

When Messages Go Wrong: Understanding Communication Faults in LLM-Based Multi-Agent Systems

JINHAO ZHOU, Waseda University, Japan

XIAOYU ZHANG*, The Hong Kong University of Science and Technology, Hong Kong SAR, China

JIONGCHI YU, Nanyang Technological University, Singapore

FAHAO CHEN, Shandong University, China

JUN WU*, Shanghai Jiao Tong University, China

Large language model-based multi-agent systems (LLM-MAS) increasingly employ specialized agents that exchange messages to accomplish complex tasks, including repository-level code generation. As these systems move from controlled demonstrations toward real-world applications, their reliance on networked protocols, remote tools, and model endpoints exposes agent communication to message loss, delay, and context expiration. Unlike traditional distributed systems that mainly exchange structured messages, LLM-MAS frequently rely on open-ended natural language to convey instructions and evidence. Communication anomalies can therefore alter task-relevant information while leaving messages processable, allowing semantic errors such as constraint loss and plan drift to propagate through subsequent agent interactions. Although previous research has examined adversarial threats and failures arising within individual agents, systematic investigations of inter-agent communication faults remain limited. This paper presents a large-scale empirical study of how communication faults manifest, propagate, and affect task outcomes in LLM-MAS. We construct a cross-layer taxonomy linking lower-stack disturbances to application-visible faults and develop a fault-injection and tracing pipeline to examine their downstream consequences. Our evaluation covers more than 29,000 executions across 100 tasks, five domains, three workflow topologies, and three model configurations. The results reveal that 20.7% of failures with complete detection records occur without an immediate failure signal, reaching 76.2% for partial-content faults. Workflow topology also substantially influences fault tolerance. Flat maintains 53.0% task success under communication faults, compared with 3.3% and 3.2% for Sequential and Hierarchical, respectively. This advantage primarily arises when an alternative information path preserves the evidence required by downstream agents. We further investigate execution-side and evidence protection mechanisms and find that they recover different failure cases. Evidence recovery, however, depends on whether trustworthy information remains accessible through an unaffected path. These observations offer empirical insights into communication-fault propagation, workflow resilience, and protection design for LLM-MAS. The code and experimental artifacts are available at https://github.com/SonGoKu21/mas_communication/, with an interactive companion website at https://songoku21.github.io/mas_communication/.

Additional Key Words and Phrases: LLM agents, multi-agent systems, communication faults, reliability, fault injection, failure propagation

1 Introduction

Large language models (LLMs) increasingly support multi-agent software systems in which specialized agents collaborate through message exchange [19, 21, 25, 37, 45]. These LLM-based multi-agent systems (LLM-MAS) already serve tasks such as web navigation, software engineering, and office automation [28, 46, 48]. As LLM-MAS move toward real-world deployment, reliable agent communication becomes critical because communication faults can disrupt coordination, alter intermediate information, and ultimately affect task outcomes. Understanding how these faults manifest and affect downstream execution is therefore essential for building dependable LLM-MAS.

*Corresponding author.

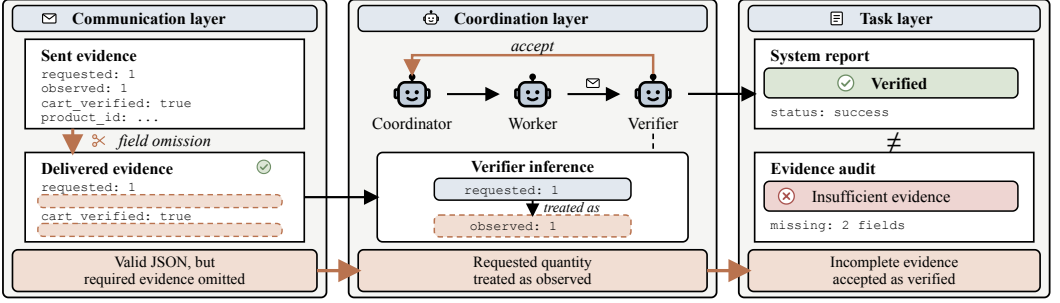


Fig. 1. A silent partial-content communication failure in an LLM-MAS. Although the message remains syntactically valid, the Verifier accepts incomplete evidence without an immediate failure signal.

Crucially, a real-world LLM-MAS rarely runs inside a single process. Its agents coordinate over networked protocols such as the Model Context Protocol and Agent-to-Agent messaging, and they access remote tools, sandboxes, and model endpoints through these channels [15, 36, 45]. Emerging deployments push this further into physically distributed settings, for example multi-robot teams driven by LLM agents. As a result, an LLM-MAS can encounter communication disturbances across different layers, including network delay and loss, connection or request timeouts, logical-message non-delivery, and context expiration [11, 16, 17]. Traditional distributed systems also face these issues, but structured protocols provide explicit mechanisms for detecting and handling many communication errors. LLM-MAS, however, often exchange open-ended natural language that each agent interprets autonomously. A communication fault can leave a message processable while altering the information consumed downstream. This is particularly important in LLM-MAS, where later agents depend on requests, observations, and evidence produced earlier in the workflow. As illustrated in Figure 1, a partial delivery can remove required evidence fields while leaving the message syntactically valid, allowing the workflow to continue without an immediate failure signal. Existing research has mainly examined adversarial threats, internal agent failures, coordination errors, and broader reliability issues [9, 22, 31, 43, 44, 47, 50]. Systematic studies of faults in inter-agent communication remain limited.

To the best of our knowledge, we conduct the first empirical study dedicated to communication faults in LLM-MAS. We formalize a cross-layer communication-fault taxonomy that connects established lower-stack communication anomalies, such as packet loss and stream interruption, to application-visible faults such as message unavailability and incomplete delivery. Building on this taxonomy, we design a fault-injection and tracing pipeline that perturbs inter-agent communication during MAS execution and records how the resulting effects manifest in downstream workflow traces. Our study covers more than 29,000 executions across 100 tasks, five domains, three workflow topologies, and three model configurations. Among the scheduled fault executions, 15,882 reach their target communication boundaries and are successfully applied. We additionally conduct targeted source-mapping and position-comparison experiments, together with a 1,098-execution main protection study and follow-up analyses of recovery conditions. Our investigation is driven by four overarching Research Questions (RQs):

- **RQ1 (Manifestation & Propagation): How do communication faults become visible to agents and propagate to MAS consequences and final task outcomes?** Across 3,780 source-mapping executions, lower-stack disturbances exhibit a many-to-many mapping to agent-visible faults. Among 15,882 applied faults, 89.4% are associated with recorded downstream MAS consequences and 80.2% end in task failure. Meanwhile, 1,457 executions still succeed after a

recorded consequence. Among 8,335 failures with complete detection records, 20.7% remain silent. These results separate fault manifestation, downstream consequence, outcome, and failure visibility.

• **RQ2 (Sensitivity): How do injection position and task context shape the impact of communication faults?** Matched comparisons show substantial differences across communication stages. In Admin, Flat retains success rates from 94.4% to 100.0% under I4 non-delivery across the three models. Sequential and Hierarchical both fall to 0% at I4, although they retain 34.7% to 75.9% success at I3. The impact of message loss therefore depends on the information carried at each stage and whether subsequent execution can obtain the missing information.

• **RQ3 (Fault Tolerance): How do topologies and model configurations affect outcomes under communication faults?** Matched comparisons show that Flat gains most of its advantage at I4. Compared with Sequential and Hierarchical, Flat achieves absolute success-rate gains of 66.5-66.6% under I4 non-delivery and 41.3-42.3% under I4 corruption, while differences at I2 and I3 remain below 0.7%. Across 1,408 common faulted units, changing to Flat recovers 19.3% to 27.0% of baseline failures, whereas changing the LLM recovers only 0.2% to 4.5%. The two changes recover largely different failure cases, showing that workflow structure and model configuration provide different forms of fault tolerance.

• **RQ4 (Protection Mechanisms): Which mechanisms protect execution and information dependencies against communication faults?** We conduct a 1,098 execution study comparing execution-side and evidence protection, while targeted follow-up analyses examine recovery conditions and overhead. The two mechanisms recover complementary failure sets. Evidence recovery remains effective while the workflow retains access to an unaffected information path, but fails when the same fault reaches every available evidence path.

