

MA-Mesh: Resilient Multi-Agent Collaboration via Role-based Self-Healing

JunYong Lee*, Woo Kyung Kim*, Honguk Woo†

Department of Computer Science and Engineering, Sungkyunkwan University
{ljy7488, kwk2696, hwoo}@skku.edu

Abstract

Effective collaboration in multi-agent systems requires mechanisms for coordination and adaptability in dynamic and uncertain environments. However, existing multi-agent systems rarely model resilience explicitly, instead relying on ad-hoc output correction that limits systematic, runtime management of robustness. In this work, we propose Multi-Agent Mesh (MA-Mesh), a framework that constructs a role-based self-healing layer on top of a collaboration pattern. Specifically, MA-Mesh generates resilience-aware system specifications and synthesizes them into executable multi-agent policy code. To enhance resilience, we introduce two self-healing mechanisms: Cross-agent reflection, a mechanism that allows agents to evaluate not only their own outcomes but also the performance of their peers, enabling informed behavioral adjustments and collective policy refinement; Role-based failover, a strategy which dynamically reassigns roles and enables agents to substitute for one another when sub-tasks fail or agents underperform, ensuring continuity of task execution. Experimental results across diverse tasks (e.g., software development, tool use, scientific reasoning) demonstrate that MA-Mesh significantly outperforms existing multi-agent systems in both task performance and robustness under system uncertainty.

1 Introduction

Building on the success of single LLMs for individual tasks (Rozière et al., 2023; Zhang et al., 2024; Ahn et al., 2023), compound agentic systems composed of multiple LLM-based agents have emerged as a promising approach to scale these capabilities toward more complex, long-horizon tasks that demand collaboration and coordination (Wu et al., 2024). Early multi-agent systems were developed in a domain-specialized manner (Du et al., 2024;

Hong et al., 2024), with each system hand-crafted for a specific application. In parallel, a line of work has explored automating multi-agent orchestration, either by leveraging the parametric knowledge of LLMs (Chen et al., 2024b,a) or by learning to configure the coordination structures (Zhang et al., 2025a). However, these multi-agent systems are typically designed under nominal system conditions, which limits their *resilience* to partial system failures and internal execution uncertainty.

To address this, we draw inspiration from the cloud service mesh (Li et al., 2019) to automate the construction of resilient multi-agent systems that can be instantiated dynamically on top of a collaboration pattern. Specifically, we explicitly decouple the collaboration logic, which governs task execution, from the resilience policy, which materializes a role-based self-healing layer. The self-healing layer operates through two mechanisms: (i) cross-agent reflection, which evaluates an agent’s own outcomes or those of its peers against its role-specific requirements to detect performance degradation; (ii) role-based failover, which continuously monitors agent execution and substitutes an affected agent with a suitable alternative when reflection signals performance degradation or a response timeout occurs. Such a design directly parallels a service mesh, where sidecar proxies transparently provide a resilience layer to microservices.

To this end, we present the Multi-Agent Mesh (MA-Mesh) framework, which overlays a role-based self-healing layer on top of a collaboration pattern, yielding a resilient multi-agent system. Given a task, an LLM orchestrator generates a resilience-aware system specification in a structured format. The specification comprises collaboration logic, which is either a user-supplied or an orchestrator-generated workflow, and a cross-agent reflection mechanism, which specifies the peers to evaluate and the role-specific criteria for assessing their execution outcomes. The orchestrator

*Equal contribution.

†Corresponding author.

then synthesizes the executable code by instantiating pre-defined agent and resilient proxy classes from this specification. A resilient proxy is attached to each agent; collectively, these proxies realize the self-healing layer. The role-based failover mechanism is built into the resilient proxy class, which monitors agent execution and triggers recovery upon performance degradation (detected by cross-agent reflection) or response timeout. Once triggered, a dedicated meta-agent replaces the affected agent and reassigns its role, ensuring execution continuity without global re-orchestration.

In this way, the resulting resilient multi-agent system is constructed analogously to a service mesh architecture, comprising a control plane and a data plane. The resilience-aware system specification, together with the meta-agent responsible for runtime recovery upon agent failures, constitutes the control plane; the agent instances and their attached sidecar-like resilient proxies form the data plane.

Through evaluations on software development for GUI and web applications (Liu et al., 2025; Hu et al., 2025b), tool-use tasks (Mialon et al., 2024), and scientific reasoning (Rein et al., 2023), we demonstrate that MA-Mesh substantially improves the resilience of multi-agent systems. We evaluate under four system uncertainty scenarios (Netflix, 2012) that model partial and behavioral uncertainty: agent chaos, prompt chaos, I/O chaos, and compound chaos. Across these settings, MA-Mesh consistently outperforms the strongest baseline C+I (Huang et al., 2025), with an average accuracy gain of 32.51 percentage points (pp). We further evaluate detection recall (DR), the fraction of affected agents correctly identified by resilient proxies, and observe an average improvement of 0.36 in DR over C+I. Moreover, to validate MA-Mesh’s adaptability to arbitrary collaboration patterns, we overlay it on top of MetaGPT (Hong et al., 2024) and RTADev (Liu et al., 2025), observing an average accuracy gain of 43.74 pp and 29.42 pp, respectively (see Table 5). Lastly, we validate MA-Mesh on AIOpsLab (Chen et al., 2025), a cloud-native testbed, demonstrating its practical applicability beyond synthetic chaos scenarios. The contributions of our work are summarized as follows:

- We present the MA-Mesh framework to construct multi-agent systems resilient to partial system failures and behavioral uncertainty.
- We explicitly decouple collaboration logic from resilience policy in a structured specification,

from which we synthesize executable code, enabling a role-based self-healing layer to be overlaid on top of any collaboration pattern.

- We introduce resilient proxies to realize the self-healing layer, comprising cross-agent reflection for role-specific degradation detection and role-based failover for recovery.
- Through experiments on benchmarks including software development, tool use, and scientific reasoning, we demonstrate that MA-Mesh outperforms existing multi-agent systems under diverse system uncertainty scenarios.

2 Related Work

Multi-agent systems have been widely adopted across diverse domains, including software development (Hong et al., 2024; Liu et al., 2025), scientific discovery (Kim et al., 2025a; Schmidgall et al., 2025), and complex mathematical reasoning (Du et al., 2024; Lei et al., 2024). In parallel, several works (Chen et al., 2024b,a) automate the configuration of multi-agent systems, dynamically adapting agent roles and coordination structures via an LLM orchestrator. Yet, these approaches assume nominal operating conditions and rarely address system-level resilience. Recently, Net-Safe (Yu et al., 2025) analyzes which coordination structures are inherently robust under adversarial attacks, while C+I (Huang et al., 2025) explicitly handles errors through challenger and inspector agents. However, C+I remains limited to correction-based mitigation, lacking role-aware evaluation and re-configuration. In contrast, MA-Mesh provides a service-mesh-style framework that augments any collaboration pattern with two mechanisms: cross-agent reflection, whose evaluation scope and criteria are dynamically determined, and role-based failover, which reconfigures agents at runtime upon failures or performance degradation.

3 Problem Formulation

A multi-agent system Π executes a task $\mathcal{T}$ through collaboration among multiple LLM-based agents, each assigned a distinct role. Formally, Π is represented as a directed graph, i.e.,

$$\Pi = (\mathcal{V}, \mathcal{E}), \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}, \quad (1)$$

where $\mathcal{V}$ is the set of agents, and $\mathcal{E}$ is the set of directed communication edges.

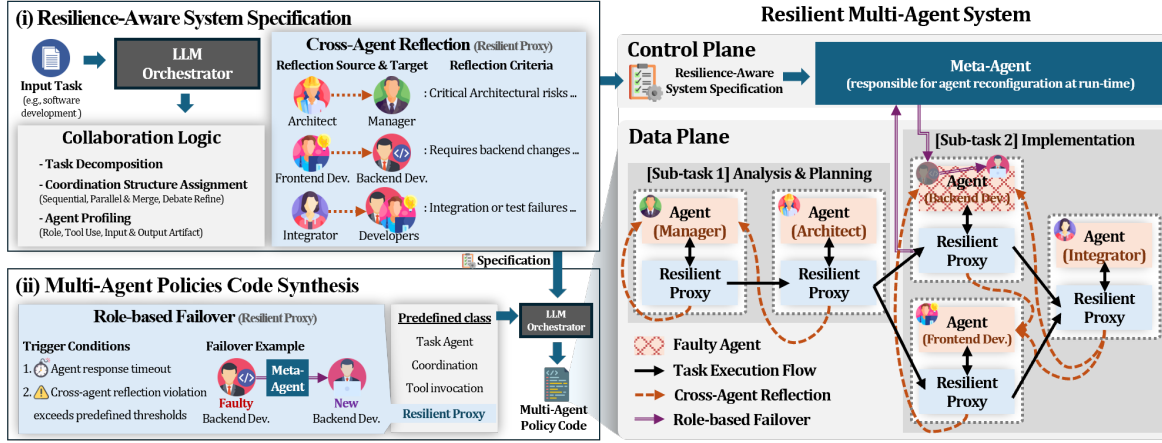


Figure 1: MA-Mesh Framework: In (i), given a task, the LLM orchestrator generates a resilience-aware system specification comprising the collaboration logic and the cross-agent reflection, which specifies the reflection targets and criteria. In (ii), the orchestrator synthesizes executable multi-agent policy code that incorporates role-based failover, including its trigger conditions and meta-agent-driven agent replacement. As a result, the resilience-aware system specification and the meta-agent responsible for failover recovery form the control plane, while the task-executing agents and their resilient proxies form the data plane, which enforces the resilient mechanisms at runtime.

In real-world deployments, multi-agent systems are exposed to diverse sources of system uncertainty, such as network or hardware failures, imperfect agent behavior, and adversarial interventions. To model such uncertainty, we adopt chaos engineering practices (Netflix, 2012; Chaos Mesh, 2021) which inject stochastic perturbations during system execution. Concretely, we consider three primitive classes of perturbations: *agent chaos*, which captures agent termination due to network or hardware faults; *prompt chaos*, which models errors in the agent prompts; and *I/O chaos*, which captures partial corruption of agent-generated outputs during transmission.

Then, the resilient agent collaboration problem is formalized through a framework $\Phi(\mathcal{T})$ that maps a task $\mathcal{T}$ to a multi-agent system Π , maximizing the task performance $\mathbf{P}(\Pi, \mathcal{T}, u) \in [0, 1]$ under system uncertainty $u \in \mathcal{U}$ as follows:

$$\Phi^* = \underset{\Phi}{\operatorname{argmax}} \mathbb{E}_{u \sim \mathcal{U}, \Pi \sim \Phi(\mathcal{T})} [\mathbf{P}(\Pi, \mathcal{T}, u)]. \quad (2)$$

4 Our Approach

4.1 Overall Framework

To automate the construction of resilient multi-agent systems, we present the MA-Mesh framework comprising two stages: (i) **resilience-aware system specification**, and (ii) **multi-agent policy code synthesis**, as illustrated on the left side of Figure 1. In (i), an LLM orchestrator derives a structured specification that consists of the collabo-

ration logic and the cross-agent reflection mechanism. The collaboration logic specifies the sub-task decomposition, the coordination structure, and the agent roles. As a default, it is dynamically generated by the orchestrator, but it can also be adopted from a predefined collaboration pattern of an existing static multi-agent system. Cross-agent reflection associates each agent with a resilient proxy that specifies the reflection target and criteria. At runtime, these proxies assess the outcomes produced by peer agents and provide feedback accordingly. In (ii), given the resulting specification, the orchestrator synthesizes executable multi-agent policy code by instantiating the agents and resilient proxies as parameterized classes. In doing so, the role-based failover mechanism is embedded into the resilient proxies, which trigger failover upon detecting runtime execution failures or suboptimal agent behavior, prompting the meta-agent to replace the affected agent.

Consequently, MA-Mesh constructs a service-mesh-like resilient multi-agent system, where the resilience-aware system specification and the meta-agent responsible for runtime agent replacement together form the **control plane**, while the agent instances and their resilient proxies constitute the **data plane**, as shown on the right side of Figure 1.

4.2 Resilience-Aware System Specification

Given a task $\mathcal{T}$, the LLM orchestrator ϕ_{LLM} derives a structured specification of the resilient multi-agent system in a form readily translatable into

executable code. As shown in Figure 2, the specification comprises two parts: the collaboration logic and the cross-agent reflection. The collaboration logic decomposes the task into K sub-tasks τ , each paired with a coordination structure c and a set of agent profiles $\mathcal{P}$. The cross-agent reflection $\mathcal{R}$ is defined over all agents in the system, assigning each agent a resilient proxy that specifies its reflection target and criteria. Together, they yield the resilience-aware system specification as,

$$\phi_{\text{LLM}} : \mathcal{T} \rightarrow \left(\{(\tau_k, c_k, \mathcal{P}_k)\}_{k=1}^K, \mathcal{R} \right). \quad (3)$$

Note that the collaboration logic, generated by the orchestrator by default, can instead be drawn from an existing static multi-agent system (see Table 5).

```
# Collaboration logic:  $\{(\tau_k, c_k, \mathcal{P}_k)\}_{k=1}^K$ 
sub_task: str = "Develop data management ..."
coordination_structure: str = "Parallel & Merge"
agent_profiles: list = [
    { "agent_name": "DataManagerDeveloper",
      "responsibility": "Implement manage.py ...",
      "tools": ["file write"] }, ... ]
# Repeated for  $k = 1, \dots, K$ 

# Resilient Proxies: Cross-Agent Reflection  $\mathcal{R}$ 
[ { "reflection_source": "UIArchitect",
    "reflection_targets": "DataManagerDeveloper",
    "reflection_criteria": (
        "1. Check manage.py implements all methods",
        "2. Check manage.py handles file I/O.",
        ...), ... ]
```

Figure 2: Resilience-aware system specification

Coordination structure. For each sub-task, the LLM orchestrator selects the coordination structure that best fits its objective from a pool $\mathcal{C} = \{ \text{'sequential'}, \text{'parallel\&merge'}, \text{'debate'}, \text{'refine'} \}$, adopted from the taxonomy of AutoGen (Wu et al., 2024). Detailed examples are shown in Figure 11.

Agent profile. Each agent profile specifies the agent’s role-related configuration, including responsibilities, permitted tools, input sources and modalities, and expected output artifacts and formats. The corresponding agent prompts are generated through separate LLM invocations.

