WHEN AGENTS CALL AGENTS: DENIAL-OF-SERVICE RISKS IN MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

One LLM agent can invoke another. It can hand off a sub-task, spawn a worker, or pass a turn in a group chat. Existing DoS-detection tools for LLM agents miss this call, as they all model resource consumption only within a single, isolated agent, leaving potential resource exhaustion undetected when one agent's action triggers, amplifies, or recursively re-invokes work in another. In this paper, we formalize multi-agent resource consumption along six independent dimensions. These are repeat count, concurrency, per-call volume, duration, release policy, and retry pacing. Composing individually bounded agents is not automatically safe. A call graph can stay unbounded even when every agent in it is bounded. This gives fourteen canonical risk patterns. Two of them require at least two independent decision-makers. We corroborate this taxonomy empirically. We built MADScan, a static-analysis-plus-LLM pipeline. It extracts resource-flow and resource-constraint graphs from source code. We applied it to 26 open-source agent systems, spanning single-agent, multi-agent, and hybrid architectures, and audited every candidate with an LLM agent. Only risk points confirmed by an executable proof-of-concept test against the real code are reported. On a benchmark of 314 resource-flow records containing 16 verified risks, MADScan achieves 100% observed precision and 100% observed recall, vastly outperforming state-of-the-art techniques. We detect and report 68 DoS risks (of which 7 have been acknowledged by project maintainers), 15 require multiple agent executions or cross-agent interaction to trigger.

1 INTRODUCTION

Today's large language model (LLM) agents (Yao et al., 2023; Shinn et al., 2023; Yang et al., 2024) are no longer deployed as single, isolated instances. As illustrated in Figure 1, production systems increasingly compose several agents into a multi-agent system (MAS) (Guo et al., 2024) that coordinates through group-chat orchestration, supervisor-worker delegation, handoff, and dynamic sub-agent spawning (Wu et al., 2024; Hong et al., 2024; Li et al., 2023; Qian et al., 2024; Chen et al., 2024). One agent's output may become another agent's input, and one agent's decision can trigger, amplify, or recursively re-invoke work elsewhere in the system. This composition enables a MAS to accomplish tasks that no individual agent could complete alone. But it also means an attacker's influence over one agent is no longer confined to that agent, and can propagate along the call graph to every reachable agent (Gu et al., 2024; He et al., 2025).

Denial-of-service (DoS) is among the most significant threats to AI agent systems, as evidenced by vulnerabilities found in widely used agent applications and warnings in joint cybersecurity guidance on agentic AI (Luo et al., 2026; Canadian Centre for Cyber Security et al., 2026). Unfortunately, existing DoS detection approaches focus only on individual agents, leaving DoS problems in MAS unexplored. AgentDoS (Luo et al., 2026) models an agent's resource lifecycle and uses directed grey-box fuzzing to expose resource-exhaustion vulnerabilities, while AgentFuzz (Liu et al., 2025) combines static localization of vulnerable operators with dynamic testing. These approaches identify risks only within an agent's execution. However, in a MAS, DoS can propagate among interacting agents and disrupt the system as a whole (Zhou et al., 2025), introducing new types of risks that existing work has not investigated. In this paper, we study DoS in MAS through resource exhaustion (Shumailov et al., 2021; Petsios et al., 2017), whose cost can be amplified along the call chain.

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

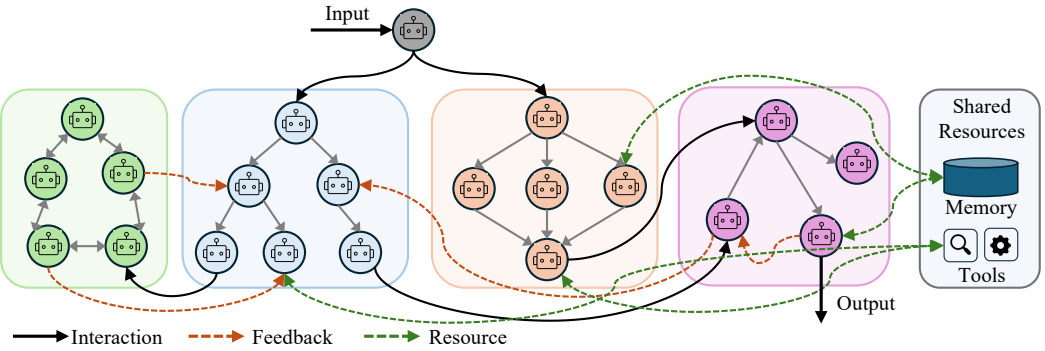


Figure 1: A multi-agent system is far more complex than a single agent. An input is routed to teams of agents, which call agents in other teams, return feedback, and share resources.