This paper makes the following contributions:

- **A cross-layer communication-fault taxonomy for LLM-MAS.** We organize lower-stack communication faults, agent-visible message faults, and downstream MAS consequences within a cross-layer taxonomy. The taxonomy distinguishes fault origin from what a receiving agent observes and further captures faults introduced by agent-runtime handling.
- **A fault-injection and tracing pipeline for inter-agent communication.** We build a pipeline that injects faults at concrete communication boundaries, verifies their application-visible manifestation, and follows each affected communication dependency through downstream workflow state and task outcome.
- **A large-scale empirical characterization of communication faults.** Across 100 tasks, five domains, three topologies, and three deployed model configurations, we quantify fault manifestation, sensitivity, and tolerance under controlled communication faults. Targeted follow-up studies further characterize silent failures, protection mechanisms, and the conditions required for successful recovery.
- **An open-source release.** We publicly release the necessary experimental results, scripts, artifacts, and an interactive companion website to support reproduction and future research on communication reliability in LLM-MAS.¹

2 Background & Related Work

2.1 LLM-MAS

An LLM-based multi-agent system (LLM-MAS) consists of specialized, model-backed agents that collaborate through message exchange to solve complex tasks [19, 21, 25, 37, 45]. Although existing frameworks adopt different orchestration mechanisms, for example, AutoGen supports message-oriented interaction [45], while MetaGPT emphasizes role-based collaboration [25, 37], they share

¹https://songoku21.github.io/mas_communication/

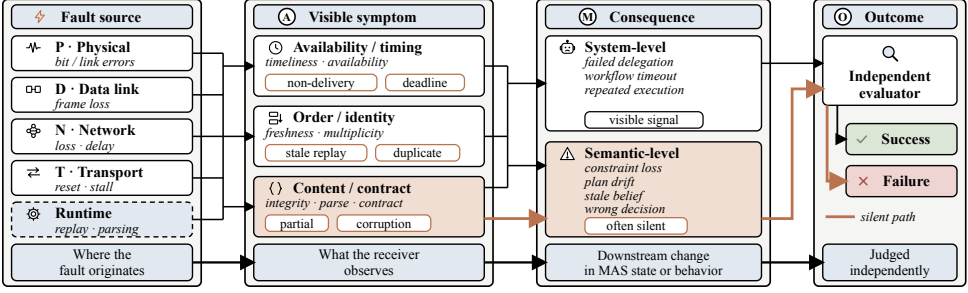


Fig. 2. Cross-layer communication-fault propagation. P/D/N/T denote conventional lower-stack communication layers; A represents application-level inter-agent communication and receiver-visible communication faults; M and O represent downstream MAS consequences and task outcomes.

a common communication dependency: a request, observation, tool result, or piece of evidence produced by one agent may become the input to a later workflow step. Such communication may further span remote tools, model endpoints, and physically distributed environments through protocols such as the Model Context Protocol (MCP) and Agent-to-Agent (A2A) messaging [15, 36]. Following standard communication reference models [7, 26], Figure 2 illustrates this communication path and its downstream effects. P/D/N/T denote the conventional physical, data-link, network, and transport layers, respectively; A denotes application-level inter-agent communication, while M and O denote downstream MAS consequences and final task outcomes. We characterize communication-fault propagation through four analytical stages: the *source* where a disturbance originates, the receiver-visible *symptom* exposed at A, the downstream MAS *consequence* at M, and the final task *outcome* at O. These stages describe the propagation of a communication fault through an LLM-MAS rather than additional protocol layers.

Formally, we model an LLM-MAS as

$$\Sigma = \langle G, \mathcal{B}, \mathcal{T}, \mathcal{S}, \mathcal{C}, \mathcal{R}, \mathcal{E} \rangle, \quad (1)$$

where G is the communication graph, $\mathcal{B}$ the deployed agents, $\mathcal{T}$ the tools, $\mathcal{S}$ the workflow states, $\mathcal{C}$ the receiver contracts, $\mathcal{R}$ the message-handling policies, and $\mathcal{E}$ the task environment. We represent a logical communication *carrier* as

$$m_t = \langle id, src, dst, task, session, round, payload, schema, version, time \rangle. \quad (2)$$

Application-layer operations such as dispatching, queueing, association, ordering, and structured-message processing determine how a carrier reaches and is consumed by a receiving agent. Recent empirical studies mainly characterize failures arising from agent reasoning, system design, verification, or individual agent components [9, 47, 50]. In contrast, we treat communication faults along the message path as first-class causes and study how their effects propagate from receiver-visible symptoms to downstream MAS behavior and task outcomes.

2.2 Communication Faults

Traditional networked and distributed systems have long treated communication faults as a major reliability concern, including loss, delay, reordering, duplication, corruption, and connection interruption [7, 26]. Communication stacks use checksums, retransmission, timers, duplicate handling, and redundancy to preserve reliable execution under such faults [4, 6, 11, 42]. Similar problems also arise in service-oriented systems, where timeouts and partial failures can propagate across remote-call dependencies [5, 24, 49].

LLM-MAS introduce an additional challenge because successful delivery does not guarantee that the receiving agent obtains valid task information. A message can remain parseable while carrying incomplete, stale, misassociated, or corrupted content, allowing the affected information to propagate through later workflow steps. Recent studies have reported related communication and information faults in LLM-based agent systems [27, 43]. Following the end-to-end principle [40], application-level validation is therefore important when task semantics determine whether delivered information remains valid. These characteristics motivate our cross-layer taxonomy and the fault-injection analysis in Section 3.

2.3 Fault Injection

Many communication faults are too rare, timing-dependent, or unsafe to reproduce by waiting on a live deployment, so fault injection deliberately introduces them at selected interfaces and observes the downstream effects [3]. This makes the space of failure conditions controllable and repeatable, which is why fault injection has become a standard way to evaluate the resilience of dependable systems. Early techniques emulate hardware and I/O faults in processor and memory state without special equipment [3, 8], and later system-level tools carry the same idea into distributed software. FATE injects failures at system and library I/O points to explore combinations of faults an application may face [18], lineage-driven fault injection targets the dependencies whose disruption is most likely to invalidate an outcome [2], and network emulators such as NetEm reproduce loss, delay, duplication, and corruption on real traffic [23]. At the service level, Gremlin perturbs inter-service messages to test microservice resilience [24], and chaos engineering injects controlled faults into production systems to check that they tolerate realistic failures [5, 49].

For agent systems, AgentChaos injects faults by modifying LLM API responses [43], while MAS-FIRE includes application-level communication faults in a broader LLM-MAS fault taxonomy [27]. Seyedghorban et al. [41] combine OpenTelemetry-based tracing with runtime fault injection across agent, tool, and LLM boundaries, and ReliabilityBench [20] evaluates tool-using agents under controlled API and execution failures. Our study focuses specifically on communication faults within LLM-MAS. We use a cross-layer taxonomy and matched executions to analyze their manifestations, stage sensitivity, fault tolerance, silent failures, and protection mechanisms across multiple task domains. Section 3 formalizes this distinction through our taxonomy and fault-injection pipeline.

3 Methodology

Our methodology consists of two components. The taxonomy consolidates communication-fault phenomena from prior work and organizes them along the cross-layer path introduced in Section 2.1. Based on this taxonomy, the MAS fault-injection pipeline instantiates selected receiver-visible faults at controlled communication boundaries and traces how their effects propagate through downstream MAS state and task outcomes. The pipeline consists of two phases: fault generation and propagation evaluation.

3.1 Taxonomy

We construct the taxonomy from 31 communication standards, protocol specifications, and prior studies on dependability and LLM-based agents [4, 7, 9, 15, 26, 27, 36, 43]. We extract 57 fault and effect descriptions and consolidate them into 43 candidate concepts. The three researchers include one with at least five years of experience in communication systems and two with at least two years of experience in LLM-based agents and multi-agent systems. After a 10-candidate pilot, two researchers independently code all candidates by source layer, receiver-visible manifestation, and downstream MAS consequence. Cohen’s κ ranges from 0.79 to 0.86 across the three dimensions before a third researcher resolves disagreements, yielding 29 final categories. We retain mappings

supported by source evidence and validate selected reproducible lower-stack mappings in our testbed. Conditional mappings are preserved when the same source fault produces different effects under different runtime conditions. As shown in Table 1, P/D/N/T denote lower-stack sources, A receiver-visible faults, and M downstream MAS consequences.

The distinction between these groups separates fault origin from receiver-visible manifestation and downstream consequence. P/D/N/T characterize where a communication disturbance originates, whereas A characterizes what the receiving agent actually observes. An A-layer fault may result from a lower-stack disturbance or originate directly in agent-runtime communication, for example during message delivery, ordering, replay, association, or structured-message handling. The source-to-A relationship is therefore conditional. A lower-stack disturbance may produce no tracked receiver-visible manifestation, while different sources may expose the same receiver-visible communication fault. M differs from both P/D/N/T and A because it records a trace-linked change in MAS state or behavior after affected information is accepted or used. We distinguish system-level consequences, such as failed delegation, workflow timeout, or repeated execution, from semantic-level consequences, such as constraint loss, incomplete aggregation, plan drift, stale-belief acceptance, or incorrect collective decisions. Errors originating purely from local model reasoning, parameters, activations, or training state remain outside the scope of this taxonomy.

