AGENTHIVE: A Composable Multi-Agent Framework with First-Class Delegation

Anonymous ACL submission

Abstract

Large-language-model (LLM) agents excel at reasoning and tool use, yet existing multi-agent systems (MAS) rely on static, hand-crafted topologies and struggle with open-ended, evolving tasks. We introduce AGENTHIVE, a delegation-centric MAS framework that treats delegation as a first-class primitive: any agent may spawn and coordinate sub-agents, enabling decentralized control without a central orchestrator. Through recursive delegation, AGENTHIVE dynamically forms task-adaptive structures—trees, forests, and star topologies—without predefined agent graphs. We evaluate AGENTHIVE across four real-world domains, showing that MAS emerge automatically from task demands. The performance gains arise from an expressive, adaptive MAS that allows agents to explore more deeply and cover a wider solution space. Our results demonstrate that first-class delegation is a powerful paradigm for scaling LLM-based autonomous systems. An anonymized version of the source code is available at https://anonymous.4open.science/r/anonymous-B-3F03/.

1 Introduction

Large-language-model (LLM) agents exhibit strong reasoning, tool use, and autonomous task execution. Coordinating multiple agents into a multi-agent system (MAS) further amplifies these capabilities through parallel exploration, hierarchical task decomposition, and cooperative problem solving. Real-world tasks, however, are often complex, open-ended, and unpredictable. The required subtasks and agents cannot be determined in advance, posing challenges for conventional MAS designs.

Most existing MAS frameworks (AutoGen (Wu et al., 2023b), DeepAgents (Inc., 2025a), and CrewAI (Inc., 2024)) employ static, hand-crafted agent topologies with fixed roles and interaction patterns (Figure 1, left). For example, AutoGen provides a predefined orchestration pipeline with fixed agent roles; DeepAgents supports task delegation but relies on a central orchestrator to coordinate agents; and CrewAI enables collaborative reasoning, yet its workflows remain largely tailored to specific task templates. As tasks grow in depth and breadth, these static MAS designs quickly become brittle: they cannot spawn new agents on demand, dynamically build the MAS, or reliably aggregate and verify results across large exploration spaces.

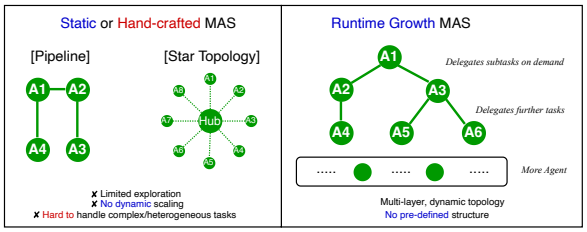

Figure 1: Motivation: Comparison of existing MAS versus dynamic MAS in AGENTHIVE. Left: static, hand-crafted agent topologies with limited flexibility and coverage. Right: dynamic, task-driven topologies with autonomous delegation, parallel exploration, and multi-domain applicability.

Meanwhile, most LLM-as-agent research has focused on prompt engineering, memory mechanism, context engineering, or single-agent planning, leaving system-level MAS design largely unaddressed. Consequently, there remains no general framework that enables LLMs to autonomously generate agents, and adapt multi-agent topologies to the evolving nature of complex, open-ended tasks. To address this gap, we introduce AGENTHIVE, a composable multi-agent framework with first-class delegation (Figure 1, right). AGENTHIVE treats delegation as a first-class primitive: any agent can dynamically generate sub-agents and coordinate their execution, effectively becoming an orchestrator itself. Consequently, the system does not rely on a central controller—every agent has the capability to manage its own sub-agents and form multi-layered topologies. This enables flexi-

ble hierarchical structures such as trees, forests, or star-shaped exploration layers, allowing the MAS to adapt naturally to complex, open-ended tasks. AGENTHIVE decouples framework-level orchestration (agent lifecycle, delegation decision, topology formation) from task-level logic, allowing developers to focus solely on defining constraints rather than manually wiring agent interactions.

We evaluate AGENTHIVE on four real-world tasks: vulnerability discovery, Linux-based filesystem security analysis, academic writing, and commercial-website crawling to extract firmware download URLs and metadata. Through first-class delegation, the system automatically constructs four different MAS structures without any hand-crafted topology: a forest for vulnerability discovery, a tree for filesystem analysis, a star for academic writing, and a two-layer star for firmware crawling. Across all tasks, the improvements stem not from enhancing the underlying LLM, but from providing a more expressive and adaptive MAS substrate that enables deeper exploration, wider coverage, and more efficient coordination.

Our contributions are threefold:

- A delegation-centric paradigm without an orchestrator. We treat delegation as a first-class operation: every agent can autonomously create, and coordinate child agents.
- A framework for dynamic, task-adaptive MAS topologies. Through recursive delegation, the system dynamically constructs multi-layer structures—including trees, forests, and star—without any hard-coded topology or pre-specified agent graph.
- A unified evaluation across four real-world tasks. We show that forest, tree, star, and two-layer star MAS topologies naturally emerge from task demands, demonstrating that performance gains stem from MAS architecture rather than LLM improvements.

2 Related Work

LLM Agents and LLM-assisted Code Analysis. LLMs have demonstrated strong reasoning and code understanding capabilities, and have been applied to code generation, patch generation, binary analysis, and vulnerability discovery (Feng and Chen, 2023; Li et al., 2023; Pearce et al., 2023; Deng et al., 2023; Lemieux et al., 2023; Wang et al., 2024; Liu et al., 2025). Single-agent LLM systems, however, are constrained by context windows, limiting deep exploration of multi-path conditions or hierarchical structures in code or binaries.

LLM-based MAS. MAS extends single-agent paradigms by coordinating multiple LLM agents, often with heterogeneous roles, memory modules, and tool integrations (Zhuge et al., 2024; Tran et al., 2025). The evolution of MAS includes: (i) manually configured systems, such as AutoGen, MetaGPT, and AgentVerse (Wu et al., 2023a; Hong et al., 2023; Chen et al., 2023), which rely on fixed, hand-coded prompts, roles, and topologies; (ii) systems with partial runtime support, like GPTSwarm and G-Designer (Zhuge et al., 2024; Zhang et al., 2024a), which can adjust certain inter-agent communication or routing rules but still execute largely fixed MAS code; and (iii) fully synthesized MAS, such as ADAS, MaAS, and AFlow (Hu et al., 2025b; Zhang et al., 2025a, 2024b), whose configurations are generated end-to-end before execution. Most existing MAS therefore follow a "generate-once-and-deploy" paradigm, meaning their code and structure remain fixed during runtime, limiting adaptability to failures or dynamic tasks.

Orchestrators. Orchestrators coordinate and manage tasks among worker agents in multi-agent systems (Xiong et al., 2025; Hu et al., 2025a; Zhang et al., 2025b; Shi et al., 2025; Fang et al., 2025; Hong et al., 2023; Qian et al., 2023; Hu et al., 2024). In most existing systems, the orchestrator is a single controlling entity, limiting hierarchical delegation and adaptability. Our system generalizes this role into an agent's capability, supporting recursive delegation, dynamic agent creation, and flexible runtime topologies for fully adaptive execution.