Extending detection to MAS presents three major challenges. ❶ No systematic taxonomy of DoS risk in a MAS exists. Hence, the attack surface is unclear, making systematic detection difficult. ❷ Resource consumption is implicit in agent operations, and a single operation can trigger multiple forms of resource usage across agents, rendering cross-agent dependencies and cumulative consumption difficult to trace. ❸ Resource constraints enforced in MAS may be bypassed or fail to cover inter-agent execution, making it difficult to determine whether they effectively bound resource consumption.

We address these challenges through a formal model and a source-guided detection framework. For ❶, we formalize MAS as interacting resource-consuming decision loops and characterize how resource bounds can fail within and across agents. Based on this model, we propose a systematic taxonomy of fourteen canonical DoS risk types organized along six resource dimensions, including two types specific to multi-agent interactions. For ❷, we reconstruct resource flows across agent boundaries from source code: a static scanner first locates the resource sites, and an LLM, told how each framework defines its agents and nodes, turns each site into a flow with its source, destination, and resource type. For ❸, we identify the associated constraints and assess whether their configuration, enforcement, and scope effectively bound those flows. The static scanner locates the constraint sites, and an LLM matches each one to the flow it is meant to bound. We integrate these components into MADScan, which combines static analysis with LLM-based semantic analysis and executable proof-of-concept (PoC) validation. Our evaluation of 26 open-source agent systems, spanning single-agent, multi-agent, and hybrid architectures, identifies 68 DoS risks. We report them, and 7 of which have been acknowledged by project maintainers.

Our contributions. We summarize our main contributions as follows.

- We propose a systematic taxonomy of DoS risks in LLM-based MAS, comprising fourteen risk types across six resource dimensions.
- We develop MADScan, a source-guided framework that combines static analysis and LLM-based semantic analysis to detect DoS risks, generate PoCs, and validate findings.[1]
- We confirm 68 DoS risk points across 26 open-source systems, including 15 that require multi-agent interaction and 7 that are acknowledged by maintainers. On a benchmark of 16 verified risk points, MADScan achieves state-of-the-art results, with 100% observed precision and 100% observed recall, outperforming existing techniques.

2 DoS Risk in MAS: Model and Taxonomy

We consider DoS attacks through ordinary user prompts or attacker-influenced inter-agent messages, without requiring privileged access or model jailbreaks. We extend the single-agent threat model of Luo et al. (2026) to a multi-agent world, and systematically identify DoS risks in MAS.

[1] Anonymized code: `https://anonymous.4open.science/r/MadScan`.

2.1 MAS Model and Resource Cost

Building on LangChain's *Model + Harness* decomposition (LangChain, 2026), we first formalize a single agent as $A = (S, I, O, Ops, \pi)$, comprising its state space, input and output channels, available operations, and model-driven policy $\pi : S \times I \to Ops$. Operations include tool calls (Yao et al., 2023; Qin et al., 2024), file and network access, and agent invocation or spawning. Each autonomous π-loop constitutes one agent, regardless of implementation boundaries.

Further, we model a multi-agent system as $MAS = (\mathcal{A}, G, \mathrm{Sem})$, where $\mathcal{A} = \{A_1, \ldots, A_n\}$ is a set of agents, with $n = 1$ representing the single-agent case. The directed graph $G = (\mathcal{A}, E)$ captures inter-agent calls: an edge $A_i \to A_j$ indicates that an operation of A_i can trigger the execution of A_j, while Sem labels the type of each edge as sync, async, spawn, broadcast, or handoff.

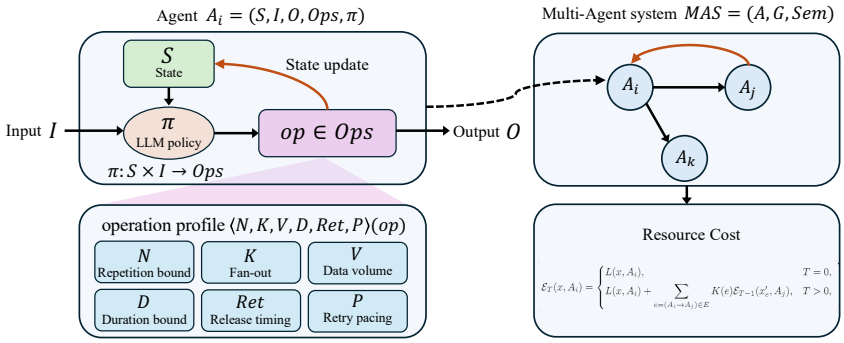


Figure 2: Agent model and compositional resource exposure. Left: an agent's model-driven decision loop and six-dimensional per-operation resource profile. Right: agents form a directed call graph. Processed-volume exposure $\mathcal{E}_T$ aggregates local volume and downstream exposure weighted by edge multiplicity under a hop budget T.