3.2 MAS Fault Injection Pipeline

Phase I: Fault generation. For each task, workflow, model configuration, and repetition, the pipeline selects a fault operator, its parameters, and a communication boundary for injection. We define four communication boundaries, I1, I2, I3, and I4. I1 denotes the initial task dispatch from the Controller to the workflow entry agent and is excluded because disrupting this boundary can prevent the collaborative workflow from starting. The pipeline therefore injects faults at three boundaries during workflow execution. *Action requests* (I2) carry instructions or tool requests required for subsequent work. *Observations* (I3) return environment or tool results to the workflow. *Evidence handoffs* (I4) transfer generated results or supporting evidence to a downstream verifier or decision agent. The selected boundary, communication edge, carrier identity, and task/session association jointly determine the communication dependency affected by the intervention.

An interceptor operates after sender-side carrier production and before receiver-side consumption. For operator f with parameters θ , it transforms the communication carrier, delivery behavior, or relevant A-layer runtime state as

$$\phi_{f,\theta}(m_t, S_t) \rightarrow (m'_t, S'_t), \quad (3)$$

where m_t is the original communication carrier and S_t denotes the relevant runtime/workflow state. Delivery interception implements omission, delay, or duplication; queue and replay handling implement ordering/freshness faults; message association implements semantic-integrity faults; and structured key/type processing implements contract-validity faults. The intervention is restricted to the communication path and does not modify local model reasoning or tool semantics; model-provider requests and responses also remain outside the injector. Clean executions traverse the same instrumentation path using an identity operation, so the experimental configuration is held fixed apart from the selected communication intervention.

Phase II: Propagation evaluation. The evaluation phase first verifies whether the scheduled intervention reaches the receiving application boundary by comparing the original and delivered carrier or the corresponding delivery-event trace. If execution terminates before the targeted communication event, we mark the intervention as not applied. This separates scheduled faults from realized exposures and ensures that downstream analysis includes only faults that reach the intended boundary. For each applied fault, the pipeline separately evaluates application exposure

Table 1. Cross-layer taxonomy and representative sources. P/D/N/T entries are lower-stack communication-source families; A entries are receiver-visible communication-fault families; and M entries are downstream MAS consequence groups. Representative sources for the underlying phenomena are listed in the final column.

Layer	Fault / effect name	Brief description	Representative source(s)
P	P-F1 Bit-level corruption	Isolated or burst errors in communicated bits.	[42]
	P-F2 Link/endpoint interruption	NIC, cable, or power disruption removes a path.	[7, 32]
	P-F3 Physical-medium interference	Interference degrades reception and delivery timing.	[29]
	P-F4 Timing instability	Clock drift affects communication timing decisions.	[34]
	P-F5 Communication-state corruption	Hardware errors affect communication buffers or runtime state.	[8, 33]
D	D-F1 Frame loss/local drop	Frames are not delivered across a local link.	[13, 29]
	D-F2 Frame corruption/integrity failure	Damaged frames fail integrity checks or escape detection.	[13, 42]
	D-F3 Duplicate/reordered delivery	Frame order or multiplicity changes.	[23]
	D-F4 MTU/fragmentation failure	Size handling or reassembly disrupts delivery.	[30]
	D-F5 Contention/link instability	Contention or flapping delays or interrupts service.	[29]
N	N-F1 Packet loss	Independent or burst loss affects a packet path.	[7, 23]
	N-F2 Path performance degradation	Delay, jitter, bandwidth limits, or congestion slow delivery.	[1, 23]
	N-F3 Ordering/multiplicity anomaly	Packets arrive out of order or more than once.	[23]
	N-F4 Reachability/routing failure	Partitioning, asymmetry, or filtering blocks a path.	[7]
	N-F5 Endpoint resolution failure	Name resolution prevents endpoint discovery.	[35]
T	T-F1 Connection/session interruption	Reset or interruption breaks an active exchange.	[11]
	T-F2 Deadline/expiration failure	A delivery or validity budget expires.	[12, 16]
	T-F3 Retry/retransmission anomaly	Repeated attempts amplify load or logical work.	[11, 17]
	T-F4 Stream completeness failure	An exchange ends before complete content arrives.	[11, 14]
	T-F5 Transport pressure/stall	Backlog or flow-control pressure prevents progress.	[1, 11]
A	Timeliness	A logical message arrives late or misses its deadline.	[16, 24]
	Availability	A required logical message is not delivered.	[24, 49]
	Semantic integrity	A processable message carries changed or misassociated meaning.	[22, 43]
	Parseability/completeness	A message cannot be parsed or lacks required content/evidence.	[36]
	Multiplicity/idempotency	A logical request or delivery is repeated.	[17, 24]
	Ordering/freshness	The receiver observes displaced, out-of-order, or stale state.	[11, 43]
	Contract validity	Structured keys, types, schema, or relations violate the expected contract.	[15, 36]
M	System-level consequences	Execution or coordination degrades, e.g., failed delegation or workflow timeout.	[9, 47, 50]
	Semantic-level consequences	Accepted evidence, plans, beliefs, or decisions degrade.	[9, 22, 31]

(A), downstream MAS consequence (M), final task outcome (O), and the auxiliary handling-response flag (B). A is determined by the observed message or delivery change, independent of final task success. The trace then follows the affected communication dependency through accepted state, actions, aggregation, verification, and termination. We assign an M consequence only when the

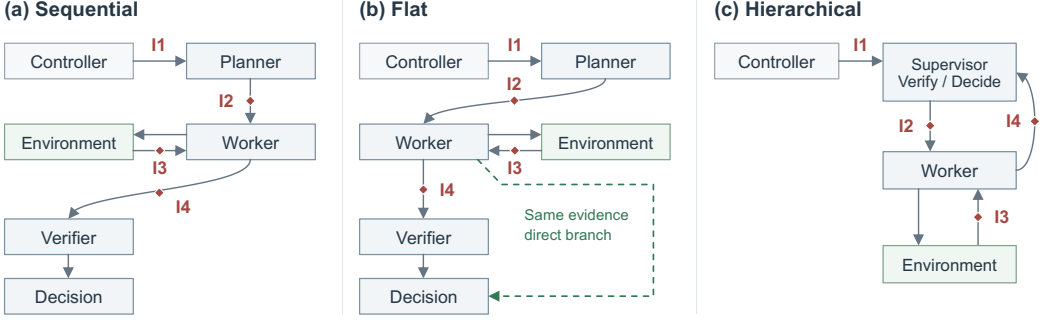


Fig. 3. AutoGen workflows and injection boundaries used in the confirmation matrix.

trace shows a downstream state or behavioral change associated with the affected dependency. For semantic consequences, the altered information must also be accepted or used by a downstream agent. For missing carriers, the trace identifies the action or state that cannot complete because the required communication is unavailable. The task evaluator determines O independently of A and M . B records a handling response after fault exposure. We retain it as an auxiliary observation because it does not establish complete detection, correction, and state restoration. Each execution records the communication event, original and delivered carrier projections, message and session metadata, timing and order, accepted state, actions, termination, and evaluator outcome. These records support the subsequent analyses of manifestation, propagation, sensitivity, fault tolerance, and protection mechanisms.

4 Experimental Evaluation

4.1 Experimental Setup

Tasks and datasets. We evaluate communication faults on 100 tasks drawn from five task domains: WebArena Shopping, WebArena Admin, WebArena Reddit [48], SWE-bench [28], and TheAgentCompany (TAC) [46]. These workloads cover interactive web tasks, administrative operations, social-platform reasoning, repository-level software engineering, and multi-step workplace tasks, providing diverse communication and evidence dependencies. Each task is executed under matched clean and faulted conditions so that fault-induced effects can be separated from failures already present in the clean execution. Task-specific adapters provide the final success endpoint; for evidence-based workflows, this endpoint may additionally require that the final result is supported by the expected delivered evidence rather than only matching the native benchmark answer.