Automating MAS Construction. Several works explore automatic MAS construction via external modules, supervised fine-tuning, or RL (Zhuge et al., 2024; Zhang et al., 2024a; Yue et al., 2025; Ye et al., 2025; Wang et al., 2025; Gao et al., 2025; Nie et al., 2025). Our system supports self-adjusting MAS that dynamically adapts to the task and environment.

3 The AGENTHIVE Framework

AGENTHIVE operationalizes first-class delegation through a lightweight, composable tool in which every agent can autonomously generate and coordinate task-specific sub-agents. This section introduces the three core components of the framework: the Base Agent, the Tool interface, and the Delega-

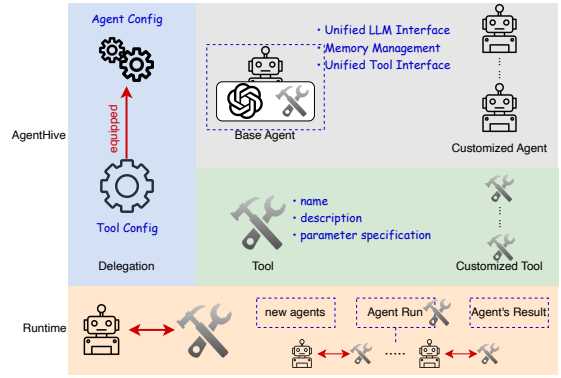

Figure 2: Architecture of AGENTHIVE. Base Agent provides the minimal execution abstraction; tools extend capabilities. Delegation is a first-class primitive that creates and coordinates sub-agents, enabling dynamic MAS structures.

tion primitive (Figure 2).

3.1 Basic Agent

In AGENTHIVE, the *Basic Agent* serves as the fundamental execution unit for MAS, which underlies all agent behaviors. Specifically, the Base Agent integrates three essential capabilities.

1) LLM-backed reasoning. The agent interacts with external LLMs through a robust client that supports synchronous and asynchronous invocation, streaming outputs, and resilience mechanisms such as retry and backoff strategies.

2) Memory and state management. Each agent maintains a short-term conversational history and optional long-term factual memory. All memory entries include origin metadata for verifiability and traceability.

3) Tool execution and background tasks. The agent invokes external tools through a unified interface. Long-running operations are executed as background tasks to avoid blocking the main loop. Returned results are reinjected into the conversation as inputs to subsequent reasoning rounds.

Decision–execution loop. At runtime, the agent repeatedly performs the following steps: (1) construct a prompt from system instructions, memory, and the latest messages; (2) query the LLM; (3) parse the LLM's response; (4) execute the corresponding action: complete the task, invoke a tool, or delegate a task.

Customized Agent. Developers can create customized agents by extending the Base Agent with domain-specific tools, memory policies, reasoning strategies, or other custom logic. Because customized agents share the same messaging interface and decision loop, they can converse and collaborate seamlessly, enabling the construction of multi-layered composable MAS.

3.2 Tools

Tools encapsulate executable capabilities used by agents during task resolution. AGENTHIVE provides a minimal yet expressive tool abstraction.

Tool Abstraction. Each tool is an object that declares: (i) a name, (ii) a natural-language description,(iii) a structured parameter specification, and (iv) a single unified execution entrypoint. This minimal viable contract—returning a single string—is compatible with emerging model–tool cooperation standards such as MCP while maximizing simplicity and ease of use.

Customized Tools. A customized tool extends the base abstraction with task- or domain-specific functionality.

3.3 Delegation: A First-Class Primitive

Delegation is the core innovation of AGENTHIVE. It is implemented as a specialized, customized tool that enables an agent to spawn and coordinate sub-agents through a minimal callable interface. From the caller's perspective, invoking the delegation tool is indistinguishable from invoking any standard tool. Internally, however, delegation orchestrates multi-agent workflows, supporting recursive delegation, isolated reasoning contexts, and hierarchical result aggregation.

Delegation workflow. The delegation process proceeds in four steps: (1) *Delegation Invocation:* The parent agent calls the delegation tool with a sub-task. (2) *Sub-Agent Instantiation:* A new sub-agent is created with its own isolated context and the tools specified in its configuration. (3) *Task Execution:* The sub-agent processes the sub-task. If the sub-agent also has delegation installed, it may recursively create additional sub-agents. (4) *Result Aggregation:* The parent agent collects and folds the sub-agent's results into a compact return value.

Formally, for an agent A with delegation tool λ:

$$
\begin{aligned}
r &= \lambda_{\mathsf{A}}(t) \\
\mathsf{B} &= \text{InstantiateSubAgent}(C_{\mathsf{A}}, \text{config}) \\
r_{\mathsf{B}} &= \text{Execute}(\mathsf{B}, t_{\mathsf{B}}) \\
r &= \text{AggregateResults}(r_{\mathsf{B}})
\end{aligned}
$$

If sub-agent B also has delegation:

$$
r_{\mathsf{B}} = \lambda_{\mathsf{B}}(t_{\mathsf{B}})
$$

Each sub-agent runs in an isolated context $C_B \perp C_A$, ensuring that execution is independent, and only the aggregated result flows back to the parent.

Key design considerations. ① Delegation is a primitive tool: it provides the interface, but policies like recursion depth, sub-task triggers, and termination conditions are left to the application layer. ② Agents without the delegation tool cannot orchestrate; installing the delegation tool equips an agent with orchestrator capability. ③ Delegation enables flexible, problem-dependent multi-agent topologies rather than fixed workflows. ④ From the system perspective, delegation is a tool like any other: its structural position is determined by whether an agent is equipped with it.

Delegation-Induced Design Implications. The delegation primitive not only enables dynamic task decomposition but also brings several intrinsic design advantages that enhance multi-agent workflows:

Context Isolation. Each sub-agent spawned via delegation executes within a strictly isolated context. This ensures that the reasoning of a sub-agent does not interfere with or depend on the parent agent's context, enabling safe parallel execution.

Context Compression. The internal reasoning of sub-agents is folded into compact results returned to the parent agent. This semantic compression allows the parent to incorporate the essential information from sub-agent executions without saturating its own context window, supporting scalable and efficient long-chain and large-scale reasoning.

Evidence Flow and Traceability. Delegation produces a structured chain of results that propagate upward through the hierarchy of agents. These compact results act as traceable evidence, providing observability for debugging, auditing, and verification purposes.

Contrast with existing MAS frameworks. Systems such as DeepAgents (Inc., 2025a), CrewAI (Inc., 2024), and AutoGen (Wu et al., 2023b) rely on bespoke code or external controllers to enable sub-agent workflows. In contrast, AGENTHIVE treats delegation as a composable, installable tool, unifying sub-agent creation with the core tool infrastructure—registration, configuration, observability, and runtime execution.