We characterize each operation with six resource dimensions, $op = \langle N, K, V, D, \mathrm{Ret}, P \rangle$ as shown in Fig. 2. N bounds repetitions within the attacker's budget; K denotes concurrent fan-out per invocation, distinct from repetition; V is the data volume processed per call; D bounds how long a call occupies its caller; $\mathrm{Ret} \in \{\text{immediate}, \text{session-end}, \text{never}\}$ specifies resource-release timing; and $P \in \{\text{adaptive backoff}, \text{constant pacing}\}$ specifies retry pacing. For inter-agent calls, each edge e carries a corresponding multiplicity $K(e)$.

To account for cycles, we define processed-volume exposure over a remaining per-path hop budget T. Let $L(x, A_i)$ denote the total volume processed locally by A_i for input x. Then,

$$\mathcal{E}_T(x, A_i) = \begin{cases} L(x, A_i), & T = 0, \\ L(x, A_i) + \displaystyle\sum_{e=(A_i \to A_j) \in E} K(e)\mathcal{E}_{T-1}(x'_e, A_j), & T > 0, \end{cases} \tag{1}$$

where x'_e is the message passed along edge e. Decreasing T ensures a well-founded definition, even in the presence of cycles. With uniformly bounded local volume, sustained fan-out $K > 1$ can yield $O(K^T)$ exposure, whereas without inter-agent edges, $\mathcal{E}_T = L$. Thus, local bounds alone do not ensure a global bound independent of interaction depth.

2.2 MAS DoS Risk Taxonomy

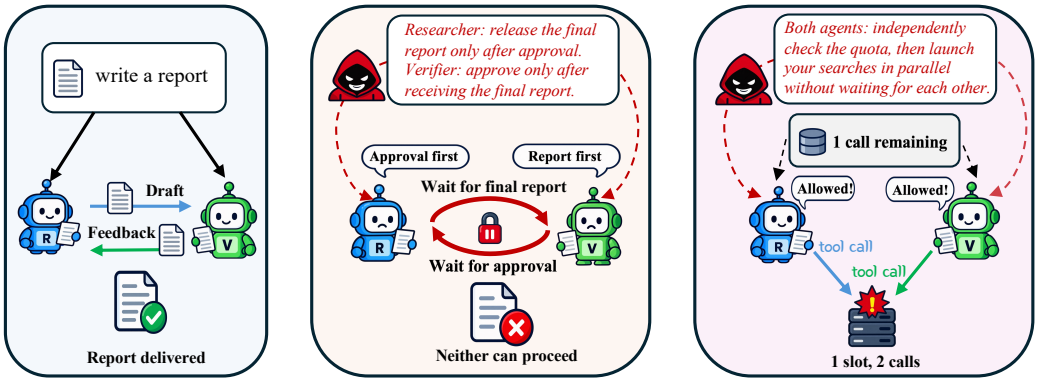
Given our systematic definition of each operation in Section 2.1, we organize DoS risks along the six resource dimensions, distinguishing DoS within a single agent (S) from those arising through multi-agent interaction (M). Composition can amplify resource consumption or change the scope of what a bound must hold. Downstream agents can fan out recursively (K), propagate growing contexts (V), transfer resource-release responsibility (Ret), and compound retries (P). We further distinguish two cross-agent coordination failures (X): inconsistent enforcement of a shared repetition budget (N-X) and circular waiting between agents (D-X). Together, these yield 14 risk types: 6 single-agent types, 6 compositional types, and 2 coordination-specific types (Table 1).

Table 1: Fourteen DoS risk types across six resource dimensions, including single-agent (S), multi-agent (M), and two additional cross-agent coordination types (X).

Dimension	Single-agent	Multi-agent
N (repeat count)	N-S	N-M, N-X†
K (concurrency fan-out)	K-S	K-M
V (per-call volume)	V-S	V-M
D (call duration)	D-S	D-M, D-X†
Ret (release policy)	Ret-S	Ret-M
P (retry pacing)	P-S	P-M

† Coordination-specific types involving multiple autonomous agents under our model.

For the two cross-agent types, D-X occurs when agents form a circular dependency, each waiting for an activation, message, or approval that only another agent in the cycle can provide. Without an effective escape mechanism, the service blocks or is terminated before completing its task. N-X happens when concurrently running agents enforce a shared quota inconsistently, through non-atomic checks or by treating a joint budget as separate local budgets, so that together they exceed it and the cap fails, overspending, or exhausting the shared resource. Figure 3 illustrates normal collaboration and these two types. Appendix A details all fourteen types.



(a) Normal collaboration **(b) Circular-wait deadlock** **(c) Shared-budget race**

Figure 3: Two-agent collaboration and coordination failures. (a) Normal execution completes after approval. (b) D-X: One agent waits for approval while the other waits for the final report. (c) N-X: Both agents observe one remaining call and proceed, exceeding the shared quota.