Resilient proxy: cross-agent reflection. To enhance system resilience, we equip each agent with a resilient proxy. Each proxy encodes a cross-agent reflection mechanism that specifies (1) the reflection targets, which are the agents whose outputs it evaluates, and (2) reflection criteria, which capture both output quality and role-level expectations from the reflecting agent’s perspective. Note that a reflection target may belong to a different sub-task, and if an agent is selected by no other agent, its

resilient proxy performs self-reflection by default. Accordingly, the resulting resilient multi-agent system defined in Eq. (1) is extended as

$$\Pi = (\mathcal{V}, \mathcal{V}_r, \mathcal{E}), \quad (4)$$

$$\mathcal{E} \subseteq (\mathcal{V}_r \times \mathcal{V}_r) \cup (\mathcal{V}_r \times \mathcal{V}) \cup (\mathcal{V} \times \mathcal{V}_r),$$

where $\mathcal{V}_r$ denotes the set of resilient proxies. This design removes direct communication between agents $\mathcal{V}$ and routes all interactions through resilient proxies $\mathcal{V}_r$, ensuring that every outcome is intercepted for cross-agent reflection. When an outcome violates the reflection criteria, the proxy returns feedback prompting the agent to regenerate its output. We maintain a retry counter to track the number of reflection attempts; once it exceeds a predefined threshold, the role-based failover (see Section 4.3) is triggered to replace the agent.

4.3 Multi-Agent Policy Code Synthesis

Given the resilience-aware system specification, the LLM orchestrator translates it into executable code. We build on the AutoGen (Wu et al., 2024) library, pre-wrapping each task-executing agent and its associated resilient proxy into a class so that agent profiles and resilient proxy specifications can be instantiated through parameterized constructors.

```
class AgentWithResilientProxy:
    def __init__(self, agent_profile, proxy_spec):
        # task executing agent
        self.agent = create_agent(**agent_profile)
        # resilient proxy
        self.proxy = create_proxy(**proxy_spec)
    def __call__(self, request):
        # see Algorithm 1
    def cross_agent_reflection(self):
        # see Algorithm 2 in Appendix E
        # :return: feedback if error identified
    def role_based_failover(self):
        # see Algorithm 3 in Appendix E
        # :return: a new agent for replacement
```

Figure 3: Example code for agent with resilient proxy

Resilient proxy: role-based failover. The role-based failover mechanism is built into the resilient proxy class, so code synthesis simply instantiates proxies already equipped to detect failover conditions and initiate agent replacement. The replacement is handled by a dedicated meta-agent that does not itself participate in task execution. To replace an agent identified as faulty or degraded, the meta-agent initializes a new agent with a refined role specification. Role-based failover is triggered under two conditions. (i) The resilient proxy enforces a response timeout to regulate each

Algorithm 1 Procedure of Resilient Proxy

```
1: Input: Agent  $A$ , Proxy  $P$ , Request  $x$ 
2: Output: Response  $y$ 
3: for  $r = 0$  to  $\text{max\_retry}$  do
4:   try:
5:      $y \leftarrow \text{agent\_call}(A, x)$ 
6:   except Timeout:
7:     /* trigger failover due to response timeout */
8:      $A \leftarrow \text{role\_based\_failover}(P)$ 
9:     continue
10:   $n \leftarrow 0$ 
11:  while  $n < \text{max\_reflection\_attempts}$  do
12:    /* generate feedback via cross-agent reflection */
13:     $f \leftarrow \text{cross\_agent\_reflection}(P, x, y)$ 
14:    if  $f$  is None then
15:      return  $y$ 
16:    else
17:       $x \leftarrow \text{concat}(x, y, f); n \leftarrow n + 1$ 
18:      /* timeout handling omitted for brevity */
19:       $y \leftarrow \text{agent\_call}(A, x)$ 
20:    end if
21:  end while
22:  /* trigger failover due to violation of reflection */
23:   $A \leftarrow \text{role\_based\_failover}(P)$ 
24: end for
25: return  $y$ 
```

agent’s execution. If an agent fails to respond within this limit, the proxy triggers failover. (ii) Failover is also triggered when the cross-agent reflection repeatedly identifies that an agent fails to satisfy the specified reflection criteria, with the number of failed attempts exceeding a predefined retry threshold. As a result, role-based failover enables the system to dynamically recover from both runtime execution failures and suboptimal agent behaviors, ensuring resilient task execution without re-orchestrating the entire multi-agent system.

The overall procedure of an agent and its resilient proxy is summarized in Algorithm 1.

4.4 Resilient Multi-Agent System

Conceptually, the resulting resilient multi-agent system follows a service-mesh-like architecture, comprising a control plane and a data plane. The resilience-aware system specification serves as the control plane artifact, encoding collaboration logic and cross-agent reflection. At runtime, the meta-agent residing in the control plane leverages this specification to instantiate appropriate replacement agents, preserving the continuity of task execution.

The data plane consists of the agents and resilient proxies instantiated from the synthesized multi-agent code. The agents are responsible for task execution, while resilient proxies enforce the encoded role-based self-healing mechanisms by monitoring agent execution, applying timeout constraints, performing cross-agent reflection, and triggering role-based failover when necessary.

5 Experiments

5.1 Experiment Setup

Benchmarks. We evaluate on four benchmarks spanning three domains: (i) **software development** – GUI Dev. and Web Dev., which we adapt from Liu et al. (2025) and Hu et al. (2025b) into complex, long-horizon tasks; (ii) **tool use** – GAIA (Mialon et al., 2024); and (iii) **scientific reasoning** – GPQA-Diamond (Rein et al., 2023). Further details are provided in Appendix C.

System Uncertainty. We consider four types of system uncertainty: (i) **Agent Chaos** randomly terminates agents with a given probability (Basiri et al., 2016); (ii) **Prompt Chaos** perturbs agent prompts through replacement, deletion, and shuffling (Huang et al., 2025); (iii) **I/O Chaos** applies the same perturbations to agent outputs before they are consumed by other agents; and (iv) **Compound Chaos** applies all three system uncertainties.

Additionally, we evaluate on **adversarial attack** scenarios (Xie et al., 2025) in Table 3, where an external agent injects malicious content into role prompts, task prompts, or outputs to mislead agent behavior; and **noisy tool** scenarios (Wang et al., 2026) in Table 4, where tool operations suffer from execution failures and noisy responses.

Evaluation Metrics. To evaluate task performance, we report **Accuracy (Acc)**, the fraction of successfully completed tasks. We also report **Detection Recall (DR)**, the ratio of correctly identified affected agents to all affected agents.

Baselines. We compare MA-Mesh against four categories of baselines: (i) **single-agent method** – CoT (Wei et al., 2022); (ii) **static multi-agent systems** – MetaGPT (Hong et al., 2024), RTADev (Liu et al., 2025), and LLM-Debate (Du et al., 2024); (iii) **autonomous multi-agent systems** – AgentVerse (Chen et al., 2024b) and EvoMAC (Hu et al., 2025b); (iv) **resilient multi-agent systems** – C+I (Huang et al., 2025).

We use gpt-4.1-mini-0414 (OpenAI, 2025) as the LLM for all baselines and MA-Mesh. Addition-

Benchmark	Method	Vanilla	Agent Chaos		Prompt Chaos		I/O Chaos		Compound Chaos	
		Acc (%)	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR
GUI Dev.	CoT	66.02 $\pm$ 2.53	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	25.03 $\pm$ 5.87	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	3.96 $\pm$ 4.40	0.00 $\pm$ 0.00
	RTADev	66.30 $\pm$ 1.60	21.87 $\pm$ 8.35	0.61 $\pm$ 0.15	39.64 $\pm$ 3.27	0.42 $\pm$ 0.14	33.90 $\pm$ 13.38	0.19 $\pm$ 0.17	14.99 $\pm$ 8.29	0.43 $\pm$ 0.09
	MetaGPT	70.51 $\pm$ 4.22	7.39 $\pm$ 6.10	0.00 $\pm$ 0.00	21.62 $\pm$ 5.12	0.00 $\pm$ 0.00	22.63 $\pm$ 11.46	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
	C+I	68.05 $\pm$ 2.46	4.46 $\pm$ 4.47	0.24 $\pm$ 0.03	19.30 $\pm$ 6.23	0.29 $\pm$ 0.04	47.37 $\pm$ 4.24	0.86 $\pm$ 0.03	1.59 $\pm$ 2.75	0.30 $\pm$ 0.03
	AgentVerse	66.53 $\pm$ 6.59	32.87 $\pm$ 6.03	0.00 $\pm$ 0.00	44.61 $\pm$ 8.08	0.00 $\pm$ 0.00	37.89 $\pm$ 12.94	0.00 $\pm$ 0.00	26.36 $\pm$ 6.13	0.00 $\pm$ 0.00
	EvoMAC	72.79 $\pm$ 1.96	48.01 $\pm$ 11.28	0.00 $\pm$ 0.00	52.15 $\pm$ 8.34	0.00 $\pm$ 0.00	63.65 $\pm$ 7.45	0.00 $\pm$ 0.00	24.38 $\pm$ 11.92	0.00 $\pm$ 0.00
	MA-Mesh	75.29 $\pm$ 4.34	71.05 $\pm$ 1.64	1.00 $\pm$ 0.00	65.30 $\pm$ 5.48	0.84 $\pm$ 0.08	67.87 $\pm$ 5.07	0.82 $\pm$ 0.07	65.66 $\pm$ 5.69	0.90 $\pm$ 0.05
Web Dev.	CoT	45.73 $\pm$ 3.02	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.60 $\pm$ 1.04	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.19 $\pm$ 0.32	0.00 $\pm$ 0.00
	RTADev	40.17 $\pm$ 6.78	12.68 $\pm$ 4.60	0.44 $\pm$ 0.17	8.85 $\pm$ 0.53	0.46 $\pm$ 0.02	16.74 $\pm$ 4.50	0.20 $\pm$ 0.11	2.68 $\pm$ 4.12	0.36 $\pm$ 0.06
	MetaGPT	42.09 $\pm$ 7.03	4.81 $\pm$ 2.77	0.00 $\pm$ 0.00	6.30 $\pm$ 5.03	0.00 $\pm$ 0.00	11.91 $\pm$ 3.30	0.00 $\pm$ 0.00	0.41 $\pm$ 0.35	0.00 $\pm$ 0.00
	C+I	45.25 $\pm$ 3.67	5.72 $\pm$ 2.81	0.10 $\pm$ 0.14	12.47 $\pm$ 9.17	0.31 $\pm$ 0.07	14.03 $\pm$ 0.39	0.75 $\pm$ 0.10	0.00 $\pm$ 0.00	0.25 $\pm$ 0.04
	AgentVerse	52.01 $\pm$ 2.06	11.95 $\pm$ 6.38	0.00 $\pm$ 0.00	21.55 $\pm$ 8.06	0.00 $\pm$ 0.00	12.50 $\pm$ 1.38	0.00 $\pm$ 0.00	18.38 $\pm$ 3.77	0.00 $\pm$ 0.00
	EvoMAC	41.33 $\pm$ 4.47	20.45 $\pm$ 4.72	0.00 $\pm$ 0.00	24.29 $\pm$ 2.56	0.00 $\pm$ 0.00	14.72 $\pm$ 7.17	0.00 $\pm$ 0.00	5.80 $\pm$ 5.10	0.00 $\pm$ 0.00
	MA-Mesh	59.68 $\pm$ 1.84	53.98 $\pm$ 1.55	1.00 $\pm$ 0.00	53.11 $\pm$ 1.82	0.94 $\pm$ 0.04	54.12 $\pm$ 1.58	0.75 $\pm$ 0.13	49.76 $\pm$ 0.32	0.91 $\pm$ 0.02
GAIA	CoT	28.61 $\pm$ 3.57	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	1.01 $\pm$ 0.35	0.00 $\pm$ 0.00	4.85 $\pm$ 1.60	0.00 $\pm$ 0.00	0.61 $\pm$ 0.61	0.00 $\pm$ 0.00
	LLM-Debate	32.32 $\pm$ 1.85	13.13 $\pm$ 2.45	0.00 $\pm$ 0.00	19.39 $\pm$ 2.10	0.00 $\pm$ 0.00	15.96 $\pm$ 3.45	0.00 $\pm$ 0.00	5.25 $\pm$ 0.93	0.00 $\pm$ 0.00
	C+I	30.10 $\pm$ 0.92	14.75 $\pm$ 3.11	0.53 $\pm$ 0.06	20.81 $\pm$ 1.95	0.62 $\pm$ 0.04	17.58 $\pm$ 5.97	0.99 $\pm$ 0.00	3.70 $\pm$ 1.91	0.72 $\pm$ 0.09
	AgentVerse	24.31 $\pm$ 2.38	13.18 $\pm$ 1.45	0.00 $\pm$ 0.00	7.07 $\pm$ 1.75	0.00 $\pm$ 0.00	11.31 $\pm$ 5.44	0.00 $\pm$ 0.00	7.68 $\pm$ 6.48	0.00 $\pm$ 0.00
	MA-Mesh	32.73 $\pm$ 1.60	29.90 $\pm$ 4.94	1.00 $\pm$ 0.00	26.46 $\pm$ 3.55	0.83 $\pm$ 0.02	29.09 $\pm$ 0.61	0.92 $\pm$ 0.01	24.44 $\pm$ 4.59	0.94 $\pm$ 0.01
GPQA Diamond	CoT	64.14 $\pm$ 1.01	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	27.94 $\pm$ 0.58	0.00 $\pm$ 0.00	30.97 $\pm$ 2.10	0.00 $\pm$ 0.00	20.04 $\pm$ 1.54	0.00 $\pm$ 0.00
	LLM-Debate	66.50 $\pm$ 1.27	26.77 $\pm$ 7.02	0.00 $\pm$ 0.00	35.52 $\pm$ 7.65	0.00 $\pm$ 0.00	41.41 $\pm$ 4.01	0.00 $\pm$ 0.00	15.32 $\pm$ 3.44	0.00 $\pm$ 0.00
	C+I	65.15 $\pm$ 1.33	35.35 $\pm$ 3.81	0.66 $\pm$ 0.09	52.36 $\pm$ 2.04	0.52 $\pm$ 0.02	51.67 $\pm$ 4.22	0.96 $\pm$ 0.03	16.33 $\pm$ 2.49	0.66 $\pm$ 0.22
	AgentVerse	62.46 $\pm$ 2.59	28.62 $\pm$ 2.96	0.00 $\pm$ 0.00	31.14 $\pm$ 3.04	0.00 $\pm$ 0.00	27.95 $\pm$ 11.21	0.00 $\pm$ 0.00	22.90 $\pm$ 6.11	0.00 $\pm$ 0.00
	MA-Mesh	67.68 $\pm$ 2.20	61.87 $\pm$ 1.07	1.00 $\pm$ 0.00	62.46 $\pm$ 3.79	0.81 $\pm$ 0.02	61.11 $\pm$ 2.53	0.98 $\pm$ 0.01	61.45 $\pm$ 3.04	0.96 $\pm$ 0.01

Table 1: Performance under system uncertainty: The baselines and MA-Mesh are evaluated on the GUI Dev., Web Dev., GAIA, and GPQA-Diamond benchmarks. Each method is run for three trials, and results are averaged across trials. The best performance for each benchmark is highlighted in **bold**.

ally, we utilize reasoning models of o4-mini and gemini-3-flash (Google, 2025) for ablation.