4 Dynamic Task-Dependent MAS

AGENTHIVE enables the construction of task-dependent MAS dynamically, allowing agents to adjust and reconfigure based on the task. Figure 3 illustrates the flexibility of our framework: through delegation, the system can generate various MAS topologies, such as star, linear, tree, and forest, at runtime. The construction of the MAS involves two main steps:

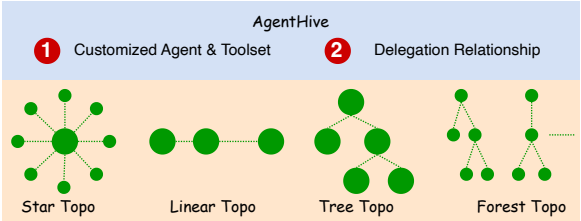

Figure 3: Two-step configuration (agent types and toolsets; declarative delegation relations) and example runtime-generated MAS topologies (star, linear, tree, and forest).

Once configured, the initial agent receives the task and enters its decision-execution loop. When the delegation tool is invoked, the runtime generates a sub-agent according to the delegation relation R, assigns the corresponding toolset and instructions, and executes the subtask in an isolated context. This process continues iteratively until the termination conditions are met. Externally, the system operates as a single agent, while internally, it dynamically orchestrates a multi-agent system.

Single-Agent Mode as a Special Case of MAS. In AGENTHIVE, single agent mode is considered a specific configuration within the broader of MAS application. In this mode, no delegation occurs, and the system behaves as a single agent performing the task.

4.1 Star Topology

In the star topology [1], there is one central coordinator agent that delegates tasks to multiple worker agents. The system dynamically generates this topology using a $1 \rightarrow n$ delegation relation. Agents with the delegation tool act as coordinators, while those without it serve as workers. The exact nature of the tasks and tools can vary depending on the specific application, but the delegation relation remains the same.

4.2 Linear Topology

A linear topology emerges when subtasks must execute sequentially. In our system, a linear topology uses a $1 \rightarrow 1$ delegation relation. Each agent with

[1] Appendix D presents an example of such a star topology for academic writing.

the delegation tool can create its successor, allowing the chain length to vary dynamically: an agent may stop after one step or delegate further based on intermediate results. AGENTHIVE enables linear structures [2] that traditional frameworks cannot express.

4.3 Tree Topology

Tree topologies are relatively uncommon in existing frameworks, where flat or star-based orchestrations are more common. Implementing a true tree structure typically requires manual coding for node expansion, branching logic, dependency management, and result aggregation. Determining when to expand nodes, how to generate child agents, and which results to propagate upward presents a complex and difficult problem in traditional systems.

In AGENTHIVE, a tree topology [3] uses $1 \rightarrow n$ and $1 \rightarrow 1$ delegation relations. Developers only need to define the delegation relations, and the framework automatically handles the creation of child agents and result aggregation.

4.4 Forest Topology

Forest topologies are even less supported in traditional frameworks. In our system, a forest topology [4] emerges when multiple agents independently apply recursive $1 \rightarrow n$ and $1 \rightarrow 1$ delegation. Each root agent generates its own tree, and the trees execute in parallel without shared state. No additional orchestration code is needed.

5 Evaluation

5.1 Experimental Setup

Benchmarks. To provide a comprehensive evaluation of AGENTHIVE, we employ eight benchmarks spanning four domains: **binary vulnerability discovery**, represented by a real-world dataset of binary programs from Mango dataset (Gibbs et al., 2024); **Linux-based filesystem security analysis**, using the Karonte dataset (Redini et al., 2020); **deep research**, represented by benchmark evaluation suite (Du et al., 2025); **web crawling**, using a custom benchmark that simulates multi-domain information gathering tasks.

Baselines. The baselines against which we compare AGENTHIVE can be broadly grouped into two categories: (1) **Single Agent System**. Frameworks that operate a single agent or use tool-augmented prompting without forming MAS structures, including SWE-Agent (Yang et al., 2024). (2) **Star MAS.**, Systems that support multi-agent collaboration but only via fixed coordinator–worker patterns, including DeepAgents (Inc., 2025a) and Open Deep Research (Inc., 2025b). (3) **Pipeline MAS.** Frameworks that construct sequential or workflow-based chains without branching, AutoGen (Wu et al., 2023b), LLM-Debate (Du et al., 2023), and CrewAI (Inc., 2024). (4) **Rule-Driven MAS.** Systems that allow limited routing but still require handcrafted role definitions and branching logic, including DyLAN (Liu et al., 2024). To our knowledge, no existing framework can automatically synthesize tree or forest topologies through declarative delegation. This capability is unique to AGENTHIVE, which supports multi-level branching, recursive expansion, and automatic result aggregation.

Table 1: Expressiveness comparison with existing MAS frameworks.

Topology/Capability	Deep Agents	Auto Gen	Crew AI	Agent Hive
Star (1→n)	Yes	Yes	Yes	Yes
Linear (1→1 chain)	No	No	No	Yes
Tree (recursive 1→n)	No	No	No	Yes
Forest (multi-tree)	No	No	No	Yes
Dynamic depth/width	No	No	No	Yes
Recursive growth	No	No	No	Yes

Parameter & Model Configurations. The backbone LLM is DeepSeek-v3, with temperature fixed at 0 for deterministic reasoning. Each agent is allocated up to 30 iterative reasoning steps. All baselines use the same task interface, analysis tools, and prompts as AGENTHIVE.

5.2 Qualitative Comparison

We qualitatively compare the expressiveness of AGENTHIVE with representative MAS frameworks. Table 1 shows that most existing frameworks support only fixed-pattern star topologies, whereas AGENTHIVE supports runtime-generated star, linear, tree, and forest configurations.

Simple star topologies (1→n) with a central coordinator are common in existing MAS. Linear chains (1→1) and recursive trees (1→n) are rarely supported, as they require explicit node expansion, branching, and result aggregation. Linear chains are a special case of trees, but typically still need

[2] This linear topology is reflected by Forest and Tree MAS

[3] Appendix C is a tree MAS for Linux-based filesystem analysis task.

[4] Appendix B is a forest MAS for binary vulnerability discovery.

Table 2: Vulnerability detection performance across baseline categories and AGENTHIVE.

#	Single Agent		Star MAS		Pipeline MAS		Rule-Driven MAS		AGENTHIVE	
Binary	Vulns	Verified	Vulns	Verified	Vulns	Verified	Vulns	Verified	Vulns	Verified
120	3.43	0.91	11.41	4.52	7.12	2.94	9.30	3.88	43.82	34.14

Table 3: SBOM extraction performance comparison between AGENTHIVE and baselines.