3 Our System

Given the taxonomy, we develop MADScan to detect and validate concrete instances of the fourteen risk types. As shown in Fig. 4, the system comprises three stages: candidate localization, risk extraction and analysis, and PoC validation. Whereas prior single-agent DoS discovery uses resource-lifecycle modeling to guide grey-box fuzzing (Luo et al., 2026), MADScan uses the taxonomy to guide source-level analysis of resource flows and constraints within and across agents. It then validates candidate risks through targeted executable PoCs.

3.1 Candidate Localization

For each repository, we first select the directories that contain agent, tool, and orchestration-configuration logic. Within this scope, our system actively identifies candidates that comprise two groups: 1) resource sites that cover concurrency, network I/O, file I/O, subprocesses, credentials, and agent definitions; and 2) constraint sites, which cover numeric bounds and slicing or truncation operations. A scanner in our system identifies these candidate sites and records their categories and source locations (file and line numbers) by combining regular expressions and Python ASTs. These

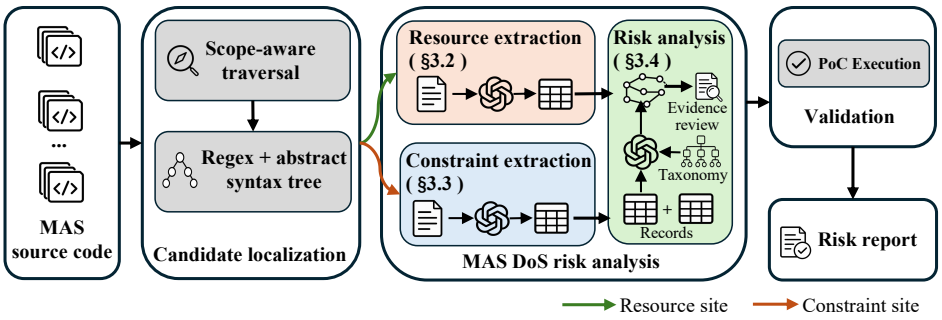


Figure 4: Overview of MADScan. Candidate localization identifies resource- and constraint-relevant code sites through scope-aware traversal and regex-plus-abstract-syntax-tree (AST) scanning. MAS DoS risk analysis extracts resource-flow and constraint records, identifies risks using the taxonomy, and reviews the source evidence for those risks. Validation runs PoCs against the original implementation; only validated findings are included in the risk report.

candidates are then fed into resource and constraint extraction, respectively. Appendix D provides further implementation details.

3.2 Resource & Constraint Extraction

Static matching identifies resource-related code sites but does not resolve their roles in framework-specific agent interactions. To address Challenge ❷, we use an LLM, as in recent LLM-assisted static analysis (Li et al., 2025; Wang et al., 2024), to interpret the candidate sites from Section 3.1, together with a brief repository-specific description of how agents and execution nodes are defined. For example, a framework may represent nodes through explicit registration or ordinary class methods. The LLM produces structured resource-flow records specifying the source, destination, resource type, and underlying system resource consumed. These records capture resource use and transfers within and across agents and serve as input to subsequent risk analysis.

For Challenge ❸, we apply the same LLM-assisted extraction procedure to the constraint sites identified in Section 3.1. For each candidate, the LLM produces a structured record describing the intended resource bound, check logic, enforcement type, and supporting source locations, including default values where available. Constraint extraction runs independently of resource extraction. The resulting records are combined during risk analysis to assess whether the relevant bounds effectively constrain each resource flow.

3.3 Risk Analysis and Validation

Risk analysis. For each repository, MADScan jointly analyzes the resource-flow and constraint records using an LLM, guided by the taxonomy and a manually analyzed example. For each candidate, we specify a potential resource-exhaustion path, the associated constraint gap, its risk type, whether inter-agent interaction is required, and supporting file-and-line evidence. Categorical fields use predefined labels, while descriptive fields explain the path and evidence. We then deploy an LLM agent that reviews each candidate against the source code at the fixed commit, tracing defaults, callers, and other relevant bounds. Unsupported claims are corrected or rejected; retained candidates proceed to PoC validation.

PoC validation. Since the aforementioned analysis is static-analysis-based, it may introduce false positives. To mitigate this issue, our system enforces a validation process in which, for each retained candidate, it executes a targeted PoC against the unmodified implementation. A test harness invokes the relevant function or class with adversarial or boundary inputs and checks whether the observed behavior supports the claimed risk mechanism. A matched control provides an effective bound for testing whether it prevents that behavior. Our system then reviews the results and reports only candidates supported by both source evidence and successful PoC validation.