MAS configurations. We instantiate three AutoGen workflow structures, shown in Figure 3. *Sequential* follows a single communication path from Planner to Worker, then through a Verifier to the final Decision. *Flat* retains this path but adds an alternate direct branch that forwards the same pre-interception worker evidence to the Decision. *Hierarchical* replaces the separate Planner-Verifier-Decision chain with a Supervisor that delegates the task to the Worker and subsequently receives the worker evidence for verification and decision. The workflows use three deployed model configurations: DeepSeek V4 Flash [10], Qwen3.8-27B [39], and Qwen3.5-9B [38]. Topology comparisons match executions on task, model configuration, fault condition, injection position, and repetition, while model-configuration comparisons additionally hold the topology fixed. Because model provider and serving environment vary with the deployed configuration, we interpret these settings as configuration-level rather than isolated model-size comparisons.

Faults and injection positions. We inject receiver-visible communication faults at I2-I4, as defined in Section 3.2 and illustrated in Figure 3. I1 is excluded because disrupting the initial task dispatch can prevent inter-agent execution from starting. The primary confirmation matrix contains one clean condition, I2 non-delivery, and five I4 faults: valid-partial delivery, semantic corruption, stale replay, deadline timeout, and non-delivery. Across 100 tasks, three models, three topologies, and three repetitions, this yields 18,900 executions. Among 16,200 scheduled faulted executions, 15,882 reach the target boundary and receive the fault; the remaining 318 are excluded from fault-effect analyses. We also construct a position-comparison matrix containing clean execution, non-delivery at I2-I4, and semantic corruption at I3-I4. It contains 16,200 scheduled cells, including 10,800 shared with the primary matrix and 5,400 additional I3 cells. RQ2 uses the non-delivery subset, while Figure 8(b) uses all five fault conditions for the topology analysis. One scheduled cell ends in retry exhaustion, so this count denotes scheduled cells rather than completed executions.

Metrics. We evaluate fault behavior at the application, MAS, and task levels. *A exposure* indicates that the scheduled intervention produces the intended receiver-visible message or delivery change. *M propagation* indicates a trace-linked downstream change in MAS state or behavior after the affected dependency is processed. *O outcome* records the final task-level success or failure independently of A and M. We additionally report the historical handling-response flag *B* as an auxiliary observation, but do not interpret it as a strict recovery measure. To quantify degradation relative to executions that would otherwise succeed, we use clean-conditional loss:

$$L_{\text{clean}} = \frac{\#\{\text{applied fault fails; matched clean succeeds}\}}{\#\{\text{applied fault has a successful matched clean}\}}. \quad (4)$$

We define a *silent-failure rate* for applied faults that continue downstream without immediate rejection or termination and ultimately fail. The rate is reported in RQ1. A detection record is complete when all fields required to determine fault handling and failure visibility are present in the execution trace.

Execution protocol. Clean and faulted executions use the same instrumented workflow, with an identity operation as the clean control. Each combination of task, configuration, and condition is executed three times. Reported rates pool all included executions across the three repetitions, while per-repetition rates and their sample standard deviations assess execution variability. Fault-effect analyses include only executions in which the scheduled fault is applied. Additional execution matrices, inclusion criteria, and sample sizes are reported with the corresponding RQs.

Statistical analysis. For the confirmation matrix, we estimate 95% confidence intervals using 10,000 task-clustered bootstrap resamples. Paired comparisons use task-level differences and two-sided sign-flip permutation tests. We compute paired effect sizes and Holm-adjusted *p*-values within each dataset and metric family. Detailed effect sizes, confidence intervals, and raw and adjusted *p*-values are provided in the research artifact. Topology and model comparisons use matched execution cells. Cross-domain comparisons remain descriptive because tasks from different datasets are not paired.

4.2 RQ1. Manifestation and Propagation

We investigate how communication faults become visible across system layers and how they affect MAS execution. We first analyze 3,780 source-mapping executions, including 3,600 faulted runs and 180 clean runs, to study how lower-stack fault sources appear at the application layer. We then examine 15,882 applied application-layer faults to analyze their MAS consequences and final task outcomes. Finally, we study failure visibility using executions with complete detection records.

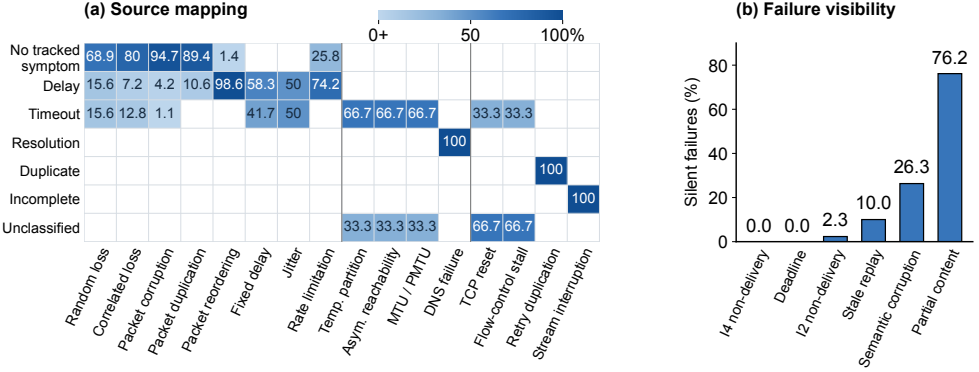


Fig. 4. Cross-layer manifestation and failure visibility under communication faults. The left panel shows the distribution of primary application-level manifestations across 3,600 faulted source-mapping executions. The right panel shows silent-failure rates among 8,335 failures with complete detection records. Lower-stack fault sources exhibit many-to-many mappings to application-level symptoms, while silent failures concentrate in partial-content and semantic-corruption faults.

Cross-layer manifestation and downstream outcomes. As shown in Figure 4 (a), lower-stack fault sources do not map one-to-one to application-level manifestations. Across 3,600 faulted source-mapping executions, 36.0% show no recorded consumption symptom within the tracked categories. The remaining executions reveal substantial variation across layers. For example, fixed delay appears as delayed delivery in 58.3% of executions and as timeout in 41.7%. Jitter is evenly divided between these two manifestations. The reverse pattern also occurs. Timeout arises from multiple lower-stack sources, including loss, jitter, temporary partition, MTU/PMTU faults, TCP reset, and flow-control stall. The same application-level manifestation can therefore originate from different communication conditions. The mapping also differs between faults that appear similar across layers. Source-level packet corruption produces no recorded accepted-corruption manifestation in our mapping runs and mostly results in no tracked consumption symptom. Likewise, lower-stack packet duplication does not directly appear as logical duplication, while retry-induced duplication consistently produces a duplicate manifestation. These results support the separation between fault sources and agent-visible faults in our taxonomy. The source layer identifies where a communication fault originates, while the application layer captures what the receiving agent observes.

We further examine whether an observed application-level fault determines the final task result. Across the canonical fault categories, task failure rates range from 73.4% for message timeout to 85.5% for partial content. Across all 15,882 applied faults, 89.4% are associated with a recorded MAS consequence and 80.2% end in task failure. However, 1,457 executions still succeed after a recorded MAS consequence, accounting for 10.3% of consequence-labeled runs. Thus, an agent-visible fault or downstream MAS consequence does not uniquely determine the final outcome.

Finding 1: Lower-stack fault sources do not map one-to-one to agent-visible faults or final outcomes. The same application-level manifestation can arise from multiple lower-stack sources, while individual sources such as delay and jitter can produce different manifestations. Across the observed application-level fault categories, task failure rates range from 73.4% to 85.5%, and 1,457 executions still succeed after a recorded MAS consequence.

Failure visibility. We next examine whether task failures become visible when the communication fault occurs. Among 8,335 failures with complete detection records, 1,722 remain silent,

corresponding to 20.7%. These executions continue without an immediate failure signal and later produce an unsuccessful task outcome. The same 1,722 cases account for 15.6% of the 11,022 injected runs with complete detection records. We exclude runs without detection records from this analysis instead of assigning them a detection outcome.

As illustrated in Figure 4 (b), the silent-failure rate varies substantially across fault types. Partial-content faults show the highest rate. Among detection-complete failures, 76.2% of partial-content faults remain silent, compared with 26.3% of semantic-corruption faults. Partial-content faults alone account for 69.6% of all silent failures in this subset. Stale replay reaches 10.0%, while I2 non-delivery reaches only 2.3%. We observe no silent failures for I4 non-delivery or deadline faults in the detection-complete subset. These differences reveal a clear pattern. Delivery faults often remove or delay an expected message and expose an execution-level symptom. Semantic faults can preserve a processable message while changing the information carried by that message. The runtime can therefore continue normal message processing even though downstream agents receive incomplete or incorrect evidence. This observation follows the end-to-end principle that successful communication at lower layers does not establish application-level correctness [40].