Method	# Components & CVEs	Reasoning Steps	Files Accessed
Single Agent	5.6	49.1	7.3
Star MAS	9.7	173.9	24.9
Pipeline MAS	7.2	102.4	18.1
Rule-Driven MAS	9.3	145.7	21.6
AGENTHIVE	**28.5**	**1718.2**	**96.5**

manual orchestration. Forests, consisting of multiple independent trees, are even less supported due to the complexity of coordinating multiple roots. In contrast, AGENTHIVE leverages first-class delegation to dynamically construct all these topologies, enabling flexible, adaptive multi-agent execution without hard-coded hierarchies.

5.3 Task Performance Overview

Forest Topology for Binary Vulnerability Discovery. This task is very challenging, requiring deep exploration of multiple taint-tracing paths to uncover hidden vulnerabilities. We evaluate AGENTHIVE's ability to automatically instantiate a forest MAS for real-world binary vulnerability discovery. Each source initiates an independent analysis tree, where an agent may delegate multiple sub-agents to explore different taint-tracing paths concurrently.

Results. AGENTHIVE consistently outperforms all baseline categories. As shown in Table 2, single-agent and star-pattern MAS systems discover only a small fraction of vulnerabilities, while pipeline and rule-driven MAS offer moderate but still limited improvements. In contrast, AGENTHIVE achieves a 4–10× increase in vulnerabilities and a 7–30× increase in verified vulnerabilities.

Analysis. Single-agent systems fail at deep binary vulnerability analysis because all reasoning must fit into a single LLM context, leading to context saturation and forgetting. Star-style MAS baselines partially mitigate this via parallel workers, but the coordinator still absorbs all outputs, causing context rot. Pipeline and rule-driven MAS similarly serialize reasoning through a growing context, limiting depth.

AGENTHIVE overcomes these limits by using tree and forest topologies: each sub-agent explores a confined subproblem in isolation, and results are recursively aggregated, enabling deep call-chain exploration, multi-path condition handling, and accumulation of semantic evidence.

Tree Topology for Linux-Based Filesystem Security Analysis. Our second task evaluates how AGENTHIVE builds a tree-structured MAS to analyze Linux-based filesystems extracted from firmware images. The goal is to identify third-party components and map them to known CVEs (SBOM). The filesystem contains many files and nested directories, which require both broad exploration and long reasoning chains. This task is naturally hierarchical: a root agent scans the filesystem layout, spawns sub-agents for directories or packages, and each sub-agent recursively explores deeper layers until all relevant components are recovered.

Results. Table 3 reports the average SBOM extraction performance per filesystem image. Single-agent systems recover only a small fraction of components due to LLM context limits. Star MAS and rule-driven MAS improve coverage via parallel workers or limited delegation, while pipeline MAS suffers from serialized reasoning along the workflow. AGENTHIVE outperforms all baselines by a large margin, recovering 28.5 components and CVEs on average, with far deeper reasoning steps and broader file access.

Analysis. This task is inherently challenging because the components and their names are unknown in advance. Agents must navigate the filesystem step by step to discover relevant files, and for each file, determine the version to link it to known CVEs. This results in very long reasoning chains. Single-agent LLMs often halt after identifying only a few components, and when multiple components are

Table 4: Results on Deep Research Bench using the RACE evaluation protocol. Scores are normalized (0–100).

Model	Overall	Comp.	Ins.	Inst.	Read.	C.Acc.	Eff.C.
Open Deep Research (Inc., 2025b)	50.60	50.06	50.76	51.31	49.72	32.94	21.06
AGENTHIVE (ours)	44.34	44.84	40.56	47.95	44.69	52.86	52.62

analyzed, the LLM tends to forget earlier findings due to context limitations. Moreover, each filesystem has different components and nested directory structures, requiring dynamic adaptation of the MAS topology.

AGENTHIVE tree topology enables adaptive expansion: agents descend into complex directory structures, spawn deeper subtrees when encountering nested archives or packaged libraries, and terminate branches when no further components are found. This flexible recursion leads to significantly richer results and deeper exploration.

Star Topology for Deep Research Generation. Our third task evaluates AGENTHIVE on long-horizon deep research generation. Given a high-level research query, the system must gather domain knowledge, analyze evidence, synthesize arguments, and produce a structured research report. Note that our system employs only a minimal star topology without any task-specific optimizations. We use an LLM (Gemini) judge that compares system-generated reports to expert-written reference reports across six dimensions: *Comprehensiveness (Comp.)*, *Insight Quality (Ins.)*, *Instruction Following (Inst.)*, *Readability (Read.)*, *Citation Accuracy (C.Acc.)*, and *Effective Citations (Eff.C.)*.

Results. Table 4 reports the Deep Research Bench results under the RACE protocol. Overall, our system achieves performance comparable to Open Deep Research. Specifically, AGENTHIVE reaches an overall score of 46.04, only 1.44 points lower than the fully optimized Open Deep Research baseline (47.48). On several dimensions, AGENTHIVE performs competitively or even slightly better. The system also achieves marginally better *Effective Citations* (22.43 vs. 21.62), demonstrating that even without specialized retrieval or citation optimization, the star configuration can coordinate external information use effectively.

Analysis. AGENTHIVE exhibits comparable performance to existing single-coordinator frameworks. This outcome is expected: the system does not perform task-specific optimizations such as prompt engineering, memory management, or context-aware planning, which primarily reside at the application layer. Furthermore, the task only requires a star topology, where a single agent can orchestrate all subtasks. Leveraging first-class delegation in this scenario does not confer additional advantage. Nevertheless, AGENTHIVE can naturally instantiate such star topologies, demonstrating that its flexible delegation mechanism preserves compatibility with traditional MAS configurations while remaining generalizable to more complex topologies.

Table 5: Firmware collection performance across vendors websites.

Vendor	Depth	Navigation	Scraping
D-Link	2	100%	61%
Foscam	2	68%	95%
MikroTik	1	—	93%
OpenWrt	3	100%	95%
QNAP	4	43%	77%
Supermicro	4	22%	97%
TP-Link (en)	2	60%	95%
UI	2	99%	96%
Zyxel	4	78%	89%

Table 6: Firmware extraction rate comparison between AGENTHIVE and baseline categories. Firmware rate indicates the percentage of firmware entries successfully identified.

Method	Firmware Rate
Single Agent	7.5%
Star MAS	58.3%
Pipeline MAS	71.2%
Rule-Driven MAS	70.3%
AGENTHIVE	87.4%

Two-Layer Star Topology for Firmware Web Crawling. Existing web crawling benchmarks typically evaluate tasks with either partially known targets or constrained navigation structures, where the number of relevant items or their identifiers is at least partially known. In contrast, commercial firmware websites present fully open-ended exploration problems: the number and names of firmware entries are unknown, multi-level menus and dynamic JavaScript routing must be handled, and download URLs may be hidden behind redi-

rects or CDNs. We evaluate AGENTHIVE on 9 commercial vendors: D-Link, Foscam, MikroTik, OpenWrt, QNAP, Supermicro, TP-Link (EN), TP-Link (CN), UI, and Zyxel. Each website presents different challenges in terms of page depth, navigation complexity, and download link structures.