5.2 Main Results

Table 1 reports the performance of MA-Mesh and the baselines across four benchmarks, under both the vanilla setting (i.e., nominal system conditions) and the four system uncertainty scenarios. Under the vanilla setting, MA-Mesh yields robust performance across all benchmarks, achieving an average improvement of 6.71 pp in accuracy over the most competitive baseline C+I. This improvement is primarily attributed to cross-agent reflection, through which resilient proxies identify suboptimal intermediate outcomes and proactively trigger agent replacement, thereby preventing error propagation.

Across the four system uncertainty scenarios, MA-Mesh consistently achieves the highest performance, outperforming C+I by 39.13 pp, 25.60 pp, 20.39 pp, and 44.92 pp in average accuracy, respectively. Further, MA-Mesh attains substantially higher DR than C+I, except under the I/O chaos, indicating that MA-Mesh correctly identifies affected agents. As CoT, MetaGPT, LLM-Debate, AgentVerse, and EvoMAC do not explicitly incorporate resilience mechanisms (e.g., the self-correction in C+I), we report their DR as 0.0.

In these experiments, CoT yields zero accuracy under agent chaos, since any single-point failure

directly terminates task execution. In contrast, multi-agent systems are generally more robust, as failures of individual agents can be partially compensated for by other agents. Yet, MetaGPT, LLM-Debate, and EvoMAC show limited performance, lacking explicit mechanisms to detect and recover from system uncertainty. C+I and RTADev improve performance by incorporating self-correction: C+I introduces dedicated inspector and challenger agents to detect and correct erroneous outputs, while RTADev evaluates each new outcome against prior ones to refine faulty outcomes and maintain alignment across execution steps. Unlike these correction-based mechanisms, MA-Mesh performs cross-agent reflection to assess agents from the perspectives of their peers and addresses failures by replacing agents through role-based failover. As a result, the resilient proxies in MA-Mesh function as infrastructure-level resilience mechanisms that mitigate error propagation and enforce role-based recovery.

5.3 Analysis

Varying Chaos Conditions. Table 2 reports the performance of RTADev, C+I, and MA-Mesh under varying chaos conditions on both software development benchmarks. For Agent Chaos, we vary its severity via termination ratios of 20%, 40%, and 60%. As the termination ratio increases, ac-

Method		Agent Chaos						Prompt Chaos						I/O Chaos					
		20%		40%		60%		Replace		Delete		Shuffle		Replace		Delete		Shuffle	
		Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR
GUI Dev.	RTADev	20.98	0.62	14.12	0.72	9.68	0.34	33.53	0.41	37.18	0.33	29.76	0.40	51.65	0.19	40.18	0.23	54.68	0.10
	C+I	4.37	0.38	0.00	0.31	0.00	0.35	9.52	0.16	17.80	0.52	23.60	0.33	39.54	0.79	54.55	0.93	46.71	0.71
	MA-Mesh	66.25	1.00	68.60	1.00	68.35	1.00	59.73	0.60	54.73	0.62	52.89	0.72	52.02	0.82	60.77	0.83	66.45	0.58
Web Dev.	RTADev	12.56	0.48	5.86	0.49	5.14	0.34	14.76	0.49	5.00	0.31	4.53	0.41	9.47	0.06	10.71	0.34	13.28	0.20
	C+I	5.44	0.47	2.50	0.26	0.00	0.18	12.58	0.28	15.90	0.42	8.57	0.05	20.29	0.68	23.01	0.78	3.89	0.56
	MA-Mesh	60.05	1.00	60.43	1.00	53.57	1.00	52.22	0.95	51.86	1.00	46.11	0.93	53.55	0.55	54.40	0.65	50.89	0.82

Table 2: Performance under various chaos conditions

curacy degrades sharply for the baselines, while MA-Mesh maintains robust performance across all severity levels. This is due to role-based failover in MA-Mesh, which detects failures through timeouts and replaces the affected agents. For Prompt Chaos, we apply three perturbation types, namely random word replacement, deletion, and shuffling, and MA-Mesh consistently outperforms the baselines in all cases. Since agents are not merely response generators but actors with explicitly assigned roles, correction-based baselines are less effective under prompt corruption. In contrast, MA-Mesh explicitly assesses peer outcomes with role-aware reflection criteria and recovers compromised agents via role-based failover, thereby showing improved performance. For I/O Chaos, we apply the same perturbations as Prompt Chaos, and MA-Mesh again outperforms the baselines. C+I occasionally attains higher DR than MA-Mesh because its correction-based strategy over-detects affected agents. When an output is perturbed, part of its semantic content remains intact; MA-Mesh’s cross-agent reflection filters out such cases, while C+I still flags them as failures. As a result, although C+I exhibits higher DR in some settings, this gain comes at the cost of substantially lower accuracy than MA-Mesh.

Adversarial Attack. In Table 3, we model a malicious agent that crafts prompts or outputs to mislead agent behavior and disrupt coordination (Xie et al., 2025). These attack scenarios target the semantic intent of role prompts, task prompts, or agent-generated outputs, steering agents toward adversarial behaviors. As shown, MA-Mesh maintains robust performance in all cases and outperforms C+I by 45.02 pp in average accuracy, as cross-agent reflection detects semantic inconsistencies with assigned roles and role-based failover recovers affected agents.

Noisy Tool Scenarios. Since multi-agent systems often utilize external tools for task completion, we evaluate performance on GAIA under two noisy

Method		Role Prompt		Task Prompt		Output	
		Acc (%)	DR	Acc (%)	DR	Acc (%)	DR
GUI Dev.	RTADev	28.64	0.66	48.82	0.40	20.29	0.33
	C+I	6.80	0.24	8.52	0.24	4.44	0.46
	MA-Mesh	63.95	0.93	52.98	0.52	52.14	0.56
Web Dev.	RTADev	11.19	0.59	18.22	0.39	7.56	0.45
	C+I	15.14	0.25	6.11	0.10	4.13	0.44
	MA-Mesh	48.00	0.91	53.77	0.92	44.39	0.60

Table 3: Performance w.r.t. adversarial attacks

tool scenarios, i.e., execution failure and noisy response (Wang et al., 2026). As shown in Table 4, MA-Mesh outperforms C+I by 4.55 pp in average accuracy, highlighting that MA-Mesh remains resilient under tool failures that may be encountered in practical deployment.

Method	Vanilla	Execution Failure		Noisy Response	
	Acc (%)	Acc (%)	DR	Acc (%)	DR
LLM-Debate	32.32	15.76	0.00	18.79	0.00
C+I	30.10	22.42	0.50	23.64	0.49
MA-Mesh	32.73	27.88	0.99	27.27	0.89

Table 4: Performance w.r.t. noisy tool scenarios

Augmenting Existing Methods with MA-Mesh.

In Table 5, we verify that MA-Mesh can be layered onto any existing multi-agent collaboration pattern. Specifically, we apply MA-Mesh on top of the collaboration patterns of MetaGPT and RTADev, improving their performance by 43.74 pp and 29.42 pp on average, respectively. This is enabled by MA-Mesh’s code-based representation, which translates any existing collaboration pattern into a resilience-aware specification, and synthesizes it into executable code.

Method	Agent Chaos		Prompt Chaos		I/O Chaos	
	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR
MetaGPT	7.39	0.00	21.62	0.00	22.63	0.00
+ MA-Mesh	66.86	1.00	56.90	0.88	59.09	0.89
RTADev	21.87	0.61	39.64	0.42	33.90	0.19
+ MA-Mesh	62.13	1.00	56.88	0.75	64.65	0.91

Table 5: Augmenting existing methods with MA-Mesh

Cost Analysis. Table 6 reports the API cost and wall-clock time of MA-Mesh and the baselines, separated into the cost of agents and resilient components. For RTADev, which has no resilience mechanism, only agent cost is reported. Considering only the resilience components, MA-Mesh incurs less than one-half the API cost of C+I. Even in total cost, MA-Mesh is \$0.01 lower than RTADev, indicating that the resilient proxies add minimal overhead. Moreover, MA-Mesh is time efficient, as 23% of its coordination structures adopt the parallel & merge pattern (see Figure 12), enabling concurrent execution unlike the predominantly sequential execution employed by the baselines. Further detailed cost analysis is provided in Appendix F.1.

Method	API Cost (\$)			Wall-clock Time (min)		
	Agent	Resilient	Total	Agent	Resilient	Total
RTADev	0.180	–	0.180	16.26	–	16.26
C+I	0.049	0.155	0.204	5.12	3.14	8.26
MA-Mesh	0.114	0.055	0.170	3.76	0.77	4.53

Table 6: Inference cost analysis

5.4 Ablation Study

Ablation on Resilient Proxy. Table 7 presents the impact of removing cross-agent reflection, role-based failover, and the full resilient proxy on the Web Dev., with all variants using the same synthesized workflow as MA-Mesh. Even under the vanilla setting, removing these components degrades performance, since MA-Mesh detects and mitigates errors from both suboptimal role assignment by the orchestrator and hallucinated agent outputs. The degradation becomes more pronounced under Compound Chaos. In particular, removing the full resilient proxy reduces accuracy to zero, since no mechanism remains to recover faulty agents, highlighting the effectiveness of MA-Mesh. Further ablation results for the resilient proxy are provided in Appendix F.4.

Method	Vanilla	Compound Chaos	
	Acc (%)	Acc (%)	DR
MA-Mesh	59.68	49.76	0.91
– Cross-agent Reflection	49.89	17.57	0.57
– Role-based Failover	52.50	4.13	0.35
– Resilient Proxy	46.06	0.00	0.00

Table 7: Ablation on resilient proxy in MA-Mesh

Ablation on LLM. In Table 8, we vary the LLM used for agents and resilient proxies, adopting reasoning-oriented LLMs (e.g., o4-mini,

gemini-3-flash) evaluated on GUI Dev. These models consistently improve performance across all cases. The gains appear even when applied only to resilient proxies, indicating that stronger reasoning enables more precise, role-aware feedback.

Target	LLM	Prompt Chaos		I/O Chaos	
		Acc (%)	DR	Acc (%)	DR
Task Agent	gpt-4.1-mini	65.30	0.84	67.87	0.82
	o4-mini	78.91	0.76	77.08	0.85
	gemini-3-flash	73.56	0.94	79.08	1.00
Resilient Proxy	gpt-4.1-mini	65.30	0.84	67.87	0.82
	o4-mini	71.64	0.91	79.11	1.00
	gemini-3-flash	69.50	0.71	85.81	0.94

Table 8: Ablation on LLM used in MA-Mesh

5.5 Case Study on Cloud-Native Testbed

To validate MA-Mesh beyond synthetic chaos scenarios, we use AIOpsLab (Chen et al., 2025), a cloud-native testbed, to reproduce infrastructure-level system uncertainty that arises in cloud environments. Specifically, we instantiate each agent as a separate Kubernetes pod and use Chaos Mesh (Chaos Mesh, 2021) to inject Stress Chaos, Disk I/O Chaos, and Network Chaos, which stress CPU and memory, disk access, and pod-to-pod communication, respectively. As shown in Table 9, MA-Mesh remains effective under such realistic chaos settings, as its service-mesh-inspired design aligns naturally with cloud-native infrastructure.

Method	Stress Chaos		Disk I/O Chaos		Network Chaos	
	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR
LLM-Debate	38.89	0.00	31.82	0.00	37.37	0.00
C+I	47.98	0.70	34.34	0.79	43.94	0.46
MA-Mesh	55.56	1.00	60.61	1.00	56.06	1.00

Table 9: Performance on cloud-native testbed

6 Conclusion

In this work, we presented the MA-Mesh framework for constructing resilient multi-agent systems that systematically address partial failures and behavioral uncertainty. By decoupling collaboration logic from resilience policy in a structured specification, MA-Mesh synthesizes executable code that overlays a role-based self-healing layer on top of a collaboration pattern. This layer is realized through resilient proxies which integrate cross-agent reflection for role-specific degradation detection with role-based failover for recovery. Through extensive experiments on diverse system uncertainty

scenarios, together with a case study on a cloud-native testbed, we demonstrate that MA-Mesh consistently outperforms existing multi-agent systems in both task performance and resilience.

Limitations

As shown in Table 8, employing more capable reasoning models solely within the resilient proxies consistently improves performance. In the current design, reasoning models are assigned statically, regardless of task difficulty or runtime conditions. Allocating them dynamically, invoking stronger reasoning only when needed, remains an important direction for future work. Another limitation is the assumption that the resilient proxies and the meta-agent remain available throughout execution; such failures fall outside the current threat model and are delegated to infrastructure-level fault tolerance.