4 EVALUATION

4.1 EXPERIMENTAL SETUP

Dataset. We collect 26 open-source agent systems, comprising 11 multi-agent, 12 single-agent, and 3 hybrid systems. The dataset combines repositories retained from the AgentDoS evaluation (Luo et al., 2026) with additional repositories selected to broaden coverage of orchestration patterns, including actor-based, graph-based, SOP-driven, hierarchical, and handoff-based designs. Each repository is pinned to a specific commit (the latest), with its clone date recorded for reproducibility. The complete repository list is provided in Appendix C.

Table 2: Composition of the 26 agent systems dataset by architecture.

	Single-agent	**Multi-agent**	**Hybrid**	**Total**
Repositories	12	11	3	26

Baseline Selection. We select baselines based on task relevance, reproducibility, and methodological diversity. AgentFuzz (Liu et al., 2025) and AgentDoS (Luo et al., 2026) serve as detection baselines because they target resource-exhaustion risks in agent systems and provide complementary analysis and testing approaches. We consider CORBA (Zhou et al., 2025) separately as an attack-based reference. It targets contagious recursive blocking rather than broad resource-exhaustion risk detection, so we do not treat it as a baseline for coverage across the full taxonomy.

Evaluation Metrics. We evaluate detection effectiveness using precision and recall. The evaluation unit is a distinct risk point rather than a repository. Detections are matched to verified risk points based on the affected code path and underlying failure mechanism, rather than exact agreement in wording or line numbers. Duplicate detections of the same risk point are counted once. TP denotes correctly detected risk points, FP denotes detections judged unsupported after source review and PoC validation, and FN denotes verified benchmark risk points that a method misses. Precision measures the proportion of detections that are supported, while recall measures the proportion of benchmark risk points detected.

4.2 RQ1: HOW ACCURATE IS MADSCAN?

In the absence of exhaustive ground truth, we construct a benchmark by random sampling. We randomly sample 20% of the 1,570 resource-flow edges (314) extracted by MADScan as shown in Table 4 and manually analyze whether each one exhibits a DoS risk. We also inspect the code around each sampled edge and find no resource-flow edge that the scanner missed. This yields a ground truth of 16 verified risk points.

Table 3: DoS detection performance on manually verified risk points. S and M denote risks arising from single-agent and multi-agent mechanisms.

Tool	**TP**		**FP**	**FN**		**Precision (%)**	**Recall (%)**
	S	**M**		**S**	**M**		
AgentFuzz	4	0	0	7	5	100	25.0
AgentDoS	8	1	0	3	4	100	56.3
MADScan	11	5	0	0	0	100	100

We score MADScan's reported findings, i.e., those confirmed by PoC validation, using the metrics defined in Section 4.1. As shown in Table 3, MADScan detects all 16 benchmark risks without any false positives, achieving 100% observed precision and 100% observed recall.

4.3 RQ2: How Does MADScan Compare with Existing Detectors?

We compare MADScan with AgentFuzz and AgentDoS, following their published procedures and repository instructions. All methods are evaluated against the 16 verified risk points introduced in Section 4.2, comprising 11 risks arising from single-agent mechanisms and 5 from multi-agent mechanisms. Table 3 summarizes the results. AgentFuzz detects 4 single-agent risks and no multi-agent risks, achieving 25.0% overall recall. AgentDoS detects 8 single-agent risks and 1 multi-agent risk, achieving 56.3% overall recall. In contrast, MADScan detects all 16 risks, achieving 100% recall in both categories. The difference is particularly pronounced for multi-agent risks: AgentFuzz misses all 5, while AgentDoS misses 4. These results suggest that explicitly analyzing inter-agent mechanisms improves coverage of risks that the baselines largely miss. However, candidate coverage remains limited by manually selected scan scopes, heuristic regex-plus-AST matching, and LLM judgments; Appendix B discusses these limitations further.

4.4 RQ3: What DoS risks can MADScan identify?

Overall findings. Across 26 repositories, MADScan extracts 1,570 resource-flow edges and 1,387 resource constraints, yielding 147 risk candidates. Source review retains 71 candidates, of which 68 pass PoC validation against the original implementation, as shown in Table 4. The remaining three are rejected because execution reveals effective bounds elsewhere in the call path (Appendix D).

Of the 68 confirmed findings, 15 ($\sim$ 22%) require agent-to-agent interaction to trigger, while 53 ($\sim$ 78%) arise within an individual agent's execution. Thus, approximately one in five confirmed findings in our dataset depends on interactions that isolated single-agent analysis does not capture.

Table 4: Extraction and validation results across 26 repositories.