Results. Table 5 presents per-vendor firmware collection statistics, including navigation depth, navigation success, and scraping coverage. Table 6 reports the aggregated *Firmware Rate* for each baseline category and AGENTHIVE. AGENTHIVE achieves the highest firmware rate of 87.4%, outperforming single-agent systems (7.5%), star MAS (58.3%), pipeline MAS (71.2%), and rule-driven MAS (70.3%).

Analysis. Extracting firmware URLs and metadata from vendor websites requires navigating multi-level menus, expandable hierarchies, and dynamic scripts from only the homepage. Single-agent systems struggle due to limited context, while star, pipeline, and rule-driven MAS partially parallelize exploration but still bottleneck at a single coordinator. AGENTHIVE employs a two-layer star MAS: sub-agents independently explore site branches, and the top-level agent aggregates results. This aligns naturally with the site's hierarchical structure, avoids context collapse, and achieves higher firmware discovery coverage.

5.4 Additional Analysis

Effect of the Backbone LLM. We evaluate the effect of the backbone model on the overall performance of AGENTHIVE with four widely used candidates: GPT-4o, o1, DeepSeek-v3, and DeepSeek-r1. The results in Table 7 show that the number of validated findings remains within a narrow band (128–162), and precision stays consistently high (70–79%). These results highlight an important property of AGENTHIVE: as long as the backend LLM possesses basic reasoning competence, the multi-agent structure—not the individual model—dominates overall performance. Different LLMs introduce only moderate variation, suggesting that the framework is largely insensitive to the specific LLM used and can operate reliably across mainstream backends.

Scalability & Cost. Table 8 summarizes the cost characteristics of three approaches. As expected, a dynamic MAS incurs more runtime and token consumption than a single-agent system. This overhead primarily comes from additional reasoning steps required when an agent evaluates whether delegation is needed—not from the number of agents itself. However, the total expense is determined by the intrinsic complexity of the task, not by the mere presence of additional agents. Table 9 further illustrates task-level cost trends. AGENTHIVE consumes more tokens because it explores more components and expands deeper into nested structures. This extra spending directly corresponds to broader discovery coverage rather than inefficiencies of the MAS itself. Star MAS variants are cheaper, but their fixed structure prevents them from handling deeply nested components, demonstrating that non-recursive or shallow agent hierarchies cannot scale to real-world task.

Table 7: Effect of different backbone LLMs on the overall performance of AGENTHIVE.

Backbone Model	Confirmed Findings	Precision
GPT-4o	33	70.2%
o1	39	77.5%
DeepSeek-v3	31	72.3%
DeepSeek-r1	31	78.8%

Table 8: Per-binary resource footprint across different system designs.

	Single Agent	Star MAS	AGENTHIVE
Runtime (min)	7.3	26.3	38.5
Token Usage (M)	0.25	0.87	1.62
Agents Spawned	1.0	22.4	30.2
Reasoning Steps	29.6	236.3	432.7

Table 9: End-to-end execution cost on SBOM (tokens in millions).

System	Time (h)	Tokens (M)	Tokens / Component
Single Agent	0.10	0.30	0.05
Star MAS	0.90	1.88	0.19
AGENTHIVE	1.40	5.66	0.20

6 Conclusion

We introduce a novel LLM agent framework where delegation is a first-class capability, allowing agents to spawn sub-agents, assign tasks dynamically, and aggregate results recursively. This stage-collapsing MAS adapts its topology at runtime, enabling efficient exploration of complex, hierarchical tasks, such as binary analysis, SBOM detection, and website crawling. By combining delegation, reasoning folding, and dynamic architecture, our system bridges single-agent simplicity and multi-agent power, providing a practical foundation for scalable, real-world AI reasoning.

Limitations

Dynamic MAS Generation. The MAS generation process in our framework is flexible, supporting dynamic depth and breadth of agent deployment based on task requirements. Delegation policies—when an agent spawns sub-agents or halts further delegation—are task-dependent and not hardcoded in the framework. Our four evaluation tasks exemplify practical configurations, demonstrating how application-level decisions can shape MAS behavior. Detailed task-specific delegation parameters are provided in Appendix B-E.

Agent Redundancy. Since delegation decisions are made by LLMs, dynamic MAS execution may create more agents than strictly necessary. This behavior is expected in LLM-driven systems and reflects uncertainty in intermediate reasoning steps rather than a flaw of the framework. Our framework does not attempt to enforce global optimization or cross-agent deduplication, because doing so would constrain the flexibility and generality of delegation-based reasoning. Instead, redundancy control is treated as an application-layer concern. Different tasks have different tolerances for parallel exploration, speculative branching, or multi-path verification, so the most suitable optimization strategies must be selected per task. In our evaluation tasks, redundancy is controlled through lightweight mechanisms embedded in agents' system prompts—such as path pruning, checklist-based progress verification, and duplication detection over navigation states.

Limitations on DAG Topology. While our framework can automatically construct tree, forest, and star topologies, it currently does not support directed acyclic graph (DAG) structures. DAGs introduce multi-parent dependencies, where multiple agents converge to spawn the same sub-agent, creating context management challenges that can lead to inconsistent or duplicated reasoning states. Addressing these issues is a promising direction for future work.

References

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *Preprint*, arXiv:2308.10848.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. 2025. Deepresearch bench: A comprehensive benchmark for deep research agents. *Preprint*, arXiv:2506.11763.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate. *Preprint*, arXiv:2305.14325.

Tianqing Fang, Zhisong Zhang, Xiaoyang Wang, Rui Wang, Can Qin, Yuxuan Wan, Jun-Yu Ma, Ce Zhang, Jiaqi Chen, Xiyun Li, Hongming Zhang, Haitao Mi, and Dong Yu. 2025. Cognitive kernel-pro: A framework for deep research agents and agent foundation models training. *Preprint*, arXiv:2508.00414.

Sidong Feng and Chunyang Chen. 2023. Prompting is all your need: Automated android bug replay with large language models. *arXiv preprint arXiv:2306.01987*.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. 2025. Flowreasoner: Reinforcing query-level meta-agents. *Preprint*, arXiv:2504.15257.

Wil Gibbs, Arvind S Raj, Jayakrishna Menon Vadayath, Hui Jun Tay, Justin Miller, Akshay Ajayan, Zion Leonahenahe Basque, Audrey Dutcher, Fangzhou Dong, Xavier Maso, and 1 others. 2024. Operation mango: Scalable discovery of {Taint-Style} vulnerabilities in binary firmware services. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7123–7139.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Bowei Xia, Tao Sun, Ziyu Ye, Zhaoxuan Jin, Yingru Li, Qiguang Chen, 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. *Preprint*, arXiv:2505.23885.

Shengran Hu, Cong Lu, and Jeff Clune. 2025b. Automated design of agentic systems. *Preprint*, arXiv:2408.08435.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. 2024. Self-evolving multi-agent collaboration

networks for software development. *arXiv preprint arXiv:2410.16946*.