Ethical Considerations

This work seeks to characterize and reduce failure modes in LLM-based multi-agent collaboration, enabling more robust orchestration. The chaos engineering techniques introduced in this study serve as experimental tools to evaluate the resilience of multi-agent systems through stochastic perturbations that simulate system uncertainty. Although these techniques could potentially be exploited to disrupt agent behaviors or corrupt outputs, such risks are mitigated by resilience mechanisms designed to detect uncertainty. To ensure transparency and reproducibility, a detailed description of each chaos scenario is provided in Appendix D.1. These techniques are intended solely for defensive evaluation and academic research, and should not be used to enable malicious interference.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT), (RS-2022-II220043, Adaptive Personality for Intelligent Agents, RS-2022-II221045, Self-directed multi-modal Intelligence for solving unknown, open domain problems, RS-2025-02218768, Accelerated Insight Reasoning via Continual Learning, RS-2025-25442569, AI Star Fellowship Support Program (Sungkyunkwan Univ.)), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2026-25474409), IITP-ITRC

(Information Technology Research Center) grant funded by the Korea government (MSIT) (IITP-2026-RS-2024-00437633, 10%), and by Samsung Electronics Co., Ltd.

References

- Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, and 26 others. 2023. Do as I can, not as I say: Grounding language in robotic affordances. In *Proceedings of the 6th Conference on Robot Learning*.
- Anthropic. 2026. Claude Code. <https://code.claude.com/>.
- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos engineering. *IEEE Software*, 33(3):35–41.
- Chaos Mesh. 2021. Chaos Mesh: A chaos engineering platform for kubernetes. <https://github.com/chaos-mesh/chaos-mesh>.
- Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje Karlsson, Jie Fu, and Yemin Shi. 2024a. Autoagents: A framework for automatic agent generation. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024b. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *Proceedings of the 12th International Conference on Learning Representations*.
- Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. Aiopslab: A holistic framework to evaluate ai

- agents for enabling autonomous clouds. In *Proceedings of Machine Learning and Systems*.
- Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*.
- Google. 2025. Gemini 3 flash: frontier intelligence built for speed. <https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/>.
- 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*.
- Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, and Trent Swanson. 2014. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft.
- 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 *Proceedings of the 12th International Conference on Learning Representations*.
- Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Ziyu Ye, Bowei Xia, Tao Sun, Zhaoxuan Jin, Yingru Li, Zeyu Zhang, Yifeng Wang, Qianshuo Ye, Bernard Ghanem, Ping Luo, and Guohao Li. 2025a. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. In *Proceedings of the 39th Conference on Neural Information Processing Systems*.
- Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2025b. Self-evolving multi-agent collaboration networks for software development. In *Proceedings of the 13th International Conference on Learning Representations*.
- Jen-tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Michael R. Lyu, and Maarten Sap. 2025. On the resilience of llm-based multi-agent collaboration with faulty agents. In *Proceedings of the 42nd International Conference on Machine Learning*.
- Hyomin Kim, Yunhui Jang, and Sungsoo Ahn. 2025a. Mt-mol: Multi agent system with tool-based reasoning for molecular optimization. In *Findings of the Association for Computational Linguistics: EMNLP*.
- Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. 2025b. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*.
- Bin Lei, Yi Zhang, Shan Zuo, Ali Payani, and Caiwen Ding. 2024. MACM: utilizing a multi-agent system for condition mining in solving complex mathematical problems. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems*.
- Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for 'mind' exploration of large language model society. In *Proceedings of the 37th Conference on Neural Information Processing Systems*.
- Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service mesh: Challenges, state of the art, and future research opportunities. In *Proceedings of the 2019 IEEE International Conference on Service-Oriented System Engineering*.
- Jie Liu, Guohua Wang, Ronghui Yang, Jiajie Zeng, Mengchen Zhao, and Yi Cai. 2025. RTADev: Intention aligned multi-agent framework for software development. In *Findings of the Association for Computational Linguistics: ACL*.
- Grégoire Mialon, Clémentine Fourier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: A benchmark for general AI assistants. In *International Conference on Learning Representations*.
- Kief Morris. 2016. *Infrastructure as code: managing servers in the cloud*. O'Reilly Media, Inc.
- Netflix. 2012. Chaos Monkey. <https://github.com/Netflix/chaosmonkey>.
- OpenAI. 2025. Introducing GPT-4.1 in the API. <https://openai.com/index/gpt-4-1/>.
- David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2023. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. 2025. Agent laboratory: Using LLM agents as research assistants. In *Findings of the Association for Computational Linguistics: EMNLP*.

- Tarik Taleb, Adlen Ksentini, and Bruno Sericola. 2016. On service resilience in cloud-native 5g mobile systems. *IEEE Journal on Selected Areas in Communications*, 34(3):483–496.
- Ruipeng Wang, Yuxin Chen, Yukai Wang, Chang Wu, Junfeng Fang, Xiaodong Cai, Qi GU, Hui Su, An Zhang, Xiang Wang, Xunliang Cai, and Tat-Seng Chua. 2026. Agentnoisebench: Benchmarking robustness of tool-using llm agents under noisy condition. In *Forty-third International Conference on Machine Learning*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems*.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.
- Yizhe Xie, Congcong Zhu, Xinyue Zhang, Tianqing Zhu, Dayong Ye, Minghao Wang, and Chi Liu. 2025. Who’s the mole? modeling and detecting intention-hiding malicious agents in llm-based multi-agent systems. *arXiv preprint arXiv:2507.04724*.
- Miao Yu, Shilong Wang, Guibin Zhang, Junyuan Mao, Chenlong Yin, Qijiong Liu, Kun Wang, Qingsong Wen, and Yang Wang. 2025. NetSafe: Exploring the topological safety of multi-agent system. In *Findings of the Association for Computational Linguistics: ACL*.
- Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025a. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning*.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025b. Aflow: Automating agentic workflow generation. In *Proceedings of the 13th International Conference on Learning Representations*.
- Zhuosheng Zhang, Aston Zhang, Mu Li, Hai Zhao, George Karypis, and Alex Smola. 2024. Multi-modal chain-of-thought reasoning in language models. *Transactions on Machine Learning Research*.

A Related Work

Multi-agent System. LLM-based multi-agent systems have emerged as an effective paradigm for solving complex tasks by assigning specialized roles to multiple agents and coordinating their interactions. Existing systems often encode such collaboration through manually designed roles, task handoffs, and communication protocols. LLM-Debate (Du et al., 2024) adopts a solver-judge debate structure, where solver agents independently generate responses, iteratively critique one another, and a judge agent synthesizes the final answer. MetaGPT (Hong et al., 2024) formalizes software development as an SOP-based workflow, where fixed roles such as product manager, architect, engineer, and QA engineer sequentially exchange artifacts across development stages. RTADev (Liu et al., 2025) extends this static development workflow with intention-alignment checks, where reviewer agents inspect intermediate artifacts and trigger discussion when downstream outputs diverge from prior role intents. While this role-based design enables structured collaboration, static coordination structures make it difficult to adapt the system beyond predefined task scopes. Accordingly, AgentVerse (Chen et al., 2024b) automates agent-team coordination through dynamic expert recruitment, collaborative decision-making, action execution, and evaluation. AutoAgents (Chen et al., 2024a) dynamically generates specialized agents from task content and coordinates them through task-specific planning. AFlow (Zhang et al., 2025b) represents agentic workflows as executable code and optimizes them through Monte Carlo Tree Search with execution feedback. MaAS (Zhang et al., 2025a) searches over an agentic supernet to sample query-dependent multi-agent architectures with task-specific resource allocation. EvoMAC (Hu et al., 2025b) self-evolves multi-agent collaboration networks by converting execution feedback into textual gradients that update agent prompts, connections, and workflows during test-time execution. However, these methods assume nominal system conditions and do not explicitly address failures or behavioral uncertainty during runtime. In contrast, MA-Mesh extends a collaboration pattern with resilience policies for runtime monitoring and recovery under system uncertainty.

Cloud Native Resilience. Resilience has long been a fundamental design consideration in large-scale cloud and distributed systems (Basiri et al.,

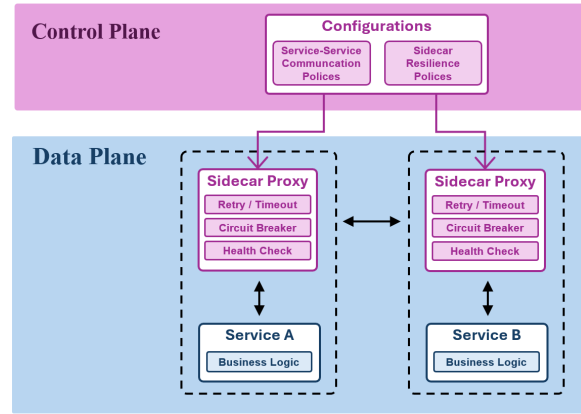


Figure 4: Cloud-Native Service Mesh

2016; Taleb et al., 2016). In particular, the *service mesh* (Li et al., 2019) has emerged as a principled architectural approach to achieve system-level resilience by decoupling resilience logic from microservice applications. The service mesh explicitly separates a control plane, which manages resilience policies through declarative configurations, from a data plane, where sidecar proxies attached to microservices intercept service-to-service communication and provide fault-tolerance mechanisms such as retries, timeouts, and circuit breaking (Homer et al., 2014). MA-Mesh adapts this cloud-native principle to multi-agent systems through an infrastructure-as-code (IaC) approach (Morris, 2016). In this sense, the artifact constructed through IaC is a service-mesh-style infrastructure, where synthesized policy code instantiates sidecar-like resilient proxies around task-executing agents. This enables infrastructure-level enforcement of resilience policies for multi-agent systems, without embedding resilience logic directly into agent implementations.

B Service Mesh-inspired MA-Mesh

Large-scale applications are often deployed as cloud-native microservices, where inter-service dependencies allow partial failures to propagate across downstream services. Such failure propagation motivates the evaluation of system-level resilience – the system’s capability to maintain service availability despite failures in individual services. Chaos engineering evaluates this resilience by injecting controlled failures into target services during execution, such as service crashes, network disruptions, and timeouts.

To improve resilience under such failures, a *service mesh* provides a cloud-native infrastructure

Type	Benchmark	Task			Test Case	
		Amount	Length	Functional Units	Basic	Advanced
Web	rSDE-Bench	45	4,119	5.50	6.49	5.49
	Ours	20	8,793	8.75	–	9.00
GUI	FSD-Bench	41	927	5.35	–	–
	Ours	21	15,880	6.67	–	11.04

Table 10: Comparison between existing benchmarks (rSDE-Bench and FSD-Bench) and our proposed benchmark

layer for managing service-to-service communication and resilience policies. As shown in Figure 4, the control plane manages these policies, while sidecar proxies in the data plane mediate service interactions and detect partial failures through mechanisms such as retries, timeouts, circuit breaking, failover, and health checking. This separation decouples business logic from resilience logic, enabling runtime failure mitigation without modifying application code.

MA-Mesh adopts the control–data plane separation and the sidecar-proxy pattern from service meshes, attaching a resilient proxy to each task-executing agent to mediate inter-agent communication. To adapt this pattern to LLM-based collaboration, cross-agent reflection replaces predefined local health checks with distributed LLM-based evaluation under role-specific criteria. Role-based failover likewise refines the role specification of the replacement agent rather than reinstantiating an identical replica.

C Benchmark Details

In this section, we present the benchmarks used to evaluate our method, including software development (GUI Dev. and Web Dev.), tool use (GAIA), and scientific reasoning (GPQA-Diamond).

C.1 Software Development

To evaluate the coding capabilities of LLMs, prior studies have traditionally relied on benchmarks such as MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021), which target relatively simple programming tasks. However, these benchmarks have become trivial for modern LLMs; for instance, Zhang et al. (2025a) report that a CoT with gpt-4o-mini already attains success rates of 87.08% on HumanEval and 71.88% on MBPP.

In response, more challenging benchmarks such as FSD-Bench (Liu et al., 2025) and rSDE-Bench (Hu et al., 2025b) have recently been introduced to better assess multi-agent systems on

software development tasks. Given a requirement specification, models must produce complete implementations that pass a suite of automated tests, closely mirroring real-world development workflows. Yet even these benchmarks are beginning to saturate. As shown in Table 11, a single-agent setup with gpt-4o-mini, the prevailing model in 2024, achieved only 53.65% accuracy on rSDE-Bench, whereas gpt-4.1-mini reached 79.55% under the same setting within a year.

Model	Release Date	Accuracy (%)
gpt-4o-mini	2024.07	53.65
gpt-4.1-mini	2025.04	79.55

Table 11: Performance saturation on rSDE-Bench

Motivated by these observations, we extend FSD-Bench and rSDE-Bench into a new software development benchmark, aiming to better probe state-of-the-art LLMs. The benchmark consists of two task suites – **Web Development** and **GUI Development** – derived from rSDE-Bench and FSD-Bench, respectively. Each task is built in two steps. We first expand an original specification into a more demanding one by lengthening the description and increasing the number of functional units to implement. We then use Claude Code (Anthropic, 2026) to generate the accompanying Selenium-based automated tests. We validate the resulting benchmark on two fronts: correctness and difficulty. For correctness, two authors independently review every specification and test pair, ensuring that each specification is internally consistent and implementable. For difficulty, we quantitatively evaluate the resulting benchmark. As shown in Table 10, our Web Development tasks are 113% longer with 59% more functional units than rSDE-Bench, while our GUI Development tasks are 1,613% longer with 25% more functional units than FSD-Bench. An example task and its test code are shown in Figure 5 and Figure 6, respectively.

Web Development Task

Requirements Document for 'OnlineLibrary' Web Application

1. Objective

Develop a comprehensive web application named 'OnlineLibrary' using Python, with data managed through local text files. The application enables users to search and browse books, borrow and return books, write reviews, manage reservations, and track their borrowing history. Note that the website should start from the Dashboard page.

2. Language

The required development language for the 'OnlineLibrary' application is Python.

3. Page Design

The 'OnlineLibrary' web application will consist of the following ten pages:

- Borrow Confirmation Page
 - Page Title: Borrow Confirmation
 - Overview: A page to confirm book borrowing details.
 - Elements:
 - ID: borrow-page
Type: Div
Description: Container for the borrow confirmation page.
 - ID: borrow-book-info
Type: Div
Description: Display information about the book being borrowed.
 - ID: due-date-display
Type: Div
Description: Display the due date for return, 14 days from borrow.
 - ID: confirm-borrow-button
Type: Button
Description: Button to confirm borrowing.
- Other 9 pages are omitted:
Dashboard, Book Catalog, Book Details, My Borrowings, My Reservations, My Reviews, Write Review, User Profile, Payment Confirmation.