	Extraction		**Analysis**	**Review**	**Validation**	**Trigger scope**	
	Resource edges	Constraints	Candidates	Retained	Confirmed	Inter-agent	Single-agent
Count	1,570	1,387	147	71	68	15	53

Risk distribution. Two resource dimensions account for two-thirds of the 68 confirmed findings: repetition N ($\sim$ 35.3%) and per-call volume V ($\sim$ 32.4%) together contribute 46 findings ($\sim$ 67.6%) as presented in Table 5. The remaining findings concern concurrency fan-out K ($\sim$ 14.7%), call duration D ($\sim$ 13.2%), and retry pacing P, ($\sim$4.4%). Beyond the 53 single-agent findings, we identify seven multi-agent variants and eight multi-agent-exclusive findings: seven shared-budget coordination failures (N-X) and one circular-wait failure (D-X). Unlike D-M, which concerns unbounded waiting on another agent, D-X involves a cycle of mutually dependent waits.

Table 5: Distribution of the 68 detected DoS risk points over the fourteen canonical types

Dimension	**Single-agent**	**Multi-agent**	**MAS-exclusive**	**Total**
N (repeat count)	N-S: 12	N-M: 5	N-X: 7	24 (35.3%)
K (concurrency fan-out)	K-S: 9	K-M: 1	–	10 (14.7%)
V (per-call volume)	V-S: 21	V-M: 1	–	22 (32.4%)
D (call duration)	D-S: 8	D-M: 0	D-X: 1	9 (13.2%)
Ret (release policy)	Ret-S: 0	Ret-M: 0	–	0 (0.0%)
P (retry pacing)	P-S: 3	P-M: 0	–	3 (4.4%)
Total	53 (77.9%)	7 (10.3%)	8 (11.8%)	68

Four of the fourteen types have no confirmed instance: D-M, P-M, Ret-S, and Ret-M. We read these zeros as limits on what our method can observe, not as evidence that the failure modes are absent. The Ret types concern resources that outlive their intended lifetime, which requires following an allocation across sessions or hand-offs. Our scanner locates bounds on repetition, volume, concurrency, duration, and retry, but does not track object lifetimes, and each proof of concept exercises

a single call rather than accumulation over many sessions. D-M and P-M are inter-agent compositions: they require a wait or a retry policy to be traced across an agent boundary. In the systems we examined, agents mostly hand over control by turn passing or task delegation rather than by blocking one another. Accordingly, every blocking wait we found was on a passive resource such as a model endpoint, an HTTP server, or a queue (D-S), and every retry loop sat inside a single agent (P-S). The one wait between agents, our D-X finding, is not a blocked call but rather a circular activation dependency caused by the scheduler configuration.

Table 6: Impact of the 68 confirmed DoS risk points, and each point is counted once.

	LLM token	**Network**	**Availability**	**Memory**	**CPU**	**Disk**	**Total**
Count	29	12	9	8	7	3	68

Resource impact. LLM token consumption is the most common impact, accounting for 42.6% of the 68 confirmed findings (29), followed by network resource consumption (12). Table 6 shows that the remaining findings primarily affect availability (9), memory (8), CPU (7), and disk (3). Each finding is counted once based on its primary impact, illustrating that agent DoS risks extend beyond model usage to include infrastructure resources and service availability.

4.5 RQ4: How Do Individual Components Contribute?

We evaluate four ablated configurations of MADScan, each disabling one component: resource extraction, constraint extraction, taxonomy guidance, or validation. Table 7 reports their performance on the same benchmark.

Table 7: Ablation results on manually verified DoS risk points.

Configuration	**TP**	**FP**	**FN**	**Precision (%)**	**Recall (%)**
w/o Resource Extraction	12	0	4	100	75.0
w/o Constraint Extraction	11	0	5	100	68.8
w/o Taxonomy	9	0	7	100	56.3
w/o Validation	16	12	0	57.1	100
MADScan	16	0	0	100	100

Removing resource extraction reduces recall from 100% to 75.0%, indicating that resource-flow information helps identify DoS risk paths. Removing constraint extraction lowers recall to 68.8%, highlighting the contribution of identifying constraints that bound resource consumption. Removing taxonomy guidance yields the lowest recall 56.3%, suggesting that structured risk categories improve detection coverage. Removing validation preserves 100% recall but introduces 12 false positives, reducing precision to 57.1%. Together, these results suggest that resource extraction, constraint extraction, and taxonomy guidance improve coverage, while validation filters out false positives, enabling the full pipeline to achieve 100% precision and recall on this benchmark.

5 Case Studies

We examine two cases that instantiate the coordination-related types D-X and N-X. Source excerpts, PoC code, and execution traces are provided in Appendix E.

Case I: Circular activation dependencies in AutoGen. At commit `027ecf0`, AutoGen's `GraphFlow` places incoming edges in a shared activation group with condition `all` by default. Consider $S \to A \to B \to A$ with exit $B \to C$. After S executes, A still waits for activation from B, while B waits for A. Neither becomes ready, yet the scheduler reports "Digraph execution is complete." The graph is user-defined, and the default configuration triggers the failure.