CrewAI Inc. 2024. CrewAI: Fast and Flexible Multi-Agent Automation Framework. https://github.com/crewAIInc/crewAI.git.

LangChain Inc. 2025a. Deep Agents: a standalone library for building agents that can tackle complex, multi-step tasks. https://github.com/langchain-ai/deepagents.git.

LangChain Inc. 2025b. Open Deep Research in LangChain. https://github.com/langchain-ai/open_deep_research.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The hitchhiker's guide to program analysis: A journey with large language models. *arXiv preprint arXiv:2308.00245*.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, and 1 others. 2025. Llm-powered static binary taint analysis. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–36.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2024. A dynamic llm-powered agent network for task-oriented agent collaboration. *Preprint*, arXiv:2310.02170.

Fan Nie, Lan Feng, Haotian Ye, Weixin Liang, Pan Lu, Huaxiu Yao, Alexandre Alahi, and James Zou. 2025. Weak-for-strong: Training weak meta-agent to harness strong executors. *Preprint*, arXiv:2504.04785.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. 25 pages, 9 figures, 2 tables.

Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE.

Yexuan Shi, Mingyu Wang, Yunxiang Cao, Hongjie Lai, Junjian Lan, Xin Han, Yu Wang, Jie Geng, Zhenan Li, Zihao Xia, Xiang Chen, Chen Li, Jian Xu, Wenbo Duan, and Yuanshuo Zhu. 2025. Aime: Towards fully-autonomous multi-agent framework. *Preprint*, arXiv:2507.11988.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms. *Preprint*, arXiv:2501.06322.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. 2024. Sanitizing large language models in bug detection with data-flow. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3790–3805.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. 2025. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *Preprint*, arXiv:2502.04306.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023a. Autogen: Enabling next-gen llm applications via multi-agent conversation framework.

Yizhou Wu, Diyi Yang, Kexin Wang, Vincent Y. F. Tan, Canwen Xu, Zhenyu Yang, Xiang Li, Xiaoxiao Tan, and 1 others. 2023b. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2309.12307*.

Weimin Xiong, Yifan Song, Qingxiu Dong, Bingchan Zhao, Feifan Song, Xun Wang, and Sujian Li. 2025. Mpo: Boosting llm agents with meta plan optimization. *Preprint*, arXiv:2503.02682.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Rui Ye, Shuo Tang, Rui Ge, Yaxin Du, Zhenfei Yin, Siheng Chen, and Jing Shao. 2025. Mas-gpt: Training llms to build llm-based multi-agent systems. *Preprint*, arXiv:2503.03686.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. 2025. Masrouter: Learning to route llms for multi-agent systems. *arXiv preprint arXiv:2502.11133*.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025a. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. 2024a. G-designer:

Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, and 1 others. 2024b. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Wentao Zhang, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. 2025b. Agentorchestra: A hierarchical multi-agent framework for general-purpose task solving. *Preprint*, arXiv:2506.12508.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*.

A AGENTHIVE Architecture Design

This section describes the architectural design and key design decisions of the AGENTHIVE framework, focusing on the foundational abstractions that enable declarative multi-agent system construction.

A.1 Core Abstractions

The framework is built on three core abstractions that together enable flexible, composable multi-agent systems.

Unified Tool Interface. To enable composability and uniform integration, all agent capabilities are encapsulated through a unified tool interface. Each tool declares its identity, a natural-language description, a structured parameter specification, and a single execution entry point that returns a string result. This minimal interface design ensures compatibility with emerging model–tool cooperation standards while maximizing simplicity. Tools receive a context object during initialization that provides access to shared state across the agent hierarchy, enabling stateful operations while maintaining clear boundaries. The framework automatically formats tool metadata into LLM-readable documentation, allowing agents to discover and invoke tools dynamically.

Context Management. The framework employs a flexible context mechanism for sharing state between parent and sub-agents. Rather than enforcing a rigid schema, the context supports dynamic attribute assignment, enabling task-specific context passing tailored to different domains. This design choice accommodates diverse use cases: filesystem operations may require paths and directory information, web crawling tasks may need browser contexts, and document editing tasks may require workspace references. When delegation occurs, the framework creates isolated context copies through deep copying, ensuring that sub-agents cannot pollute parent state while still allowing selective attribute inheritance. This isolation mechanism is critical for enabling safe parallel execution and recursive delegation.

Message and Logging Infrastructure. Conversation history is structured as typed message objects with roles (system, user, assistant, tool) and content payloads. The framework applies content filtering to sanitize sensitive information and normalize outputs before messages enter the conversation history. All messages are logged to structured files in each agent's dedicated directory, enabling complete trace reconstruction for debugging and analysis. This logging infrastructure supports observability across the entire agent hierarchy, making it possible to understand the decision flow even in complex multi-agent scenarios.

A.2 Agent Execution Model

Reasoning Loop Architecture. Agents operate through an iterative reasoning loop that alternates between LLM reasoning and tool execution. Each iteration begins with prompt construction from system instructions, conversation history, and available tool descriptions. The LLM responds with a structured output specifying its reasoning, the selected action (either tool invocation or task completion), and the action parameters. The agent parses this output, validates it against available tools, and either executes the selected tool or terminates if the task is complete. This loop continues until completion or until a maximum iteration limit is reached, ensuring bounded execution.

The framework requires LLMs to produce structured JSON outputs, enabling reliable parsing and tool selection. This design choice trades some flexibility for robustness: while it requires the LLM to conform to a specific output format, it eliminates ambiguity in action selection and enables predictable execution flow.

Tool Execution and Error Handling. When an agent invokes a tool, the framework validates the parameters against the tool's specification before execution. Tool execution is wrapped with timeout

protection to prevent indefinite blocking, with configurable timeout limits (typically 30 seconds by default). If execution fails or times out, the error is captured, logged, and returned to the agent as a structured error message. This approach allows agents to receive feedback about failures and potentially retry with corrected parameters, rather than crashing the entire system. The framework supports both synchronous and asynchronous tool execution modes, enabling optimization for different tool characteristics (I/O-bound vs. compute-bound operations).

A.3 Declarative Configuration

To enable programmatic construction of multi-agent topologies, the framework adopts a declarative configuration approach that separates agent specification from runtime instantiation.

Configuration Structure. Agent configurations specify three key aspects: the agent type, the set of available tools, and execution parameters (system prompt, iteration limits, etc.). The configuration structure supports two types of tool specifications: regular tools that provide domain-specific capabilities, and delegation tools that carry nested sub-agent configurations. This recursive structure is fundamental to the framework's ability to construct arbitrary topologies: a delegation tool's sub-agent configuration can itself contain further delegation tools, enabling multi-level hierarchies.