As shown in Figure 5, a representative Reddit execution illustrates this pattern. An I4 partial-content fault delivers a syntactically valid message from the Evidence Worker to the Verifier. The message remains parseable, but three required evidence fields are missing. The Verifier accepts the message even though the evaluation contract expects rejection. The workflow records no detection or recovery signal and later fails. The answer string still matches the expected answer, showing that the failure arises from the evidence contract rather than the answer text alone.

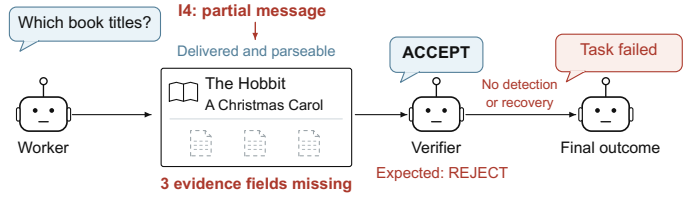


Fig. 5. Representative silent failure under partial-content delivery. The message remains parseable after required evidence is removed, and the Verifier accepts it without recorded detection or recovery.

Finding 2: Silent failures concentrate in faults that preserve a processable message.

Among failures with complete detection records, 20.7% remain silent. The rate reaches 76.2% for partial-content faults and 26.3% for semantic corruption, compared with 2.3% for I2 non-delivery and 0% for I4 non-delivery and deadline faults. Partial-content faults alone contribute 69.6% of the observed silent failures.

4.3 RQ2. Sensitivity

RQ2 investigates why the same communication fault can produce different task outcomes. We study two factors. First, we compare matched non-delivery faults at I2, I3, and I4 to examine how the affected communication stage changes fault impact. I2, I3, and I4 represent the request, state-return, and evidence-handoff stages, respectively. Second, we compare semantic faults across five task contexts to examine how the information required by each task changes fault sensitivity. For the stage comparison, we retain matched task-repeat units with a successful clean execution, applied faults at all three positions, and known outcomes. For the task-context comparison, we fix the Flat topology and compare semantic corruption with partial-content faults.

Sensitivity to communication stage. The matched results show no fixed severity order across I2, I3, and I4. Table 2 presents the clearest example in Admin. Under Flat, I4 non-delivery retains

success rates from 94.4% to 100.0% across the three models. Sequential and Hierarchical fall to 0% at the same stage. The pattern differs at I3, where these two topologies still retain success rates from 34.7% to 75.9%. The same non-delivery fault can therefore produce very different outcomes at different communication stages. Trace analysis explains this difference. In Admin, I3 occurs inside an iterative tool loop. Among 110 matched DeepSeek executions under Sequential and Hierarchical, 82 receive a later table carrier after the I3 fault. Among these 82 executions, 74 eventually succeed. All 28 executions without a later table carrier fail. I4 occurs after the navigation loop has finished. None of the 110 matched I4 executions contains a later tool call. The workflow therefore retains an opportunity to obtain another observation after I3, while this opportunity is no longer available after I4.

Figure 6 illustrates the same mechanism at the message level. The task asks the agents to identify two book titles. The paired executions use the same task, fault operator, topology, model configuration, and repetition. When the I2 request is lost, the Worker receives no task request and produces no corresponding evidence. The downstream agents therefore have no equivalent information to use. When the I4 handoff is lost, the Worker has already produced the evidence. The affected handoff removes one delivery path, while the direct Flat branch still carries the existing evidence to the Coordinator. The final outcome therefore depends on whether the affected information can still be obtained after the communication failure.

The other evaluated task contexts support this interpretation. In Reddit, SWE-bench, and TAC, I2 and I3 non-delivery produce no successful matched executions under the evaluated configurations. Our trace audit finds no later evidence reacquisition step after these faults. At I4, the required evidence has already been generated. Flat then retains substantial success through its additional evidence path. Across the three models, its I4 success ranges from 92.7% to 100.0% on Reddit, from 78.9% to 100.0% on SWE-bench, and from 42.9% to 100.0% on TAC. Shopping shows a weaker I4 effect, with Flat success ranging from 0% to 53.3%. These results show that communication stage affects fault impact through the information available afterward. A later observation can restore information lost inside an iterative workflow. An additional path can preserve information that already exists

Table 2. Matched task-success rates under non-delivery at I2, I3, and I4 in Admin. Each model uses the same matched task-repeat cohort across the three communication stages.

Model	Topology	I2 (%)	I3 (%)	I4 (%)
DeepSeek	Sequential	90.7	72.2	0.0
	Hierarchical	94.6	62.5	0.0
	Flat	87.0	75.9	94.4
Qwen3.5-9B	Sequential	72.4	75.9	0.0
	Hierarchical	75.0	75.0	0.0
	Flat	75.0	78.6	96.4
Qwen3.8-27B	Sequential	56.2	35.4	0.0
	Hierarchical	55.1	34.7	0.0
	Flat	55.1	34.7	100.0

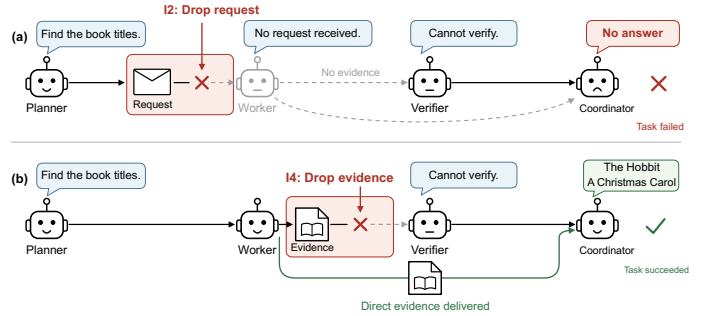


Fig. 6. Representative impact of non-delivery at different communication stages. At I2, request loss prevents the Worker from producing the required evidence. At I4, the evidence already exists and remains available through the Flat topology’s additional path.

before a downstream handoff fails. When neither source remains available, the missing information cannot reach subsequent agents. This behavior is consistent with prior work showing that execution failures depend on the data dependencies affected by a fault [2].

Finding 3: Fault impact depends on whether the affected information remains recoverable after the communication failure. The same non-delivery fault produces different outcomes across communication stages. Later observations recover some I3 losses in Admin, while an additional evidence path preserves many I4 executions under Flat.

Sensitivity to task context. The same semantic fault also shows substantially different effects across task contexts. We fix the Flat topology and compare semantic corruption with partial-content faults in Figure 7. Their relative severity changes sharply across the five evaluated settings. Shopping is most sensitive to semantic corruption. Its failure rate reaches 99.6%, compared with 53.0% under partial content. SWE-bench shows the opposite pattern. Semantic corruption causes only 3.1% failure, while partial content reaches 100.0%. TAC shows a similar contrast, with rates of 33.3% and 100.0%, respectively. Reddit is also more sensitive to partial content, with 97.7% failure compared with 66.7% under corruption. Admin remains less sensitive to both faults in this comparison. The differences follow how each evaluated task uses the affected information. The Reddit, SWE-bench, and TAC workflows require explicit evidence for the final decision. Partial-content faults can remove information required by this evidence contract. Shopping follows a different execution path. An environment action may already succeed before a later communication fault changes the evidence used for final acceptance. Admin retains an interactive tool loop and can obtain additional observations before the final evidence handoff. These differences change which communication faults become critical for task completion. Semantic corruption and partial-content faults exhibit different severity patterns across task contexts. Semantic corruption dominates partial content in Shopping, while partial content dominates corruption in SWE-bench and TAC. The impact of a semantic fault therefore depends on the information required by the task and how the workflow evaluates that information.

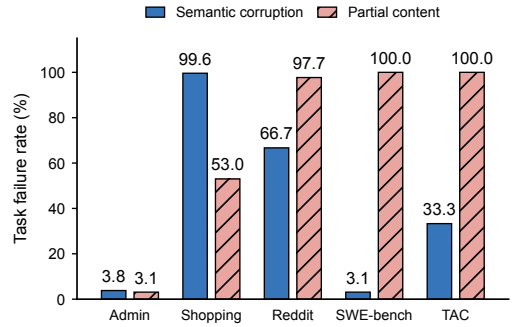


Fig. 7. Task-failure rates under semantic corruption and partial-content faults across five task contexts with the Flat topology.

Finding 4: The most harmful communication fault changes with task context. Under Flat, semantic corruption causes 99.6% failure in Shopping but only 3.1% in SWE-bench. Partial-content faults show the opposite pattern, reaching 53.0% in Shopping and 100.0% in SWE-bench. Fault sensitivity therefore depends on how the task uses and evaluates the affected information.