Our PoC in AutoGen 0.7.5 confirms that only S executes: neither A nor B ever runs. The configured `MaxMessageTermination(12)` is never reached because execution ends before further messages are generated. Placing the back edge in a separate activation group with condition `any` allows

execution to continue until the message cap. This contrast identifies the activation configuration as the cause, matching D-X.

Case II: Shared-budget overspending in AutoGPT CoPilot. At commit `bbf396a`, CoPilot's `check_rate_limit` checks recorded daily usage before a turn, while `record_cost_usage` records its actual cost afterward. Concurrent turns can therefore observe the same usage snapshot and all pass admission. We exercise the original functions with unrelated dependencies stubbed and an in-memory Redis substitute. Under a $1.00 daily cap, ten concurrent simulated turns all pass admission and subsequently record $0.30 each, bringing the recorded usage to $3.00. Each turn is an autonomous agent loop, so the ten turns act as ten agents sharing one budget (N-X).

6 Related Work

Resource-exhaustion detection and attacks. AgentDoS studies resource-exhaustion risks in LLM-based agents through resource-lifecycle modeling and directed grey-box fuzzing (Luo et al., 2026). AgentFuzz provides another approach to detecting agent resource-exhaustion risks (Liu et al., 2025). Our work complements these approaches by explicitly modeling resource flows and constraints across autonomous agents, organizing the resulting risks into a unified taxonomy, and validating candidate findings through executable PoCs. Related work constructs prompts that induce excessive resource consumption or disrupt LLM-based systems (Zhang et al., 2024a;b). More broadly, resource-exhaustion research includes fuzzing for algorithmic-complexity and performance issues (Petsios et al., 2017; Lemieux et al., 2018) and energy–latency attacks on neural networks (Shumailov et al., 2021).

Multi-agent systems and security. LLM-based MAS supports conversational coordination, role-based workflows, debate, and agent simulation (Wu et al., 2024; Hong et al., 2024; Li et al., 2023; Qian et al., 2024; Chen et al., 2024; Du et al., 2024; Park et al., 2023). Studies examine their failure modes and communication costs (Cemri et al., 2025; Zhang et al., 2025a). Security research investigates prompt-injection robustness, jailbreak propagation, malicious inter-agent messages, and resilience to faulty agents (Liu et al., 2024; Zhan et al., 2024; Debenedetti et al., 2024; Zhang et al., 2025b; Gu et al., 2024; He et al., 2025; Huang et al., 2025). CORBA demonstrates that adversarial instructions can propagate blocking behavior across a multi-agent system (Zhou et al., 2025), providing evidence of interaction-driven availability risks. Such blocking does not by itself establish the circular wait dependencies required by D-X. Our focus is on identifying the resource-bound failures underlying DoS risks through source analysis and targeted validation.

LLM-assisted program analysis. LLMs have been incorporated into static analysis (Li et al., 2025; Wang et al., 2024) and fuzzing (Xia et al., 2024; Deng et al., 2023). MADScan applies LLM-assisted semantic analysis to resource operations, inter-agent flows, and associated constraints. Its distinguishing focus is taxonomy-guided DoS risk analysis within and across agents, followed by source review and executable validation.

7 Conclusion

We present a taxonomy-guided approach to detecting and validating DoS risks in LLM-based agent systems. Grounded in a formal model of agent composition, our taxonomy organizes fourteen risk types across six resource dimensions. We develop MADScan to analyze resource flows and constraints within and across agents, review source evidence, and validate candidate risks through executable PoCs. On a benchmark of 314 resource-flow records containing 16 verified risks, it achieves state-of-the-art results, with 100% precision and 100% recall, compared with 25.0% and 56.3% recall for AgentFuzz and AgentDoS, respectively. Across 26 open-source systems, MADScan confirms 68 risk points, including 15 whose triggers require agent-to-agent interaction. These results highlight the importance of examining how resource bounds compose across agents, rather than considering local limits alone.

AI USE STATEMENT

We used generative AI assistants, including Claude Code (Anthropic) and ChatGPT/Codex (OpenAI), to assist with polishing and revising the manuscript, developing analysis. The authors reviewed AI-assisted text and code against the research records and supporting evidence. Two authors manually checked every PoC result. The authors take full responsibility for the manuscript, its claims and citations, and all accompanying artifacts.

ETHICS STATEMENT

This research aims to improve the security and reliability of LLM-based agent systems. We analyzed publicly available source code and conducted controlled experiments within authorized environments, with safeguards to limit resource consumption. We did not attack third-party deployments or use unauthorized credentials or infrastructure. Our evaluation was designed to validate risk mechanisms without disrupting services or affecting other users. We recognize the potential for misuse of vulnerability research and support responsible disclosure and remediation.

Vulnerability disclosure. We follow responsible disclosure practices by communicating confirmed findings to the affected maintainers and coordinating the release of technical details to support remediation and reduce misuse risks.