Builder Pattern for Topology Construction. The framework uses a builder pattern to convert declarative configurations into runnable agent instances. The builder processes tool configurations recursively: for regular tools, it instantiates them directly with the current context; for delegation tools, it creates tool instances that carry their nested sub-agent configurations. When an agent invokes a delegation tool at runtime, the tool uses its stored configuration to instantiate sub-agents on demand. This lazy instantiation design ensures that sub-agents are only created when actually needed, avoiding unnecessary resource allocation for unused delegation paths.

This configuration-driven approach enables developers to specify complex multi-agent topologies declaratively, without writing imperative orchestration code. The framework handles the complexity of agent lifecycle management, context isolation, and result aggregation, allowing developers to focus on defining the logical structure of their multi-agent system.

A.4 Delegation Mechanism

Delegation is implemented as a specialized tool type that wraps sub-agent creation and coordination. From the invoking agent's perspective, calling a delegation tool is identical to calling any other tool: it provides input parameters and receives a result string. Internally, however, the delegation tool orchestrates a multi-agent workflow.

Delegation Execution Flow. When an agent invokes a delegation tool, the tool receives the task parameters and prepares isolated execution contexts for sub-agents. The tool then instantiates sub-agents using their pre-configured specifications, passing each sub-agent its isolated context. Sub-agents execute independently, processing their assigned subtasks and generating evidence strings as results. The delegation tool aggregates these results and returns a compact summary to the parent agent. This aggregation step is crucial for maintaining context efficiency: rather than passing all intermediate reasoning steps upward, only the essential results flow back to the parent.

Recursive Delegation. Since delegation tools are themselves tools, sub-agents can be equipped with delegation capabilities, enabling arbitrary nesting. This recursive design allows the framework to construct hierarchies of arbitrary depth, with the depth controlled by the configuration structure rather than hard-coded limits. Each level of the hierarchy operates in isolation, with context boundaries enforced through the context copying mechanism described earlier. The framework's logging infrastructure ensures that the entire execution trace remains observable despite this hierarchical structure.

B Forest MAS for Binary Vulnerability Discovery

Forest topologies are even less supported in traditional frameworks. Multiple independent trees require coordinating several root agents, which usually demands complex orchestration.

B.1 Customized Toolset

For binary vulnerability discovery, we implement specialized tools that agents use to analyze binaries and coordinate subtasks, as summarized in Table 10.

Table 10: Customized Tools for Binary Vulnerability Analysis

Tool	Purpose	Key Features
`r2`	Reverse engineering wrapper	Persistent session; auto initialization (`aaa`); state persistence; cleanup on exit
`StoreStructuredFindings`	Vulnerability knowledge base	Structured storage (type, function, constraints); JSON persistence; file locking; metadata enrichment
`ParallelTaskDelegator`	Parallel task distribution	Subtask delegation; per-function/module decomposition; lifecycle management; result aggregation
`ParallelFunctionDelegator`	Inter-procedural delegation	Taint tracking; explicit entry points; context passing (registers, global state); callee analysis

Table 11: Customized Agents for Binary Vulnerability Discovery

Agent Type	Role	Key Responsibilities
`FunctionAgent`	Function-level analyzer	Taint tracking; disassembly analysis; vulnerability pattern matching; callee delegation
`KnowledgeAgent`	Findings validator	Vulnerability assessment; false positive filtering; structured storage; severity classification
`ForestAgent`	Root coordinator	Task decomposition; parallel function delegation; evidence aggregation; knowledge base coordination

B.2 Customized Agent

Our binary vulnerability discovery system employs three specialized agent types that form a forest topology, as summarized in Table 11.

B.3 Forest Topology Construction

The forest topology is constructed through declarative configuration rather than explicit orchestration code. The multi-agent structure is specified using nested configuration objects that define delegation relationships and agent capabilities at each level.

Three-Layer Architecture. The forest topology is organized into three hierarchical layers:

1. **Root Layer**: The root agents coordinate the overall binary analysis by decomposing the binary into per-function analysis tasks. Each root agent is equipped with a parallel task delegation capability that distributes function-level analysis to task-layer agents.

2. **Task Layer**: Each task-layer agent focuses on analyzing a single function and carries a parallel function delegation capability that enables inter-procedural analysis. When a function calls other functions, the agent delegates the analysis of callee functions to recursive tree agents.

3. **Recursive Function Tree**: This layer forms a recursive tree structure with configurable depth (typically 4 levels) for tracking taint propagation through function call chains. Each level of the tree analyzes one function in the call chain, and delegates to the next level when further callees need to be analyzed. This recursive structure enables tracking of deep call chains while maintaining isolation between different analysis paths.

The declarative configuration approach allows developers to specify this complex three-layer structure without writing explicit orchestration logic. The framework handles the instantiation, coordination, and result aggregation across all layers automatically.

C Tree MAS for Linux-based Filesystem Analysis

Tree topologies are uncommon in existing frameworks. Flat or star orchestrations dominate; implementing a true tree requires manually coding node expansion policies, branching logic, dependency management, and result aggregation. Deciding when to expand nodes, how to spawn sub-agents, and what results to propagate upward is complex and hard to generalize, so most frameworks rely on bespoke hierarchical planners.

C.1 Customized Toolset

For filesystem analysis, we implement tools that operate on Linux-based filesystem images extracted from firmware. Agents use these tools to explore the filesystem, analyze binaries, manage findings, and search for known vulnerabilities, as summarized in Table 12.

Table 12: Customized Tools for Filesystem Analysis

Tool	Purpose	Key Features
`execute_shell`	Safe shell executor	Restricted command set; directory scope enforcement; read-only operations; path validation
`get_context_info`	Context query	Current file/directory info; relative paths; analysis focus tracking
`StoreStructuredFindings`	Knowledge base writer	Append-mode storage; file locking; automatic context enrichment
`QueryKnowledgeBase`	Knowledge base reader	Field-based queries (`file_path`, `link_identifiers`, `notes`); exact/substring matching
`ListUniqueValuesInKB`	Knowledge base explorer	Unique value enumeration; query formulation support; coverage assessment
`cve_search_nvd`	Vulnerability lookup	NVD API integration; keyword search; CVSSv3 scoring; result ranking
`DeepDirectory-AnalysisAssistant`	Directory delegation	Subdirectory scope enforcement; path validation; context switching; single/parallel modes
`DeepFileAnalysisAssistant`	File delegation	File-level scope isolation; path resolution; context preparation; single/parallel modes

Table 13: Customized Agents for Filesystem Analysis

Agent Type	Role	Key Responsibilities
`ExecutorAgent`	Leaf node analyzer	Tool execution; file/directory inspection; shell command execution; binary analysis
`PlannerAgent`	File-level planner	File analysis coordination; embedded knowledge agent; finding validation; result storage
`KnowledgeBaseAgent`	Knowledge manager	Finding storage/query; duplicate detection; cross-reference support; knowledge base maintenance
`FirmwareMasterAgent`	Root orchestrator	Analysis coordination; verification management; report generation; token tracking

C.2 Customized Agent

Our filesystem analysis system employs a hierarchical tree structure with four specialized agent types, as summarized in Table 13.