4.4 RQ3. Fault Tolerance

RQ3 investigates how topology and model configuration affect fault tolerance. We first compare Flat with Sequential and Hierarchical while fixing the task context, task, repetition, fault condition, and model. The paired analysis records both failure-to-success and success-to-failure transitions for the same execution units. We then use 1,408 common faulted units with known outcomes across all three models and three topologies to compare topology changes with model changes.

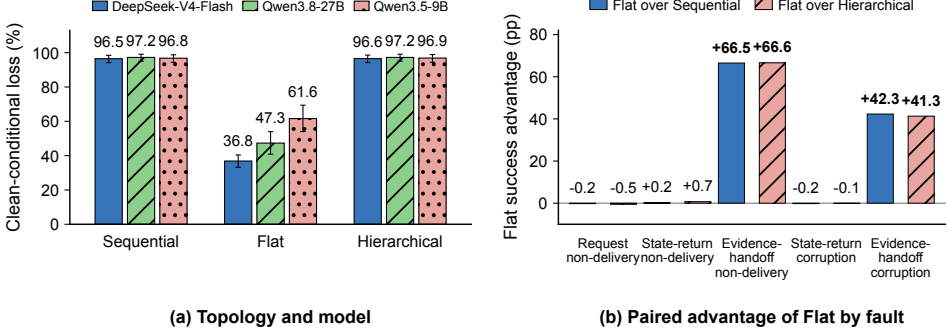


Fig. 8. Impacts of topology and model configuration under communication faults. (a) Clean-conditional loss across three topologies and three model configurations. Error bars report task-clustered 95% confidence intervals. (b) Paired success advantage of Flat over Sequential and Hierarchical under five fault conditions. Flat gains concentrate at the evidence handoff stage, where its branch carries the required evidence.

Impact of topology. Figure 8(a) shows a consistent topology pattern across the three model configurations. Across all applied faulted executions with known outcomes, Flat retains 53.0% task success, more than 16 $\times$ the 3.3% and 3.2% achieved by Sequential and Hierarchical. Consistently, Sequential and Hierarchical exhibit clean-conditional losses of 96.5-97.2%, while Flat reduces the loss to 36.8-61.6%. The paired analysis in Figure 8(b) further shows that Flat gains most of its advantage at the evidence handoff stage. Under non-delivery, changing Sequential to Flat recovers 584 failed executions without introducing new failures, corresponding to an absolute increase of 66.5% in success rate; the increase over Hierarchical is 66.6%. Evidence-handoff corruption shows a similar pattern, with absolute increases of 42.3% and 41.3% in success rate over Sequential and Hierarchical, respectively. The topology difference becomes small when the fault occurs before the additional Flat branch receives the required evidence.

At the request stage, the success-rate difference remains within 0.5%. At the state-return stage, it remains within 0.7% for non-delivery and 0.3% for semantic corruption. Sequential and Hierarchical also show similar behavior across these conditions. The large Flat advantage therefore appears when its additional branch already contains the information required by downstream agents. The topology in Figure 3 explains this pattern. The Flat branch diverges after the Worker has generated the required evidence. A fault at the evidence handoff can affect one route

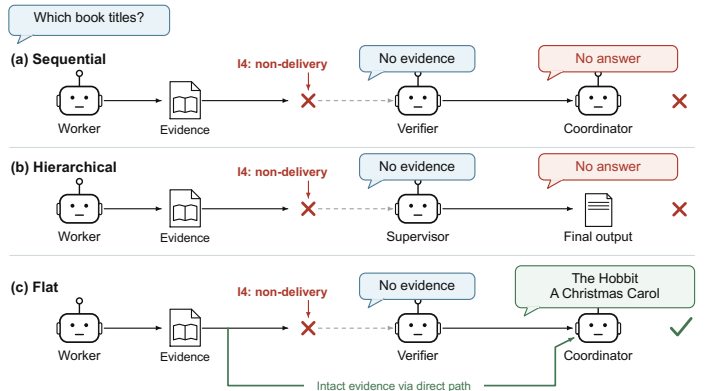


Fig. 9. Representative topology-dependent outcome under the same I4 non-delivery fault. Sequential and Hierarchical lose their only downstream evidence path, whereas Flat preserves the required evidence through its additional direct branch.

while the second route contin-

ues to carry the same evidence to the final decision. Faults at the request and state return stages occur before this divergence. Both Flat branches then depend on the same affected information. Figure 9 provides a trace-level example of this paired advantage. Under the same I4 non-delivery fault, only Flat preserves the required evidence through its direct branch. This is consistent with dependability principles in which redundancy helps when an alternative path avoids the affected dependency [2, 4]. The topology effect also remains stable across task compositions. Removing each task context in turn leaves the net Sequential-to-Flat success-rate improvement between 17.6% and 29.5%. The Hierarchical-to-Flat improvement remains between 17.5% and 29.4%. The same direction holds when we use all executions with known outcomes and when we restrict the analysis to executions with successful matched clean runs. Flat substantially reduces overall failures, while the remaining failures become increasingly concentrated in silent semantic faults. In the detection-complete matched cohort, Sequential and Hierarchical fail on 94.6% and 94.7% of executions, respectively, compared with 32.1% under Flat. The silent-failure rate per execution changes only slightly, from 17.6% under Sequential to 16.6% under Flat. Consequently, silent failures account for 51.6% of the failures that remain under Flat, compared with 18.6% under Sequential and 18.1% under Hierarchical. Semantic corruption and partial-content faults contribute 94.8% of these residual silent failures. The additional information path therefore removes many execution failures, while semantic faults remain the main residual risk.

Finding 5: Flat improves fault tolerance when its additional path still carries the required evidence. Its success-rate gain reaches 66.5% to 66.6% for evidence-handoff non-delivery and 41.3% to 42.3% for evidence-handoff corruption. The gain remains below 0.7% when the affected communication occurs before the additional path receives the required evidence.

Impact of model configuration. Model configuration changes the magnitude of fault impact, especially under Flat. As shown in Figure 8(a), clean-conditional loss under Flat ranges from 36.8% to 61.6% across the three models. Sequential and Hierarchical remain between 96.5% and 97.2% under the same model configurations. Model choice therefore affects how well agents handle the information that remains available, while substantial communication-induced degradation persists across all three models. We further quantify the recovery provided by model changes using 1,408 common faulted units. Changing the deployed model configuration recovers only 0.2% to 4.5% of baseline failures. In comparison, changing the topology to Flat recovers 19.3% to 27.0%. Across the 12 matched comparisons, the topology change provides an additional net success-rate gain of 17.5% to 23.4%. The failures recovered by the two changes also show little overlap, with Jaccard similarity ranging from 0 to 0.047. This result indicates that model changes can recover a small set of executions that respond differently to model configuration. Their overall recovery coverage, however, remains limited compared with the impact of changing the information path.

Finding 6: Changing the LLM provides limited recovery from communication faults. Across the common mixed-fault cohort, changing the LLM recovers only 0.2% to 4.5% of baseline failures, while changing to Flat recovers 19.3% to 27.0%. The recovered sets show little overlap, indicating that model changes affect a small and largely different subset of failures.

4.5 RQ4. Protection Mechanisms

RQ4 investigates how communication protection should respond to execution faults and information faults, and when recovery remains effective. Following recent studies on fault tolerance and verification in LLM-based multi-agent systems [9, 27], we organize the evaluated mechanisms into execution-side protection and application-level evidence protection. Execution-side protection uses

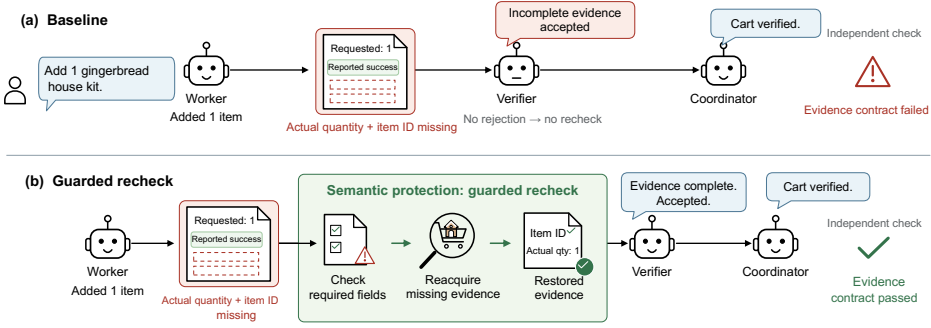


Fig. 10. Guarded evidence recovery repairs incomplete evidence without repeating the completed action.