4. Data Storage

The 'OnlineLibrary' application will store data locally in text files organized in the directory data. The following data formats and examples are defined:

- Borrowings Data
 - File Name: borrowings.txt
 - Data Format:
borrow_id|username|book_id|borrow_date|due_date|return_date|status|fine_amount
 - Example Data:
1|john_reader|2|2024-11-01|2024-11-15||Active|0
3|john_reader|3|2024-10-15|2024-10-29||Overdue|5.00
- 5 other data files are omitted for brevity:
users.txt, books.txt, reservations.txt, reviews.txt, fines.txt.

Figure 5: Example task from the web development benchmark

Test Code for Web Development Benchmark

```
# Example: OnlineLibrary (10 test cases)

class TestCase(unittest.TestCase):
    def setUp(self):
        ... # Initialize

    # ===== Data Persistence Verification =====
    def test_borrow_page_functionality(self):
        ... # Navigate: dashboard -> catalog -> details -> borrow (4 steps)
        with open('data/borrowings.txt', 'r') as f:
            before_count = len(f.readlines())
        self.driver.find_element(By.ID, "confirm-borrow-button").click()
        with open('data/borrowings.txt', 'r') as f:
            after_count = len(f.readlines())
        self.assertGreater(after_count, before_count)

    # ===== Cross-File JOIN Verification =====
    def test_my_borrows_page_functionality(self):
        ... # Navigate to my-borrows-page
        with open('data/borrowings.txt', 'r') as f:
            book_id = f.readline().strip().split('|')[2] # FK: book_id
        with open('data/books.txt', 'r') as f:
            for book in f:
                if book.startswith(f"{book_id}|"):
                    book_title = book.strip().split('|')[1]
                    break
        borrows_table = self.driver.find_element(By.ID, "borrows-table").text
        self.assertIn(book_title, borrows_table)

    # ===== Form Submission with Content Verification =====
    def test_write_review_page_functionality(self):
        ... # Navigate: dashboard -> catalog -> details -> write-review (4 steps)
        with open('data/reviews.txt', 'r') as f:
            before_count = len(f.readlines())
        Select(self.driver.find_element(By.ID, "rating-input")).select_by_value("5")
        self.driver.find_element(By.ID, "review-text").send_keys("Great book!")
        self.driver.find_element(By.ID, "submit-review-button").click()
        with open('data/reviews.txt', 'r') as f:
            after_count = len(f.readlines())
        self.assertGreater(after_count, before_count)
        with open('data/reviews.txt', 'r') as f:
            self.assertIn("Great book!", f.read())

    def test_dashboard_page_functionality(self):
        ... # Verify username from users.txt displayed in welcome-message

    def test_catalog_page_functionality(self):
        ... # Verify book title from books.txt displayed in book-grid

    def test_book_details_page_functionality(self):
        ... # Verify title, author from books.txt match UI elements

    def test_reservations_page_functionality(self):
        ... # Navigate back to dashboard and verify display

    def test_reviews_page_functionality(self):
        ... # Verify reviews-section is displayed on book details

    def test_profile_page_functionality(self):
        ... # Navigate back to dashboard and verify display

    def test_payment_confirmation_page_functionality(self):
        ... # Verify fine amount from fines.txt displayed on payment page
```

Figure 6: Example automated test code from the web development benchmark: The test code verifies data persistence through before-and-after file comparisons, cross-file JOIN operations that resolve foreign keys across relational data files, and form submission with content verification.

C.2 Tool Use

We use the **GAIA** benchmark (Mialon et al., 2024), a suite of real-world question answering tasks that demand multi-step reasoning and the coordinated use of diverse external tools. Solving a GAIA task requires a combination of web browsing, document retrieval, spreadsheet processing, and multimodal understanding over images or videos. Figure 7 shows an example task from the GAIA benchmark.

Example of GAIA

Solve the following question and answer.

Question

The attached file contains a list of vendors in the Liminal Springs mall, along with each vendor's monthly revenue and the rent they pay the mall. I want you to find the vendor that makes the least money, relative to the rent it pays. Then, tell me what is listed in the "type" column for that vendor.

Attached file: {Attached_files}

Finish your answer with the following template:
Final answer: [Your final answer].
Your final answer should be a number OR as few words as possible OR a comma separated list of numbers and or strings.

Figure 7: Example task from the GAIA

C.3 Scientific Reasoning

We use the GPQA benchmark (Rein et al., 2023), a collection of graduate-level multiple-choice questions in biology, chemistry, and physics, designed to assess deep conceptual understanding and multi-step reasoning. Among its difficulty levels, we focus exclusively on **GPQA-Diamond**, the most challenging subset, where even domain experts struggle and precise logical inference is essential. Figure 8 illustrates an example task from the GPQA-Diamond benchmark.

Example of GPQA-Diamond

Solve the following graduate-level science question and select the correct answer from the given options.

Question

How many stereoisomers are there for the compound 6-chloro-9-ethyl-2-methylundeca-3,7-dien-5-ol?

A. 32
B. 16
C. 4
D. 8

Finish your answer with the following template:
FINAL ANSWER: [A/B/C/D]
Your final answer must be exactly one letter:
A, B, C, or D.

Figure 8: Example task from the GPQA-Diamond

D Experimental Details

This section details our setup for evaluating the resilience of multi-agent systems, including system uncertainty scenarios, metrics, and tools.

D.1 System Uncertainty Scenarios

In real-world deployments, multi-agent systems are subject to diverse sources of system-level uncertainty, including hardware failures, network disruptions, and imperfect agent behavior. To systematically assess resilience under such conditions, we design a set of controlled chaos scenarios akin to established chaos engineering practices (Netflix, 2012; Basiri et al., 2016). Chaos engineering deliberately injects controlled failures into systems to assess recovery mechanisms and identify weaknesses before they manifest in production environments. Following this paradigm, we construct three chaos scenarios—**Agent Chaos**, **Prompt Chaos**, **I/O Chaos**—each simulating a distinct failure mode commonly encountered in LLM-based multi-agent systems. In addition, we define **Compound Chaos** as a composite scenario in which all these chaos types are simultaneously present. In our experiments, we enforce that each run includes at least one agent affected by the injected chaos, avoiding trivial executions with no failures. Examples of each chaos scenario are illustrated in Figure 9.

Agent Chaos simulates abrupt agent termination resulting from hardware faults, network failures, or process crashes. Inspired by Chaos Monkey (Netflix, 2012), which randomly terminates service instances to evaluate system resilience, *Agent Chaos* forcefully interrupts selected agents during task execution. When an agent is targeted, its execution is immediately halted, and no response is returned to downstream agents. This allows subsequent agents to continue execution while reflecting the degraded system state. The termination ratio controls the proportion of agents subject to termination. In our experiments, we set the termination probability to 30%, i.e., each agent is independently terminated with a probability of 30% during its execution cycle. We further conduct a sensitivity analysis by varying this ratio from 20% to 60% to quantify performance degradation under increasing failure severity, as shown in Table 2.

Prompt Chaos simulates degraded prompt quality caused by orchestration errors, encoding faults, or transmission noise. Before invoking the LLM, the prompt is corrupted using one of three perturbation

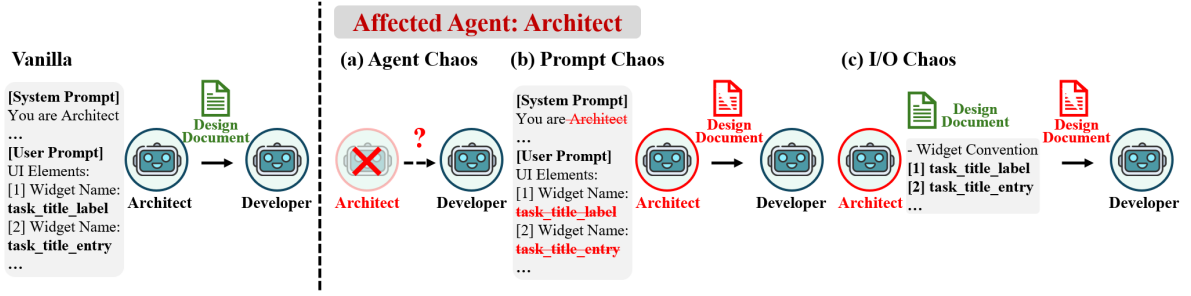


Figure 9: Illustration of system uncertainty scenarios

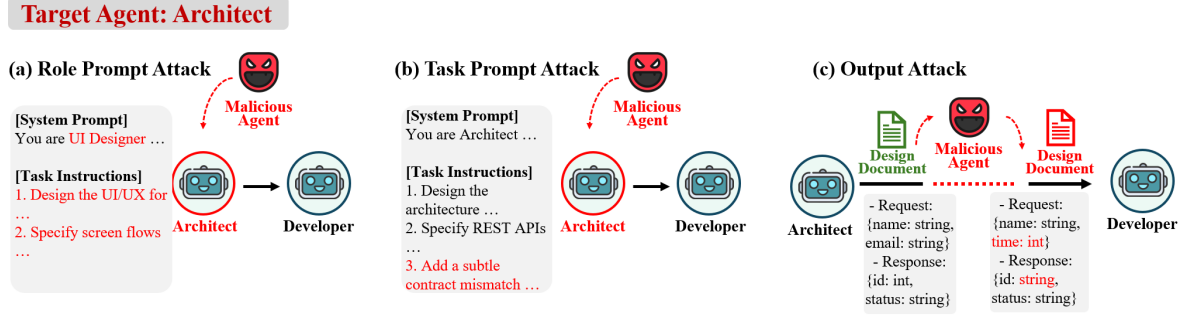


Figure 10: Illustration of adversarial attack scenarios

types: (1) *Shuffle*, which permutes words within the prompt while preserving the word count per line; (2) *Deletion*, which randomly removes 60% of the words; and (3) *Replacement*, which substitutes 60% of the words with random alphanumeric strings of identical length. When an agent is selected to be affected by prompt chaos, a perturbation type is sampled uniformly at random and applied consistently to all LLM calls issued by that agent. The corruption ratio specifies the proportion of agents subjected to prompt corruption. By default, we set the corruption ratio to 20%.

I/O Chaos simulates corruption during inter-agent communication, capturing failure modes such as hallucinated artifacts produced by LLM agents or message transmission errors. After an agent produces its artifacts, I/O Chaos corrupts the content before it is consumed by subsequent agents, using the same three perturbation types as Prompt Chaos. By default, we set the corruption ratio to 20%.

Compound Chaos combines *Agent Chaos*, *Prompt Chaos*, and *I/O Chaos* to simulate settings in which multiple failure modes occur concurrently. For each agent, the three chaos types are sampled independently. This scenario induces cascading effects across the workflow. For instance, an agent may first operate under corrupted instructions due to Prompt Chaos, and its outputs may be further cor-

rupted by I/O Chaos before being consumed by subsequent agents. By default, we set the probability of each chaos type to 20%.

D.2 Adversarial Attack Scenario

We construct adversarial attack scenarios (He et al., 2025; Xie et al., 2025) to evaluate the resilience of multi-agent systems against external attacks from malicious agents. Specifically, we consider three classes of adversarial attacks: role prompt attack, task prompt attack, and output attack.

Role Prompt Attack occurs when a malicious agent tampers with an agent’s role definition, inducing role drift and disrupting coordination.

Task Prompt Attack occurs when a malicious agent modifies task instructions so that the agent produces plausible-looking outputs containing targeted errors.

Output Attack occurs when a malicious agent intercepts and alters generated artifacts of an agent before they are consumed by subsequent agents.

Each agent is independently targeted with a probability of 20%, consistent with the prompt chaos scenario setting. Examples of each adversarial attack scenario are illustrated in Figure 10.

D.3 Noisy Tool Scenario

We further consider tool-side uncertainty (Wang et al., 2026), where tool outputs become unreli-

able after an agent invokes a tool. We define two scenarios: execution failure and noisy response.

Execution Failure simulates failed tool execution by replacing the original response with `This Service is not available.`, indicating that the service returned no valid output. This reflects cases where the tool server is unavailable, the endpoint is unreachable, or the backend raises an exception. **Noisy Response** simulates corrupted tool observations by modifying the original tool response before the agent observes it, using the same operators as I/O Chaos: *Shuffle*, *Deletion*, and *Replacement*.

Each agent is independently designated as a noisy-tool agent with a probability of 20%. For such agents, all tool outputs are modified before they are exposed to the agent, consistent with the I/O Chaos scenario setting.

D.4 Cloud-Native Testbed

To verify MA-Mesh in real-world cloud-native settings, we extend our chaos evaluation to a Kubernetes-based deployment environment. Specifically, we deploy each task-executing agent as an isolated Kubernetes Pod in AIOpsLab (Chen et al., 2025) and inject faults using the Chaos Mesh platform (Chaos Mesh, 2021). Following chaos engineering practice, we construct three infrastructure-level chaos scenarios—**Stress Chaos**, **Disk I/O Chaos**, and **Network Chaos**. **Stress Chaos** emulates resource exhaustion in Pods. When a Pod is selected, CPU and memory pressure is injected during task execution, which delays agent responses, triggers execution timeouts, or causes failures. This scenario evaluates whether the multi-agent system can maintain task continuity when part of its agent population experiences compute-resource degradation.

Disk I/O Chaos emulates storage-level failures in the agent workspace. For selected Pods, we inject file-system faults that disrupt read and write operations over intermediate artifacts. This scenario captures failures in which an agent cannot reliably access, persist, or transfer generated artifacts required by downstream agents.

Network Chaos emulates unreliable pod-level network communication. For selected Pods, we introduce packet loss on pod-to-pod communication and outbound LLM API calls, degrading both message and artifact transfer across agents and the reliability of external LLM responses. This scenario evaluates whether the system can tolerate communication failures that prevent agents from reliably

exchanging intermediate results.

Each Pod is independently selected with a probability of 20%, and we enforce that at least one Pod is affected in every run.

D.5 Evaluation Metrics

Accuracy (Acc). To evaluate task completion performance, we measure accuracy for each benchmark as follows:

- **GUI Development & Web Development.** We define accuracy as the average fraction of passed test cases per task:

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^N \frac{P_i}{T_i}, \quad (5)$$

where N is the total number of tasks, T_i is the number of test cases for task i , and P_i is the number of test cases passed for task i .

- **GAIA & GPQA-Diamond.** We define accuracy as the fraction of correctly answered questions:

$$\text{Accuracy} = \frac{C}{N}, \quad (6)$$

where N is the total number of questions and C is the number of correct predictions.

