed teams. Internet of Agents (IoA) enables heterogeneous agents to discover and collaborate through an Internet-like architecture (Chen et al., 2025), while AgentNet and MACNET explore decentralized coordination and scalable multi-agent collaboration (Yang et al., 2026; Qian et al., 2025). Symphony similarly targets scalable collective intelligence, although communication remains mediated by shared infrastructure (Wang et al., 2025). PPAI supports capability-aware agent selection and load balancing in a P2P population (Wang et al., 2026), while Zhang et al. (2026) study team formation and cooperation among agents. These systems primarily focus on discovery, routing, or a particular coordination mechanism. **PANDA** instead solves for both of these and leverages its collective to improve fault tolerance.

Multi-agent orchestration. A complementary line of work studies how agents should communicate once selected. Centralized frameworks such as Magentic-One and MetaGPT impose predefined planning and coordination structures (Fourney et al., 2024; Hong et al., 2024), whereas DyLAN dynamically selects agents and communication structure for individual tasks (Liu et al., 2024). GPTSwarm represents agent workflows as optimizable computational graphs (Zhuge et al., 2024), and Graph-of-Agents similarly performs agent selection and graph-structured message passing (Yun et al., 2026). **PANDA** separates the execution substrate from the orchestration strategy but draws inspiration from existing systems for its star, chain, and mesh topologies. G-Designer, AgentPrune, and AdaptOrch explore how to adaptively select an optimal MAS topology for a given task, these works could be integrated into **PANDA**'s protocol to allow for dynamic topology selection (Zhang et al., 2025b;a; Yu, 2026).

Fault tolerance and recoverable agent execution. Reliability in LLM-based MAS has largely been studied at the level of agent outputs and coordination errors. AgentScope provides robustness against faulty LLM or tool APIs (Gao et al., 2024). Cemri et al. (2026) and tse Huang et al. (2025)

show how MAS failures can arise from system design, inter-agent interaction, and verification failures, rather than solely from errors in individual model outputs. Zheng et al. (2026) and Lee et al. (2026) study resilience to faulty or Byzantine agents through consensus and ExoFlow explores execution and recovery for DAG-based workflows (Zhuang et al., 2023). **PANDA** additionally considers infrastructure failures and connects them with workflow recovery.

Decentralized trust and governance. Governance for agentic systems commonly assumes a logically centralized provider. SAGA (Syros et al., 2026) enforces identity and interaction policies through a governance service, and subsequent work strengthens such governance against post-quantum and byzantine threats (Avizeh et al., 2026; Laws et al., 2026). The NANDA index enables cryptographically verifiable discovery, but is designed as a one-time federated substrate to initiate interactions, not a collective architecture (Raskar et al., 2025). Garzon et al. (2025) anchor agent identities on a decentralized ledger, but still rely on credentials issued by commonly trusted third parties. Decentralized trust itself predates agents. SDSI proposed public-key-based authorization using certificates and linked local namespaces instead of a centralized hierarchy (Rivest & Lampson, 1996), PGP (Zimmermann, 1995) introduced the concept of a web-of-trust and Mass & Shehory (2001) proposed distributed certificate-based trust for e-commerce applications. **PANDA** builds on these ideas for LLM-based MAS by integrating decentralized trust into its collective.