resubmission and duplicate suppression to preserve intended actions, while evidence protection checks task-relevant evidence and reacquires information when the received content is incomplete or inconsistent. We then evaluate their effectiveness, recovery conditions, and execution overhead. **Fault-specific protection and evidence recovery.** The main protection study shows that execution faults and information faults benefit from different mechanisms. Among 106 baseline failures, execution-side protection uniquely recovers 67.9%, while evidence protection uniquely recovers the remaining 32.1%. The two recovery sets do not overlap. As shown in Figure 11, execution-side protection raises success from 0% to 100% under the evaluated non-delivery and duplicate-delivery faults. Its effect is small for the tested evidence faults. Evidence protection shows the complementary pattern. For valid partial content, success increases from 55.6% to 100%. For identity corruption, success increases from 50.0% to 100%. Combined protection covers both fault classes and reaches 100% success across the five tested conditions. Stale replay already achieves full task success in this task family, leaving no additional recovery gain for the protection mechanisms. The targeted evidence-protection study explains how this recovery occurs. Under Sequential, the baseline succeeds in only 5.6% of the evaluated valid-partial and identity-corruption executions. Evidence checking restores success when rejection leads to an available unaffected recovery path. The check identifies evidence that violates the expected contract. The recovery path then provides valid information for the downstream decision. This separation distinguishes fault detection from information recovery. Final task success can also hide errors inside the workflow. Under Flat, the baseline completes all evaluated faulted executions successfully, while 97.2% contain a recorded incorrect intermediate evidence acceptance. Evidence protection preserves the final success rate and removes these recorded incorrect acceptances. Protection therefore improves the internal evidence flow even when the final task outcome remains unchanged. Figure 10 illustrates this mechanism in a representative execution. The same pattern appears in the targeted

Reddit study. Protection raises success from 44.4% to 77.8% for valid partial content and from 33.3% to 77.8% for semantic corruption, matching the contemporaneous clean success rate. All protected

Success rate (%)	Execution-side		Information-side		No observed gain
Baseline	0	0	56	50	100
Execution only	100	100	56	50	100
Evidence only	0	0	100	100	100
Combined	100	100	100	100	100
	Non-delivery	Duplicate delivery	Valid partial	Identity corruption	Stale replay

Fig. 11. Success rates under the main protection study. Execution-side protection primarily recovers execution faults, whereas evidence protection recovers information faults.

faulted executions trigger fresh reacquisition, and 88.9% subsequently adopt the new evidence. The remaining failures persist after reacquisition, showing that recovery removes communication-induced degradation when trustworthy evidence can still be restored.

Recovery boundary and overhead. Recovery depends on whether the workflow can still obtain unaffected evidence. The source-level analysis makes this requirement explicit. Forwarding another copy of corrupted evidence preserves the same incorrect information, whereas independent reacquisition obtains a fresh observation for downstream agents. Additional copies therefore help only when they preserve access to information outside the affected fault path. The repeated-exposure study further identifies this recovery boundary. As summarized in Table 3, Combined protection restores all Sequential failures when corruption is limited to the initial evidence handoff and the recovery delivery remains unaffected. Recovery disappears when the same fault also reaches subsequent recovery deliveries. Flat remains successful under this broader exposure because its additional evidence path remains available. When the fault also reaches this path, Flat can no longer recover. The environment action remains successful across these conditions, showing that the final failures arise from unavailable trustworthy evidence rather than failed environment execution. Conditional reacquisition also reduces unnecessary communication during clean execution. Guarded readback triggers no dedicated reacquisition in the evaluated clean runs, whereas always-on readback triggers it in every run. Guarded readback removes two workflow HTTP requests per clean execution. This corresponds to a 15.4% reduction under Sequential and a 13.3% reduction under Flat, with no change in task success or model-call count. Under the evaluated fault conditions, both strategies invoke reacquisition when recovery is required. Guarded recovery therefore concentrates the additional communication cost on executions that fail the evidence check.

Table 3. Recovery boundary under repeated fault exposure. Entries show Baseline $\rightarrow$ Combined successful runs.

Fault scope	Sequential	Flat
Initial handoff	0 $\rightarrow$ 18	18 $\rightarrow$ 18
Handoff and recovery deliveries	0 $\rightarrow$ 0	18 $\rightarrow$ 18
All evidence paths	0 $\rightarrow$ 0	0 $\rightarrow$ 0

Each condition and protection arm contains 18 runs.

Finding 7: Effective protection must match the failed dependency and retain access to trustworthy evidence. Execution-side and evidence protection recover complementary failure sets. Evidence recovery remains effective while an unaffected evidence path is available, but fails when the same fault reaches every available path. Guarded reacquisition preserves this recovery capability while reducing unnecessary communication during clean execution.

5 Discussion

5.1 Implications

① **Information dependency.** Our results show that the impact of a communication fault depends strongly on the affected information dependency. The same fault can produce different outcomes across communication stages and task contexts, depending on whether later execution can obtain equivalent information. ② **Topology and redundancy.** Workflow topology improves fault tolerance when it preserves an unaffected information path. This explains why Flat provides large gains at evidence handoff but little benefit when the fault occurs before its additional branch receives the required information. ③ **Silent information faults.** Structural redundancy removes many overt execution failures, but it does not eliminate incomplete or corrupted information that remains processable. These faults can therefore account for a larger share of the remaining failures. Reliable debugging and protection should consider both message delivery and the trustworthiness of information consumed downstream.

5.2 Future Work

① **Broader systems and environments.** Future work can extend the study to additional agent frameworks, workflow topologies, communication protocols, and controlled model configurations. Such evaluations could clarify which observations generalize across systems and which depend on specific runtimes. ② **Task-centric sensitivity.** Characterizing tasks by evidence depth, recoverable state, alternative information paths, and critical dependencies may better explain fault sensitivity than dataset identity alone. This could also support task-aware fault prediction and protection design. ③ **Adaptive recovery.** Finer-grained instrumentation of detection and recovery could support adaptive protection that selects resubmission, alternate routing, evidence checking, or fresh reacquisition according to the affected dependency. Evaluating these mechanisms under persistent or correlated faults could further identify the limits of redundancy and recovery. Such analysis may also reveal when recovery itself becomes a new source of communication overhead or failure.

5.3 Threats to Validity

① **Construct validity.** Our measurements may not capture every stage of fault detection and recovery. We reduce this threat by separating receiver-visible exposure (A), downstream MAS consequence (M), and final task outcome (O), while also recording the handling-response flag B and trace-linked state changes. We assign semantic consequences only when altered information is accepted or used downstream, and we do not interpret B as evidence of complete detection, correction, or state restoration. ② **Internal validity.** Fault injection may introduce artifacts if an intervention does not represent a plausible communication fault or affect the intended dependency. We ground fault mappings in communication standards and prior studies, validate reproducible lower-stack mappings against their expected receiver-visible manifestations, verify each applied intervention in execution traces, and use matched clean and faulted executions to isolate fault-induced effects. Some physical and hardware-dependent mechanisms require conditions beyond our testbed, so we do not claim direct experimental reproduction of every underlying mechanism. ③ **External validity.** We evaluate 100 tasks from five domains, three AutoGen topologies, and three deployed model configurations. The observed effects may still depend on the evaluated communication paths, task contracts, runtime mechanisms, and framework implementation, so we restrict our conclusions to the evaluated settings. ④ **Conclusion validity.** LLM-MAS executions contain stochastic variation that may affect observed outcomes. We reduce this threat through repeated executions, matched comparisons, task-clustered statistical analyses, and uncertainty estimates reported with the quantitative results.

6 Conclusion

This work presents a cross-layer taxonomy and fault-injection pipeline for studying communication faults in LLM-based multi-agent systems. Across 15,882 applied faults, 89.4% propagate to an MAS consequence and 80.2% end in task failure, while some propagated faults still complete successfully, showing that manifestation, propagation, and final outcome capture different aspects of reliability. Our results further show that fault impact depends on the affected dependency, task context, workflow topology, and deployed model configuration. Protection must therefore match the failed dependency: execution faults benefit from action-side handling and path redundancy, whereas corrupted or incomplete information requires semantic checking and, in some cases, independent reacquisition. Reliable LLM-MAS communication requires protecting both message delivery and the trustworthiness of the information consumed downstream.

7 Data Availability

Our artifact provides source code, processed results, analysis configurations, operator mappings, trace/adjudication definitions, and scripts for reproducing the reported figures and tables. The artifact repository is available at [MAS-Communication](#), with a companion website at [MAS-Communication-Website](#). Raw execution logs, complete task environments, credentials, private content, and private model reasoning are not included.