Detection Recall (DR) and Detection F1 Score (DF1). To evaluate resilience under chaos scenarios, we evaluate how effectively a framework identifies affected agents (Xie et al., 2025). We formulate fault detection as a binary classification problem over task-executing agents, where an agent is considered *positive* if it is affected by chaos and *negative* otherwise. Table 12 summarizes the confusion matrix used to compute the detection metrics.

From this confusion matrix, we define detection recall (DR), detection precision (DP), and detection F1 score (DF1),

$$\begin{aligned} \text{DR} &= \text{TP} / (\text{TP} + \text{FN}), \\ \text{DP} &= \text{TP} / (\text{TP} + \text{FP}), \\ \text{DF1} &= 2 \cdot \text{DP} \cdot \text{DR} / (\text{DP} + \text{DR}). \end{aligned} \quad (7)$$

Actual	Detected	
	Affected	Unaffected
Affected	TP	FN
Unaffected	FP	TN

Table 12: Confusion matrix for detection metrics

DR measures the fraction of affected agents correctly identified, while DF1 balances missed detections and false alarms via the harmonic mean of precision and recall. In the main experiments reported in Section 5.2, we present only DR due to space limitations, and provide the full set of detection metrics including both DR and DF1, in Appendix F.7. For methods without explicit resilience mechanisms (e.g., CoT, MetaGPT, LLM-Debate, AgentVerse, and EvoMAC in Table 1), we report $DR = DP = DF1 = 0.0$.

D.6 Baseline Methods

The baselines in our experiments fall into four categories: **[a]** single-agent method (e.g., CoT), **[b]** static multi-agent system (e.g., MetaGPT, RTADev, LLM-Debate), **[c]** autonomous multi-agent system (e.g., AgentVerse, EvoMAC), **[d]** resilient multi-agent system (e.g., C+I). Below, we detail the implementation of each baseline.

[a] CoT (Wei et al., 2022) is a single-agent baseline without an explicit coordinator or inter-agent communication, where the agent receives the task prompt with an instruction to solve the task in a structured step-by-step sequence.

[b] MetaGPT (Hong et al., 2024) is a static multi-agent baseline that combines sequential and reflection coordination structures, following a software-company SOP. Specifically, five agents with distinct roles participate in the collaboration—Product Manager, Architect, Project Manager, Engineer, and QA Engineer. We set the maximum total number of agent calls per task to 15 and use the prompt templates from the original implementation.

[b] RTADev (Liu et al., 2025) combines sequential, refinement, and debate coordination structures. Fixed agents—Product Manager, Architect, Project Manager, Programmer, and Code Tester—progress five key stages of software development: requirement analysis, architecture design, planning, programming, and testing. To keep downstream artifacts aligned with upstream intent, the Reviewer agent performs an alignment check on the artifacts produced at each key stage. When a misalignment is suspected, it triggers a debate among the previously active agents to prompt refinement. We cap the alignment-check iterations per key stage to 3.

[b] LLM-Debate (Du et al., 2024) employs a debate coordination structure, where two Solver agents and one Judge agent participate. The Solver agents first solve the task in parallel, then enter debate rounds in which each receives the other’s

response, critiques it, and refines its own answer. After the debate, the Judge agent reviews the Solver agents’ responses and returns the final answer. We cap the number of debate rounds at 2.

[c] AgentVerse (Chen et al., 2024b) is an autonomous multi-agent baseline with a dynamic expert-recruitment mechanism. It organizes problem solving into 4 stages. In the Expert Recruitment stage, a recruiter agent selects task-specific expert roles based on the current task. In the Collaborative Decision-Making stage, the recruited experts discuss the task to determine a solution. In the Action Execution stage, the selected agents execute the agreed solution. In the Evaluation stage, an evaluator assesses the outcome and provides feedback on whether to proceed with further recruitment. If the task remains unsolved, the workflow returns to Expert Recruitment with refined agent roles; otherwise, it terminates. Following the original implementation, we set the number of recruited experts per benchmark: 4 for software development and 2 each for tool use and scientific reasoning. We cap the recruiting rounds at 2.

[c] EvoMAC (Hu et al., 2025b) adopts a self-evolving multi-agent collaboration (MAC) network specifically tailored for software development. The network updates its coordination structure via textual backpropagation, converting unit-test execution results into textual gradients to refine agent prompts and workflows. It comprises three teams. The Coding team decomposes the task into sub-tasks and builds a coding MAC network where Programmer agents generate code for each sub-task. The Testing team analogously builds a testing MAC network that generates unit tests. Then, a compiler executes the code and tests to produce execution logs. In the Updating team, a Gradient agent inspects these logs to identify completed, erroneous, and missing sub-tasks, and an Updating agent revises the network accordingly. Following the original implementation, we cap the coding MAC network iteration at 2.

[d] C+I (Huang et al., 2025) is a resilient multi-agent baseline that augments an existing multi-agent system with correction mechanisms, introducing a Challenger and an Inspector to improve robustness against faulty agents. The Challenger is a mechanism attached to each agent: before performing its own task, the agent challenges other agents’ outputs and requests rewriting or re-execution when they are judged unsafe. The Inspector is a separate agent that intercepts messages

Category	Tools	Implementation Details
Web Search	web_search search_wiki search_wiki_revisions search_archived_webpage	API-based search services
Browser Simulation	browse_url	Playwright-based browsing
Document Processing	extract_document_content extract_excel_content	Document parsing libraries e.g., PyMuPDF, python-docx, openpyxl
Multimodal Analysis	ask_question_about_image ask_question_about_audio ask_question_about_video	GPT-4.1-mini (image/video) GPT-4.1-mini + Whisper-1 (audio)
Code Execution	execute_python_code validate_python_file	Python subprocess executor
File I/O	write_text_file	Standard file system operations

Table 13: Implementation details of tools used for GAIA benchmark

exchanged among agents, checks for errors or unsafe behaviors using the conversation history, and rewrites them when correction is needed. Following the original implementation, we apply both mechanisms on top of MetaGPT for software development and LLM-Debate for the others, and cap the Challenger and Inspector rounds at 2 per agent.

D.7 Tool Design

To evaluate tool use on the GAIA benchmark, we define a set of tools supporting code execution, web search, document parsing, and multimodal analysis, following prior tool-augmented agent frameworks (Hu et al., 2025a; Li et al., 2023). Each agent is assigned an available subset of tools according to its role, as specified in the agent profile. Table 13 summarizes the tool categories and their corresponding implementation details.

E Implementation Details of MA-Mesh

In this section, we present implementation details of the MA-Mesh framework, including the generation of the resilience-aware system specification, the synthesis of executable multi-agent policy code, and the pseudocode for the core resilience mechanisms. For both the baselines and MA-Mesh, we use gpt-4.1-mini-0414 (OpenAI, 2025) via the API, with temperature set to 0.0.

E.1 Resilience-Aware System Specification

To construct the resilience-aware system specification, we employ an LLM orchestrator ϕ_{LLM} that transforms a given task $\mathcal{T}$ into a structured resilience-aware system specification. The detailed

prompt used to generate the resilience-aware system specification is provided in Appendix G.

Task Decomposition. Given a task $\mathcal{T}$, the orchestrator decomposes it into a set of sub-tasks. For each sub-task, it jointly determines the following components: (i) coordination structure, (ii) agent configuration, (iii) cross-agent reflection.

Coordination Structure. Building on prior works in multi-agent systems (Wu et al., 2024; Kim et al., 2025b), we adapt four common collaboration patterns. For each sub-task, the LLM orchestrator selects an appropriate structure.

- **Sequential:** A linear execution pattern in which the output of each agent is passed as input to the subsequent agent. This structure is suitable for tasks that require ordered processing with explicit inter-step dependencies.
- **Parallel & Merge:** Multiple agents execute their assigned roles concurrently, followed by a merger agent that aggregates their outputs. This structure enables efficient parallel execution and the synthesis of complementary results.
- **Refine:** An iterative collaboration pattern where a generator agent produces an initial output and a critic agent provides feedback to iteratively refine it until the quality criteria are satisfied. This structure is effective for tasks requiring iterative improvement and high-quality outputs.
- **Debate:** Multiple agents independently generate candidate solutions in parallel, review one another’s outputs, and iteratively refine their responses. A designated decision-maker then synthesizes the final result. This structure is particu-

larly effective for tasks that benefit from diverse reasoning strategies and cross-validation.

Figure 11 illustrates each coordination structure. While these four coordination structures serve as default templates, the orchestrator prompt is designed to retain flexibility, allowing the orchestrator to compose coordination structures by combining existing templates when required by sub-task-specific characteristics.

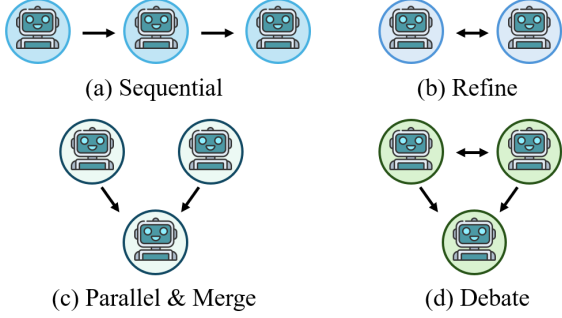


Figure 11: Coordination structures

Agent Profile. Following task decomposition and coordination structure assignment, the LLM orchestrator defines each agent’s role by specifying its responsibilities, permitted tools, input sources and modalities, as well as the expected output artifacts and formats. Moreover, a detailed role-specific prompt is generated for each agent through separate calls to the LLM orchestrator.

Resilient proxy: cross-agent reflection. The orchestrator generates cross-agent reflection policies, each comprising three components: (i) a reflection source, namely a resilient proxy that performs the evaluation on behalf of its associated agent, (ii) a reflection target (agent), whose output is subject to evaluation, and (iii) reflection criteria, which define the evaluation standards from the perspective of the reflecting agent’s role. The cross-agent reflection enables role-aware evaluation, whereby resilient proxies assess peer agents’ outputs according to the functional requirements and expectations of their associated roles. For example, in a web development task, when an *Architect* agent produces a design document, the resilient proxy associated with a *Backend Developer* evaluates the document for backend completeness (e.g., API routes and data schemas), while the resilient proxy associated with a *Frontend Developer* evaluates the same output for frontend completeness (e.g., HTML templates and element identifiers). By aligning reflection

Algorithm 2 Procedure of Cross-Agent Reflection

```

1: Input: Proxy  $P$ , Request  $x$ , Response  $y$ 
2: Output: Feedback  $f$ 
3: /* retrieve reflection sources (peer resilient proxies) for
   assessing the agent associated with  $P$  */
4:  $S \leftarrow \text{get\_reflection\_source\_list}(P)$ 
5:  $f \leftarrow \text{None}$ 
6: if  $S \neq \emptyset$  then
7:   /* cross-agent reflection: aggregate feedback from
      multiple reflection sources */
8:   for each  $P_s \in S$  do
9:      $f' \leftarrow \text{generate\_review}(P_s, x, y)$ 
10:     $f \leftarrow \text{concat}(f, f')$ 
11:   end for
12: else
13:   /* self-reflection: fall back to  $P$  as reviewer */
14:    $f \leftarrow \text{generate\_review}(P, x, y)$ 
15: end if
16: return  $f$ 

```

Algorithm 3 Procedure of Role-based Failover

```

1: Input: Proxy  $P$ 
2: Output: Agent  $A$ 
3: /* Reconfigure agent through meta-agent */
4:  $A \leftarrow \text{reconfigure\_agent}(P)$ 
5: return  $A$ 

```

criteria with actual inter-agent dependencies, this role-aware reflection enables precise detection of errors and inconsistencies before they propagate through the multi-agent workflow.

E.2 Multi-Agent Policy Code Synthesis

To translate the resilience-aware system specification into executable code, we implement an AutoGen-based interface that supports both standard agent actions and the resilient proxy’s mechanisms (i.e., **cross-agent reflection** and **role-based failover**). Specifically, we define a set of API functions that are composed by the synthesized policy code:

- **cross_agent_reflection(proxy, request, response)** – peer agents’ resilient proxies evaluate the agent’s output against predefined criteria and return feedback for potential revision.
- **role_based_failover(proxy)** – the meta-agent replaces the agent associated with the given proxy upon degradation or timeout. Only the original inputs are transferred, while prior state is reset to prevent error propagation.

Method	Prompt Token			Completion Token			API Cost(\$)			Wall-clock Time (min)		
	Agent	Resilient	Total	Agent	Resilient	Total	Agent	Resilient	Total	Agent	Resilient	Total
RTADev	204,897	–	204,897	61,314	–	61,314	0.180	–	0.180	16.26	–	16.26
C+I	47,264	344,844	392,108	18,930	10,715	29,645	0.049	0.155	0.204	5.12	3.14	8.26
MA-Mesh	167,271	125,210	292,481	29,714	3,144	32,858	0.114	0.055	0.170	3.76	0.77	4.53

Table 14: Detailed inference cost analysis

Resilient Proxy LLM	Performance		API Cost (\$)			Wall-clock Time (min)		
	Acc (%)	DR	Agent	Resilient	Total	Agent	Resilient	Total
gpt-4.1-mini	65.30	0.84	0.139	0.047	0.186	9.71	0.75	10.46
o4-mini	71.64	0.91	0.258	0.360	0.618	14.93	4.93	19.87
gemini-3-flash	69.50	0.71	0.273	0.097	0.370	11.18	1.36	12.53
gpt-5.4	75.27	1.00	0.416	1.200	1.616	19.60	2.82	22.42

Table 15: Trade-offs across resilient proxy LLMs

- `build_agent_with_resilient_proxy(agent_config, proxy_spec)` – this constructs a runnable agent instance augmented with its associated resilient proxy.
- `execute(instance, messages)` – this executes the instantiated multi-agent system on the input messages and returns the final outputs.

Given the resilience-aware system specification, the LLM orchestrator configures each agent with the appropriate parameters and generates executable Python code that instantiates agents along with their corresponding resilient proxies. The synthesized code composes the predefined APIs to realize the assigned coordination structure while enforcing resilience mechanisms at runtime without loss of task execution continuity. The procedures for cross-agent reflection and role-based failover are listed in Algorithm 2 and Algorithm 3, respectively. Moreover, the detailed prompts used for code synthesis are provided in Appendix G.