C.3 Tree Topology Construction

The tree topology is constructed through declarative configuration, where nested configuration objects define delegation capabilities at each level. The configuration generates a hierarchical structure with controlled depth and dual delegation paths: one for file-level analysis and another for directory-level exploration.

Three-Layer Architecture with Dual Branching. The tree configuration is organized into three distinct layers:

1. **Root Layer**: The root agents explore the firmware root directory and coordinate analysis by delegating to either file analysis or directory exploration based on discovered structure. Root agents are equipped with four types of delegation capabilities: single file analysis, parallel file analysis, single subdirectory exploration, and parallel subdirectory exploration. This dual delegation design allows the system to adapt to different firmware structures, handling both file-heavy and directory-heavy layouts efficiently.

2. **File Analysis Branch**: This branch operates separately from the directory hierarchy and uses a fixed two-level structure for analyzing individual files. The first level coordinates file analysis and delegates to the second level, which performs the actual file analysis without further file-level delegation. This branch also includes embedded function call chain analysis with configurable depth (typically 4 levels), enabling inter-procedural taint tracking within binary files. The function call analysis uses recursive delegation to track taint propagation through nested function calls.

3. **Recursive Directory Expansion**: The directory hierarchy expands recursively through iterative configuration wrapping. Starting from a terminal executor configuration (which cannot delegate directories further), each iteration wraps the previous configuration with a

Table 14: Customized Tools for Academic Writing

Tool	Purpose	Key Features
`add_item`	Content insertion	Chapter/paragraph creation; precise positioning (`insert_after_id`, `insert_before_id`); automatic reference management; Markdown formatting
`delete_item`	Content removal	ID-based deletion; workspace permission checking; recursive chapter deletion; orphan prevention
`get_node_content`	Content retrieval	Dual-format output (JSON structure, Markdown rendering); ID enumeration; hierarchy inspection
`update_block_text`	Content modification	Text rewriting; reference updating; workspace scope enforcement; metadata preservation
`search_report_content`	Content search	Keyword/phrase matching; position tracking; duplication detection; cross-reference lookup
`web_search`	External knowledge	Web crawling; depth-controlled retrieval; headless browser support; fact verification
`FocusedChapterEditor`	Single chapter delegation	Task-specific editing; automatic chapter creation; context isolation; workspace path validation
`ParallelChapterEditor`	Batch chapter delegation	Parallel task distribution; duplicate path detection; efficient structure scaffolding; concurrent writing

Table 15: Customized Agent for Academic Writing

Agent Type	Role	Key Responsibilities
`SelfReviewingAgent`	Two-phase writer	Content creation; self-review; format verification; iterative refinement

new layer that can delegate subdirectory exploration. The iteration continues for a configurable number of levels (typically 3 levels), creating a depth-controlled tree structure. The deepest level serves as the terminal executor, which analyzes directories without further delegation, ensuring bounded exploration depth.

This configuration-driven approach enables the construction of complex tree structures with controlled depth, dual branching paths, and embedded recursive analysis, all specified declaratively without explicit orchestration code.

D Star MAS for Academic Writing

A star topology has one central coordinator and multiple workers. Traditional systems implement this with a dedicated orchestrator that creates workers and merges results, requiring custom code tightly coupled to task logic.

D.1 Customized Toolset

For academic writing, we implement tools that manage hierarchical document structures and delegate writing tasks to specialized agents, as summarized in Table 14.

D.2 Customized Agent

Our academic writing system employs a two-phase self-reviewing agent that forms a star topology, as summarized in Table 15.

D.3 Star Topology Construction

The star topology is constructed through iterative configuration wrapping, where each layer adds delegation capability to the previous layer, forming a depth-controlled hierarchy with a single root coordinator.

Iterative Configuration Assembly. The star configuration is generated through a bottom-up iterative process. The base layer consists of agents equipped only with document manipulation tools (for adding, updating, deleting, and querying document content) but no delegation capability. For each depth level (configurable via a depth parameter), the configuration is wrapped with a new layer that adds delegation capabilities: single-chapter delegation for focused editing tasks and parallel multi-chapter delegation for batch operations. Both delegation types reference the previous layer's configuration, enabling recursive delegation. After the specified number of iterations, the final configuration represents a multi-layer star topology where the root agent can delegate to multiple chapter-scoped workers, and each worker can recursively

Table 16: Customized Tools for Web Crawling

Tool	Purpose	Key Features
`search_online`	Search engine query	Google search integration; top-5 results; title/URL/snippet extraction; error handling
`tavily_search`	Premium search API	Tavily API integration; high-quality results; local SQLite caching; configurable result count
`read_pdf_content`	PDF extraction	Direct PDF download; text extraction via PyMuPDF; content-type validation; image-based detection
`navigate_to_url`	Page navigation	URL loading; network-idle waiting; 30-second timeout; error recovery
`click_element`	Element interaction	Numerical label-based clicking; bounding box mapping; DOM update waiting; validation checks
`type_text`	Text input	Input element targeting; keyboard simulation; focus management; confirmation feedback
`scroll_page`	Page scrolling	Viewport-based scrolling; up/down direction; 80% viewport step; smooth navigation
`get_hyperlinks`	Link extraction	Visible link enumeration; keyword filtering; absolute URL resolution; text/URL matching
`go_back`	History navigation	Browser history traversal; DOM load waiting; URL tracking; error handling
`ParallelTaskProcessor`	Parallel task delegation	Multi-task distribution; isolated browser contexts; independent agent instances; result aggregation

Table 17: Customized Agent for Web Crawling

Agent Type	Role	Key Responsibilities
`WebSearchAgent`	Browser controller	Page navigation; element interaction; screenshot annotation; visual context capture; tool execution

delegate to its sub-chapters down to the leaf level.

E Two-layer star MAS for Web Crawling

E.1 Customized Toolset

For web crawling, we implement tools that enable automated browser interaction and web content extraction, as summarized in Table 16.

E.2 Customized Agent

Our web crawling system employs a specialized browser-aware agent that forms a two-layer star topology, as summarized in Table 17.

E.3 Two-Layer Star Topology Construction

The two-layer star topology is constructed through recursive configuration building, where each depth level adds a delegation tool wrapping the previous level's configuration.

Recursive Configuration Assembly. The star configuration is generated through top-down recursion with a configurable depth parameter. At depth 0 (the base case), the configuration contains only terminal agents with basic browser interaction tools (for web navigation, content extraction, and search) but no delegation capability. For depth greater than 0, the configuration recursively builds a sub-agent configuration for depth-1, wraps it with a delegation tool, and creates a manager configuration that includes both the delegation capability and basic browser tools. After the specified number of recursions, the final configuration represents a two-layer star: the root layer has delegation capability to distribute tasks, and the worker layer consists of terminal agents with only browser tools.