References

- [1] Mark Allman, Vern Paxson, and Ethan Blanton. 2009. *TCP Congestion Control*. RFC 5681. RFC Editor. <https://doi.org/10.17487/RFC5681>
- [2] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-Driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 331–346. <https://doi.org/10.1145/2723372.2723711>
- [3] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. 1990. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering* 16, 2 (1990), 166–182. <https://doi.org/10.1109/32.44380>
- [4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. <https://doi.org/10.1109/TDSC.2004.2>
- [5] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (2016), 35–41. <https://doi.org/10.1109/MS.2016.60>
- [6] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst.* 2, 1 (Feb. 1984), 39–59. <https://doi.org/10.1145/2080.357392>
- [7] Robert T. Braden. 1989. *Requirements for Internet Hosts—Communication Layers*. Internet Standard RFC 1122. RFC Editor. <https://doi.org/10.17487/RFC1122>
- [8] João Carreira, Henrique Madeira, and João Gabriel Silva. 1998. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering* 24, 2 (1998), 125–136. <https://doi.org/10.1109/32.666826>
- [9] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why Do Multi-Agent LLM Systems Fail?. In *Advances in Neural Information Processing Systems*, Vol. 38. <https://doi.org/10.52202/085713-4082>
- [10] DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348* (2026).
- [11] Wesley M. Eddy. 2022. *Transmission Control Protocol (TCP)*. Internet Standard RFC 9293. RFC Editor. <https://doi.org/10.17487/RFC9293>
- [12] Lars Eggert and Fernando Gont. 2009. *TCP User Timeout Option*. RFC 5482. RFC Editor. <https://doi.org/10.17487/RFC5482>
- [13] Gorry Fairhurst and Lloyd Wood. 2002. *Advice to Link Designers on Link Automatic Repeat reQuest (ARQ)*. RFC 3366. RFC Editor. <https://doi.org/10.17487/RFC3366>
- [14] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. *HTTP/1.1*. RFC 9112. RFC Editor. <https://doi.org/10.17487/RFC9112>
- [15] The Linux Foundation. 2026. Agent2Agent. <https://a2a-protocol.org>.
- [16] gRPC Authors. 2025. gRPC Deadlines Guide. <https://grpc.io/docs/guides/deadlines/>. Accessed 2026-08-28.
- [17] gRPC Authors. 2025. gRPC Retry Guide. <https://grpc.io/docs/guides/retry/>. Accessed 2026-08-28.
- [18] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruva Borthakur. 2011. {FATE} and {DESTINI}: A framework for cloud recovery testing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.
- [19] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model Based Multi-agents: A Survey of Progress and Challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*. 8048–8057. <https://doi.org/10.24963/ijcai.2024/890>
- [20] Aayush Gupta. 2026. ReliabilityBench: Evaluating LLM agent reliability under production-like stress conditions. *arXiv preprint arXiv:2601.06112* (2026).
- [21] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5, Article 124 (2025), 30 pages. <https://doi.org/10.1145/3712003>

- [22] Pengfei He, Yuping Lin, Shen Dong, Han Xu, Yue Xing, and Hui Liu. 2025. Red-teaming llm multi-agent systems via communication attacks. In *Findings of the association for computational linguistics: ACL 2025*. 6726–6747.
- [23] Stephen Hemminger. 2005. Network Emulation with NetEm. In *Proceedings of the 6th Australian National Linux Conference*. Canberra, Australia.
- [24] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 57–66. <https://doi.org/10.1109/ICDCS.2016.11>
- [25] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In *International Conference on Learning Representations*. https://proceedings.iclr.cc/paper_files/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract-Conference.html
- [26] International Organization for Standardization. 1994. *Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*. International Standard ISO/IEC 7498-1:1994. ISO/IEC. <https://www.iso.org/standard/20269.html>
- [27] Jin Jia, Zhiling Deng, Zhuangbin Chen, Yingqi Wang, and Zibin Zheng. 2026. Mas-fire: Fault injection and reliability evaluation for llm-based multi-agent systems. *arXiv preprint arXiv:2602.19843* (2026).
- [28] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations*. <https://arxiv.org/abs/2310.06770>
- [29] Phil Karn, Carsten Bormann, Gorry Fairhurst, Dan Grossman, Reiner Ludwig, Jamshid Mahdavi, Gabriel Montenegro, Joe Touch, and Lloyd Wood. 2004. *Advice for Internet Subnetwork Designers*. RFC 3819. RFC Editor. <https://doi.org/10.17487/RFC3819>
- [30] Kevin Lahey. 2000. *TCP Problems with Path MTU Discovery*. RFC 2923. RFC Editor. <https://doi.org/10.17487/RFC2923>
- [31] Ruichao Liang, Le Yin, Jing Chen, Yebo Feng, Cong Wu, Xiaoyu Zhang, Huangpeng Gu, Zijian Zhang, and Yang Liu. 2025. Don't Trust Your Upstream: Exploiting LLM Multi-Agent System via Topology-Guided Adversarial Propagation. *arXiv e-prints*, Article arXiv:2512.04129 (Dec. 2025), arXiv:2512.04129 pages. <https://doi.org/10.48550/arXiv.2512.04129> [cs.CR]
- [32] Linux Kernel Documentation. 2011. Linux Ethernet Bonding Driver HOWTO. <https://docs.kernel.org/networking/bonding.html> Latest documented update: 27 April 2011; accessed 2026-09-12.
- [33] Linux Kernel Documentation. 2026. Error Detection and Correction (EDAC) Devices. <https://docs.kernel.org/driver-api/edac.html> Accessed 2026-09-12.
- [34] David L. Mills, Jim Martin, Jack Burbank, and William Kasch. 2010. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. RFC Editor. <https://doi.org/10.17487/RFC5905>
- [35] Paul Mockapetris. 1987. *Domain Names—Implementation and Specification*. RFC 1035. RFC Editor. <https://doi.org/10.17487/RFC1035>
- [36] Model Context Protocol. 2026. What is the Model Context Protocol (MCP)? <https://modelcontextprotocol.io>.
- [37] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Bangkok, Thailand, 15174–15186. <https://doi.org/10.18653/v1/2024.acl-long.810>
- [38] Qwen Team. 2026. Qwen3.5: Towards Native Multimodal Agents. <https://qwen.ai/blog?id=qwen3.5>
- [39] Qwen Team. 2026. Qwen3.8-Max: A New Bar for Coding and Cowork. <https://qwen.ai/blog?id=qwen3.8>
- [40] Jerome H Saltzer, David P Reed, and David D Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984), 277–288.
- [41] Zahra Seyedghorban, Egor Klimov, Arie van Deursen, Annibale Panichella, and Burcu Kulahcioglu Ozkan. 2026. Observability and Fault Injection for LLM-Based Multi-Agent Systems in Software Engineering. In *2026 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 211–215.
- [42] Jonathan Stone and Craig Partridge. 2000. When the CRC and TCP Checksum Disagree. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*. Association for Computing Machinery, 309–319. <https://doi.org/10.1145/347059.347561>
- [43] Gou Tan, Zhensu Sun, Jieke Shi, Ting Zhang, Zilong He, Qingfu Wu, Shuai Liang, Weifeng Sun, Junda He, Pengfei Chen, Chuanfu Zhang, Lwin Khin Shar, and David Lo. 2026. AgentChaos: Chaos Engineering for Agent Systems via Programmatic Fault Injection. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 13 pages. <https://doi.org/10.1145/3832783.3837437>

- [44] Kuncan Wang, Ziting Wang, Peizhuo Lv, Haoyang Li, Guoliang Li, Gao Cong, and Wei Dong. 2026. Data Agents Under Attack: Vulnerabilities in LLM-Driven Analytical Systems. *arXiv preprint arXiv:2606.08661* (2026). <https://doi.org/10.48550/arXiv.2606.08661>
- [45] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryan W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155* (2023). <https://doi.org/10.48550/arXiv.2308.08155>
- [46] Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. 2025. Theagentcompany: benchmarking llm agents on consequential real world tasks. *Advances in Neural Information Processing Systems* 38 (2025).
- [47] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025. Which Agent Causes Task Failures and When? On Automated Failure Attribution of LLM Multi-Agent Systems. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 76583–76599. <https://proceedings.mlr.press/v267/zhang25cq.html>
- [48] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *International Conference on Learning Representations*. <https://arxiv.org/abs/2307.13854>
- [49] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 47, 2 (2021), 243–260. <https://doi.org/10.1109/TSE.2018.2887384>
- [50] Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You. 2025. MultiAgentBench: Evaluating the Collaboration and Competition of LLM Agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vienna, Austria, 8580–8622. <https://doi.org/10.18653/v1/2025.acl-long.421>