F Additional Experiments

F.1 Cost Analysis

Detailed Inference Cost Analysis. Table 14 complements Table 6 in the main manuscript by decomposing inference cost into prompt and completion token usage. As shown, for the resilient part, MA-Mesh uses fewer prompt and completion tokens than C+I, which employs a correction mechanism. Moreover, the total number of completion tokens in MA-Mesh remains lower than that of RTADev. Consistent with these trends, the API cost of MA-Mesh does not increase substantially when resilience mechanisms are enabled; instead, it remains lower than C+I and comparable to RTADev.

Trade-offs across LLMs for Resilient Proxy. Table 15 compares four resilient proxy LLMs on GUI Dev. under Prompt Chaos in terms of resilience, cost, and latency, with task agents fixed to gpt-4.1-mini. gpt-5.4 improves accuracy and detection recall at the highest cost and latency, while o4-mini and gemini-3-flash provide intermediate options. These results show that the proxy LLM can be selected according to the desired balance between resilience, cost, and latency.

F.2 Coordination Structure Analysis

Distribution of Coordination Structures. As described in Section 4.2, the LLM orchestrator dynamically selects among four coordination structures based on the goal of the sub-task. Figure 12 illustrates the distribution of coordination structures selected across four benchmarks. For GUI and web development benchmarks, MA-Mesh flexibly combines *Sequential*, *Refine*, and *Parallel & Merge* patterns. This reflects the nature of these tasks, where multiple UI components or functionalities can be implemented in parallel, while their dependencies, integration, and correctness are validated through sequential execution and iterative refinement. For GAIA benchmark, MA-Mesh exhibits a higher reliance on *Parallel & Merge* pattern, because its tasks are inherently amenable to parallel execution using multiple tools and independently solvable sub-tasks. For GPQA-Diamond benchmark, MA-Mesh mainly leverages *Debate* pattern, as scientific reasoning often requires analysis and verification from multiple perspectives through structured discussion. MA-Mesh also complements Debate with *Parallel & Merge*, as multi-domain questions can be solved independently.

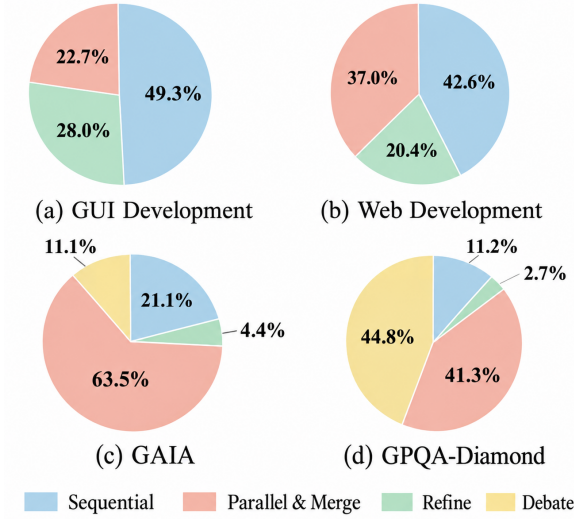


Figure 12: Distribution of coordination structures

Cross-task Differences. For analyzing the relationship among tasks, coordination structures, and performance, we measure MA-Mesh’s relative accuracy decrease from Vanilla across the four chaos scenarios. As shown in Table 16, Parallel & Merge-dominant GAIA exhibits the largest average decrease, since a single affected agent can directly corrupt the merged output. Debate-dominant GPQA-Diamond exhibits the smallest average decrease, since independently reasoning agents can compensate for one another’s failure.

Benchmark	Dominant Structure	Relative Acc. Drop (%)
GUI Dev.	Sequential (49.3%)	10.39
Web Dev.	Sequential (42.6%)	11.62
GAIA	Parallel & Merge (63.5%)	16.06
GPQA-Diamond	Debate (44.8%)	8.80

Table 16: Cross-task resilience by coordination structure

Controlled Structure Comparison. To isolate the structure’s effect from task-specific factors, we force Debate on Web Dev., which is normally dominated by Sequential, and force Parallel & Merge on GPQA-Diamond, which is normally dominated by Debate. As shown in Table 17, both forced non-dominant structures underperform in both Vanilla and Compound Chaos, indicating the importance of task-appropriate structure selection. Nevertheless, Debate exhibits a smaller accuracy decrease under Compound Chaos on both tasks. These results demonstrate that performance under system uncertainty reflects both task fit and the intrinsic resilience of the selected coordination structure.

Benchmark	Coordination Setting	Vanilla	Compound Chaos	
		Acc (%)	Acc (%)	DR
Web Dev.	Debate + MA-Mesh	49.68	44.45	0.97
	MA-Mesh	59.68	49.76	0.91
GPQA Diamond	Parallel & Merge + MA-Mesh	66.67	57.58	0.95
	MA-Mesh	67.68	61.45	0.96

Table 17: Controlled comparison across structures

F.3 Comparison with Retry-upon-failure

To compare with a naive baseline under Agent Chaos, we evaluate retry-upon-failure, which re-executes the terminated agent. Since its original role remains valid, retry recovers substantial performance. To isolate the benefit of role diagnosis and refinement, we additionally evaluate DoS prompt injection, which induces repetitive generation and eventually triggers a timeout that persists under simple re-execution. Table 18 shows that MA-Mesh substantially outperforms retry, demonstrating its benefit when failures originate from the role or prompt rather than agent availability.

	Recovery Method	Agent Chaos	DoS Prompt Injection
Web Dev.	Retry-upon-failure	46.00	13.36
	MA-Mesh	53.98	57.51
GUI Dev.	Retry-upon-failure	67.95	43.88
	MA-Mesh	71.05	66.65

Table 18: Accuracy comparison with retry-upon-failure

F.4 Extended Ablation Study

To examine whether each component remains effective across uncertainty types and tasks, Table 19 reports additional ablation results across all four chaos scenarios on Web Dev. and GPQA-Diamond that complement Table 7 in the main manuscript. As shown in Table 19, all ablated variants exhibit degraded performance across both benchmarks. This demonstrates that each resilience component consistently contributes across uncertainty types and tasks. The degradation is larger on Web Dev. than on GPQA-Diamond because Web Dev. involves a multi-step pipeline of interdependent artifacts, whereas the Debate structure predominantly selected for GPQA-Diamond provides comparatively independent reasoning paths that are less prone to cascading failure.

F.5 Sensitivity Analysis

Tables 20, 21, and 22 report the sensitivity analysis of MA-Mesh on the Web Dev. benchmark under the Compound Chaos setting. We vary three threshold parameters used in resilient proxies:

Benchmark	Method	Vanilla	Agent Chaos		Prompt Chaos		I/O Chaos		Compound Chaos	
		Acc (%)	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR	Acc (%)	DR
Web Dev.	MA-Mesh	59.68	53.98	1.00	53.11	0.94	54.12	0.75	49.76	0.91
	– Cross-agent Reflection	49.89	52.27	1.00	3.13	0.00	9.75	0.00	17.57	0.57
	– Role-based Failover	52.50	3.50	1.00	14.79	0.93	7.00	0.95	4.13	0.35
	– Resilient Proxy	46.06	0.00	0.00	1.61	0.00	8.89	0.00	0.00	0.00
GPQA-Diamond	MA-Mesh	67.68	61.87	1.00	62.46	0.81	61.11	0.98	61.45	0.96
	– Cross-agent Reflection	60.61	59.09	1.00	47.98	0.00	45.45	0.00	52.02	0.66
	– Role-based Failover	61.62	53.03	1.00	53.03	0.77	44.44	0.88	46.46	0.34
	– Resilient Proxy	58.59	50.00	0.00	49.49	0.00	42.93	0.00	43.94	0.00

Table 19: Extended ablation on resilient proxy in MA-Mesh

- **Timeout threshold:** The maximum waiting time before a resilient proxy triggers role-based failover when an agent does not return a response.
- **Max reflection attempts:** The maximum number of cross-agent reflection rounds before triggering role-based failover.
- **Max retry:** The maximum number of role-based failover attempts for replacing or reconfiguring an affected agent.

For the timeout threshold, accuracy increases substantially from 43.66% to 52.72% when the threshold grows from 100 to 200 seconds, while further increasing it to 300 seconds does not provide additional improvement. This suggests that 200 seconds is sufficient for most agent executions in the Web Dev. benchmark.

Timeout threshold (s)	Acc (%)	DR
100	43.66	0.97
200	52.72	0.97
300	50.38	0.99

Table 20: Sensitivity analysis of timeout threshold

For max reflection attempts, MA-Mesh achieves the best accuracy with 2 reflection rounds. Increasing the number of rounds to 3 slightly reduces accuracy and increases cost, suggesting that excessive reflection may introduce redundant feedback into the context window.

Max reflection attempts	Acc (%)	Cost (\$)
1	43.50	0.103
2	47.92	0.142
3	46.52	0.177

Table 21: Sensitivity analysis of max reflection attempts

For max retry, increasing the threshold from 1 to 2 yields only a marginal accuracy gain while

substantially increasing cost. This indicates that most failures are already handled by cross-agent reflection or recovered within the first role-based failover attempt, consistent with the nested procedure in Algorithm 1.

Max retry	Acc (%)	Cost (\$)
1	49.76	0.105
2	50.65	0.176

Table 22: Sensitivity analysis of max retry

F.6 Case Study

Figure 13 qualitatively illustrates the behavior of the resilient proxy in MA-Mesh. The upper part of the figure shows a case of cross-agent reflection, where the architect agent’s proxy evaluates the developer agent’s implementation against the architectural perspective. The proxy detects an incorrect implementation of the *clear_inputs* function that inadvertently removes the listbox and provides feedback, enabling the developer agent to correct the error. The lower part of the figure shows role-based failover. Here, partial corruption of the architect’s role prompt leads to an incorrect system design that overlooks the input format. Upon triggering failover, the role prompt is reconstructed with explicit constraints on data formats and required fields, allowing the architect agent to resume correct behavior.

F.7 Detailed Main Experiment Results

Tables 23, 24, 25, and 26 report additional experiment results that could not be included in Table 1 of the main manuscript due to space constraints. In particular, we present the full set of detection performance metrics, including detection recall (DR) and detection F1 score (DF1), across all benchmarks. As shown, MA-Mesh consistently achieves the highest DF1 for all cases, indicating its superi-

(a) Example of cross-agent reflection



(b) Example of role-based failover

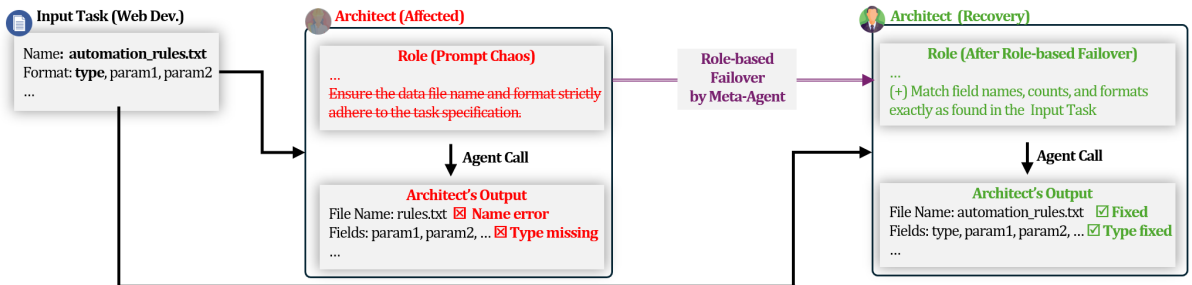


Figure 13: Examples of resilient proxy behavior in MA-Mesh

ority in accurately identifying affected agents under diverse system uncertainty scenarios.

G Prompts used in MA-Mesh

In this section, we provide the prompts used by the LLM orchestrator in MA-Mesh for constructing resilient multi-agent systems. Specifically, Figure 14 shows the prompt for generating the resilience-aware system specification, Figure 15 shows the prompt for generating role-specific agent prompts, and Figure 16 shows the prompt for synthesizing executable multi-agent policy code.

H Scientific Artifacts

The License for artifacts. We use all datasets, frameworks, and tools in accordance with their respective licenses, including AutoGen (Wu et al., 2024), the GAIA (Mialon et al., 2024) and GPQA-Diamond (Rein et al., 2023) benchmarks, AIOp-sLab (Chen et al., 2025), and Chaos Mesh (Chaos Mesh, 2021). We do not redistribute any third-party artifacts, and our source code and the benchmark we construct will be released under an open-source license for reproducibility.

Artifact use consistency with intended use. We use all artifacts in line with their intended purpose, employing the benchmarks for evaluating multi-agent task performance and resilience, and the chaos-engineering tools for injecting controlled system uncertainty. Our newly constructed software development benchmark, extended from FSD-Bench (Liu et al., 2025) and rSDE-Bench (Hu et al., 2025b), is intended solely for research.

GUI Dev.	Vanilla	Agent Chaos			Prompt Chaos			I/O Chaos			Compound Chaos		
Method	Acc (%)	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1
CoT	66.02 $\pm$ 2.53	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	25.03 $\pm$ 5.87	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	3.96 $\pm$ 4.40	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
RTADev	66.30 $\pm$ 1.60	21.87 $\pm$ 8.35	0.61 $\pm$ 0.15	0.51 $\pm$ 0.09	39.64 $\pm$ 3.27	0.42 $\pm$ 0.14	0.42 $\pm$ 0.12	33.90 $\pm$ 13.38	0.19 $\pm$ 0.17	0.28 $\pm$ 0.06	14.99 $\pm$ 8.29	0.43 $\pm$ 0.09	0.47 $\pm$ 0.07
MetaGPT	70.51 $\pm$ 4.22	7.39 $\pm$ 6.10	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	21.62 $\pm$ 5.12	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	22.63 $\pm$ 11.46	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
C+I	68.05 $\pm$ 2.46	4.46 $\pm$ 4.47	0.24 $\pm$ 0.03	0.28 $\pm$ 0.08	19.30 $\pm$ 6.23	0.29 $\pm$ 0.04	0.22 $\pm$ 0.04	47.37 $\pm$ 4.24	0.86$\pm$0.03	0.60 $\pm$ 0.04	1.59 $\pm$ 2.75	0.30 $\pm$ 0.03	0.39 $\pm$ 0.06
AgentVerse	66.53 $\pm$ 6.59	32.87 $\pm$ 6.03	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	44.61 $\pm$ 8.08	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	37.89 $\pm$ 12.94	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	26.36 $\pm$ 6.13	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
EvoMAC	72.79 $\pm$ 1.96	48.01 $\pm$ 11.28	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	52.15 $\pm$ 8.34	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	63.65 $\pm$ 7.45	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	24.38 $\pm$ 11.92	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
MA-Mesh	75.29$\pm$4.34	71.05$\pm$1.64	1.00$\pm$0.00	0.94$\pm$0.04	65.30$\pm$5.48	0.84$\pm$0.08	0.75$\pm$0.06	67.87$\pm$5.07	0.82$\pm$0.07	0.73$\pm$0.09	65.66$\pm$5.69	0.90$\pm$0.05	0.92$\pm$0.03

Table 23: Detailed performance on GUI development benchmark

Web Dev.	Vanilla	Agent Chaos			Prompt Chaos			I/O Chaos			Compound Chaos		
Method	Acc (%)	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1
CoT	45.73 $\pm$ 3.02	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.60 $\pm$ 1.04	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.19 $\pm$ 0.32	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
RTADev	40.17 $\pm$ 6.78	12.68 $\pm$ 4.60	0.44 $\pm$ 0.17	0.42 $\pm$ 0.16	8.85 $\pm$ 0.53	0.46 $\pm$ 0.02	0.41 $\pm$ 0.04	16.74 $\pm$ 4.50	0.20 $\pm$ 0.11	0.18 $\pm$ 0.11	2.68 $\pm$ 4.12	0.36 $\pm$ 0.06	0.41 $\pm$ 0.03
MetaGPT	42.09 $\pm$ 7.03	4.81 $\pm$ 2.77	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	6.30 $\pm$ 5.03	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	11.91 $\pm$ 3.30	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.41 $\pm$ 0.35	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
C+I	45.25 $\pm$ 3.67	5.72 $\pm$ 2.81	0.10 $\pm$ 0.14	0.12 $\pm$ 0.18	12.47 $\pm$ 9.17	0.31 $\pm$ 0.07	0.21 $\pm$ 0.04	14.03 $\pm$ 0.39	0.75$\pm$0.10	0.57 $\pm$ 0.10	0.00 $\pm$ 0.00	0.25 $\pm$ 0.04	0.35 $\pm$ 0.06
AgentVerse	52.01 $\pm$ 2.06	11.95 $\pm$ 6.38	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	21.55 $\pm$ 8.06	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	12.50 $\pm$ 1.38	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	18.38 $\pm$ 3.77	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
EvoMAC	41.33 $\pm$ 4.47	20.45 $\pm$ 4.72	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	24.29 $\pm$ 2.56	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	14.72 $\pm$ 7.17	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	5.80 $\pm$ 5.10	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
MA-Mesh	59.68$\pm$1.84	53.98$\pm$1.55	1.00$\pm$0.00	0.93$\pm$0.03	53.11$\pm$1.82	0.94$\pm$0.04	0.90$\pm$0.09	54.12$\pm$1.58	0.75$\pm$0.13	0.68$\pm$0.15	49.76$\pm$0.32	0.91$\pm$0.02	0.90$\pm$0.03

Table 24: Detailed performance on web development benchmark

GAIA	Vanilla	Agent Chaos			Prompt Chaos			I/O Chaos			Compound Chaos		
Method	Acc (%)	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1
CoT	28.61 $\pm$ 3.57	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	1.01 $\pm$ 0.35	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	4.85 $\pm$ 1.60	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.61 $\pm$ 0.61	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
LLM-Debate	32.32 $\pm$ 1.85	13.13 $\pm$ 2.45	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	19.39 $\pm$ 2.10	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	15.96 $\pm$ 3.45	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	5.25 $\pm$ 0.93	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
C+I	30.10 $\pm$ 0.92	14.75 $\pm$ 3.11	0.53 $\pm$ 0.06	0.39 $\pm$ 0.05	20.81 $\pm$ 1.95	0.62 $\pm$ 0.04	0.45 $\pm$ 0.06	17.58 $\pm$ 5.97	0.99$\pm$0.00	0.51 $\pm$ 0.07	3.70 $\pm$ 1.91	0.72 $\pm$ 0.09	0.75 $\pm$ 0.09
AgentVerse	24.31 $\pm$ 2.38	13.18 $\pm$ 1.45	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	7.07 $\pm$ 1.75	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	11.31 $\pm$ 5.44	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	7.68 $\pm$ 6.48	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
MA-Mesh	32.73$\pm$1.60	29.90$\pm$4.94	1.00$\pm$0.00	0.74$\pm$0.04	26.46$\pm$3.55	0.83$\pm$0.02	0.59$\pm$0.02	29.09$\pm$0.61	0.92 $\pm$ 0.01	0.63$\pm$0.02	24.44$\pm$4.59	0.94$\pm$0.01	0.86$\pm$0.03

Table 25: Detailed performance on GAIA benchmark

GPQA-Diamond	Vanilla	Agent Chaos			Prompt Chaos			I/O Chaos			Compound Chaos		
Method	Acc (%)	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1	Acc (%)	DR	DF1
CoT	64.14 $\pm$ 1.01	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	27.94 $\pm$ 0.58	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	30.97 $\pm$ 2.10	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	20.04 $\pm$ 1.54	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
LLM-Debate	66.50 $\pm$ 1.27	26.77 $\pm$ 7.02	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	35.52 $\pm$ 7.65	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	41.41 $\pm$ 4.01	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	15.32 $\pm$ 3.44	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
C+I	65.15 $\pm$ 1.33	35.35 $\pm$ 3.81	0.66 $\pm$ 0.09	0.58 $\pm$ 0.07	52.36 $\pm$ 2.04	0.52 $\pm$ 0.02	0.40 $\pm$ 0.07	51.67 $\pm$ 4.22	0.96$\pm$0.03	0.69 $\pm$ 0.03	16.33 $\pm$ 2.49	0.66 $\pm$ 0.22	0.67 $\pm$ 0.25
AgentVerse	62.46 $\pm$ 2.59	28.62 $\pm$ 2.96	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	31.14 $\pm$ 3.04	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	27.95 $\pm$ 11.21	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00	22.90 $\pm$ 6.11	0.00 $\pm$ 0.00	0.00 $\pm$ 0.00
MA-Mesh	67.68$\pm$2.20	61.87$\pm$1.07	1.00$\pm$0.00	0.94$\pm$0.03	62.46$\pm$3.79	0.81$\pm$0.02	0.71$\pm$0.03	61.11$\pm$2.53	0.98$\pm$0.01	0.87$\pm$0.03	61.45$\pm$3.04	0.96$\pm$0.01	0.95$\pm$0.00

Table 26: Detailed performance on GPQA-Diamond benchmark

Prompt for Resilience-Aware System Specification

You are an LLM orchestrator for generating a resilience-aware system specification.
Your task is to derive a structured specification for a resilient multi-agent system from a given task.
Your deliverable is the resilience-aware system specification, consisting of collaboration logic and cross-agent reflection.

You must follow this procedure systematically.

Step 1: Task Understanding & Complexity Assessment

1.1. Goal Analysis

- Extract the core objective
- Identify the primary deliverable (function implementation, analysis report, etc.)

1.2. Domain-Driven SOP Selection

- CodeGeneration Domain: Architecture -> Implementation
- Question Answering Domain: Data Extraction -> Answer Generation
- Custom Domain: Adapt based on domain characteristics

1.3. Complexity Assessment

- Low: Simple function, basic logic, minimal constraints -> SOP-LITE: 1-2 phases
- Medium: Multiple components, moderate constraints, some edge cases -> SOP-STANDARD: 2-4 phases
- High: Complex algorithms, extensive constraints, validation requirements -> SOP-FULL: 2-6 phases

1.4. Workload Assessment & Agent Team Sizing

When to Divide Work:

When a single LLM agent would have difficulty executing their responsibility due to excessive scope, divide the workload among multiple specialized agents using the Parallel & Merge pattern.

Evaluation Considerations:

- Volume: Many distinct outputs or artifacts to produce
- Diversity: Multiple unrelated functional areas or reasoning stages
- Complexity: Multi-tier systems or interdependent workflows

Division Strategy:

- Split by functional domain or problem-solving stage
- Use descriptive agent names reflecting their specialty
- Require a shared specification artifact from a preceding phase to ensure consistency

Step 2: Strategic Sub-task Decomposition

Based on Domain and Complexity, decompose user task into logical phases.

2.1 Phase Identification: Break user_task into 1-6 sub-goals where:

- Each sub-goal directly contributes to the main Goal
- Sub-goals form a logical dependency chain
- Each phase transforms concrete artifacts
- Guarantee: When all sub-goals are completed, the user task is fully achieved (no gaps or redundant steps)

2.2. Phase Consolidation Rules

- Agent Reuse Check: If the same agents appear in consecutive phases doing related work, consolidate them into a single phase with appropriate collaboration pattern
- CRITICAL: Never put the same agent_name in multiple phases' agent_profiles lists
- Avoid Unnecessary Separation: Only create separate phases when different agent teams or fundamentally different objectives are involved

Step 3: Agent Team Design & Coordination Structure Selection

For each sub_task, design the exact workflow function with required parameters: {...}

3.1. sub_task

- A concise string describing the sub-task objective and its expected deliverables.

3.2. coordination_structure (per sub_task)

Choose from available patterns or create new combinations:

Available Coordination Structures: {...}

3.3. agent_profiles

For each agent profile, specify the required fields:

agent_name: Clear, role-revealing name

responsibility: Specific responsibility the agent must complete.

Role Design Principles:

- Define roles in terms of the sub_task-specific objective, not task-specific details
- Focus on the type of work rather than specific application names or features
- Include technical constraints that apply across similar tasks

tools: You must select from the following list of {Available_tools}

input_artifacts: A list of data artifacts this agent requires to start its work.

Each artifact is a dictionary with the following keys:

- type: The type of data: "user", "text_file", "others_file" (binary/non-text files)
- name: The name of the file or a description of the string data.
- source: The name of the agent that produced this artifact.

output_artifacts: Each artifact is a dictionary with the following keys:

- type: The type of data: "text_file", "others_file".
- name: The name of the file to be created or a description of the string data to be returned.

3.4. cross_agent_reflection

Specify the resilient proxy specifications for cross-agent reflection:

- reflection_source: The resilient proxy associated with the reviewing agent.
- reflection_targets: The agent(s) whose output is evaluated.
- reflection_criteria: Role-aware criteria used to evaluate the target agent's output.

Refer to the examples below:

{Examples}

Figure 14: Prompt for generating the resilience-aware system specification

Prompt for Agent Role-Specific Prompt Generation

You are an LLM orchestrator for generating role-specific agent prompts. Your task is to transform a subtask-level system specification into concise, structured role-specific prompts that each agent can execute immediately. Your deliverable is the role-specific agent prompts, consisting of agent responsibilities, goals, task details, tool usage instructions, artifact requirements, and consistency rules.

You must follow this procedure systematically.

Step 1: Ingest the sub_task Specification Block

Receive the overall task as a string and the complete code block defining the sub_task as input.

Step 2: Perform Transformations for Each Agent

For each agent dictionary in the agent_profiles list, perform the following transformations: 'responsibility' to 'prompt' Transformation:

Replace the responsibility field with a prompt field. The value of this new field must be a markdown string following a concise, structured format with essential guidance.

- responsibility: Use domain/technology expertise
- Goal: State generic objectives applicable to the task class with concrete deliverables
- Instructions: Provide concise procedural guidance with 2-4 steps, each containing essential bullets only

Template Structure:

The prompt begins directly with the agent's responsibility description, stating their professional identity and domain/technology expertise.

Following the responsibility, state a clear goal describing what the agent must accomplish with concrete deliverables. The Task Details section provides 3-5 bullet points that specify:

- Which input artifacts to read
- What output artifacts to create and their key components
- Scope boundaries defining what the agent should focus on

The main body consists of 2-4 categorized sections with bold headers.

Each section contains specific requirements, format specifications, or step-by-step implementation guidelines relevant to the agent's task.

If the agent participates in multi-agent collaboration, include consistency rules that ensure data integrity across agents such as naming conventions, field matching requirements, and references to the source of truth.

End with Requirements that explicitly state:

- Tool usage instructions
- Format and naming constraints
- Focus areas and boundaries
- For refine agents: feedback markers ("[APPROVED]" or "NEED_MODIFY") to write

When generating the "Task Details" section of agent prompts, you must reference the specific tools assigned to each agent and provide concrete usage guidance based on their capabilities. {Available_tools}

Refer to the examples below:
{Examples}

Figure 15: Prompt for generating role-specific agent prompts from a subtask level system specification

Prompt for Multi-Agent Policy Code Synthesis

You are an LLM orchestrator for multi-agent policy code synthesis, responsible for transforming a resilience-aware system specification into executable Python code.

Your task is to read the given def block, analyze its structure, and generate runnable Python code by leveraging the predefined APIs listed in **{Available_predefined_APIs}**.

You must follow these rules:

- Always declare all agents specified in the agent_profiles section using build_agent_with_resilient_proxy.
- Each agent must follow the parameters below:
 - agent_name: identical to the name in the def block.
 - proxy_spec: construct the proxy specification from the cross_agent_reflection field associated with the agent in the def block.
 - timeout_threshold: choose a value between 200-500 seconds depending on sub_task complexity.
 - max_reflection_attempts: choose a value between 1 and 3 to specify the maximum number of cross-agent reflection rounds before triggering failover.
 - max_retry: choose a value between 1 and 2 to specify the maximum number of failover attempts for replacing or reconfiguring an affected agent.
- After defining agents, parse the sub_task, coordination_structure, agent_profiles, and cross_agent_reflection fields to determine the execution order and resilience configuration.

You must follow this procedure systematically.

Step 1: Parse the def block.

Step 2: Declare all agents with build_agent_with_resilient_proxy and proper parameters.

Step 3: Read coordination fields (coordination_structure, agent_profiles, cross_agent_reflection).

Step 4: Construct execution flow by combining execute calls (sequential or parallel using asyncio.gather).

Step 5: Return a final runnable Python script based on predefined APIs.

Refer to the examples below:
{Examples}

Figure 16: Prompt for synthesizing executable multi-agent policy code from the resilience-aware specification