REPRODUCIBILITY STATEMENT

We describe the dataset, implementation details, and evaluation protocols in the main text and appendices. Anonymized code is available at `https://anonymous.4open.science/r/MadScan`.

REFERENCES

Canadian Centre for Cyber Security, Australian Signals Directorate's Australian Cyber Security Centre, Cybersecurity and Infrastructure Security Agency, National Security Agency, New Zealand National Cyber Security Centre, and UK National Cyber Security Centre. Careful adoption of agentic AI, 2026. URL `https://www.cyber.gc.ca/en/guidance/careful-adoption-agentic-ai`. Joint cybersecurity guidance. Last modified September 10, 2026. Accessed September 26, 2026.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, 2025.

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. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, 2024.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, 2024.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 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*, pp. 423–435, 2023.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *International Conference on Machine Learning*, 2024.

Xiangming Gu, Xiaosen Zheng, Tianyu Pang, Chao Du, Qian Liu, Ye Wang, Jing Jiang, and Min Lin. Agent smith: A single image can jailbreak one million multimodal LLM agents exponentially fast. In *International Conference on Machine Learning*, 2024.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 8048–8057, 2024.

Pengfei He, Yuping Lin, Shen Dong, Han Xu, Yue Xing, and Hui Liu. Red-teaming LLM multi-agent systems via communication attacks. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6726–6747, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.349. URL https://aclanthology.org/2025.findings-acl.349/.

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. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Jen-tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Michael Lyu, and Maarten Sap. On the resilience of LLM-based multi-agent collaboration with faulty agents. In *International Conference on Machine Learning*, 2025.

LangChain. Agents. https://docs.langchain.com/oss/python/langchain/agents, 2026. Accessed: 2026-09-17.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 254–265, 2018.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for mind exploration of large language model society. In *Advances in Neural Information Processing Systems*, 2023.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *International Conference on Learning Representations*, 2025.

Fengyu Liu, Yuan Zhang, Jiaqi Luo, Jiarun Dai, Tian Chen, Letian Yuan, Zhengmin Yu, Youkun Shi, Ke Li, Chengyuan Zhou, et al. Make agent defeat agent: Automatic detection of taint-style vulnerabilities in LLM-based agents. In *Proceedings of the USENIX Security Symposium*, Seattle, WA, USA, August 2025.

Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *Proceedings of the USENIX Security Symposium*, 2024.

Jiaqi Luo, Jiarun Dai, Fengyu Liu, Songyang Peng, Youkun Shi, Tong Bu, Geng Hong, Xudong Pan, and Yuan Zhang. Autonomy comes with costs: Detecting denial-of-service vulnerabilities caused by resource abusing in LLM-based agents. In *Proceedings of the USENIX Security Symposium*, 2026.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22, 2023.

Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2155–2168, 2017.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. Sponge examples: Energy-latency attacks on neural networks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 212–231, 2021.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. LLMDFA: Analyzing dataflow in code with large language models. In *Advances in Neural Information Processing Systems*, 2024.

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. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *Conference on Language Modeling*, 2024.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 10471–10506, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.624. URL https://aclanthology.org/2024.findings-acl.624/.

Boyang Zhang, Yicong Tan, Yun Shen, Ahmed Salem, Michael Backes, Savvas Zannettou, and Yang Zhang. Breaking agents: Compromising autonomous LLM agents through malfunction amplification. *arXiv preprint arXiv:2407.20859*, 2024a.

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

Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents. In *International Conference on Learning Representations*, 2025b.

Yuanhe Zhang, Zhenhong Zhou, Wei Zhang, Xinyue Wang, Xiaojun Jia, Yang Liu, and Sen Su. Crabs: Consuming resource via auto-generation for LLM-DoS attack under black-box settings. *arXiv preprint arXiv:2412.13879*, 2024b.

Zhenhong Zhou, Zherui Li, Jie Zhang, Yuanhe Zhang, Kun Wang, Yang Liu, and Qing Guo. CORBA: Contagious recursive blocking attacks on multi-agent systems based on large language models. *arXiv preprint arXiv:2502.14529*, 2025.

A Taxonomy: Detailed Type Descriptions

A.1 The Fourteen Canonical Risk Types

For each resource dimension, S denotes a single-agent failure and M is its multi-agent variant. N-X and D-X additionally distinguish failures of cross-agent coordination. These labels describe resource-bound failure mechanisms; their exploitability and impact require validation in the target system. Where applicable, we assign X rather than M to distinguish shared-budget inconsistency from ordinary repetition and circular waiting from one-way waiting.

N-S: Unbounded local repetition. An effective repetition bound is absent from an agent's own decision loop, allowing repeated execution of an operation without a finite enforced limit.

N-M: Unbounded inter-agent repetition. An effective bound is absent from repeated agent invocations, handoffs, or coordination rounds. Unlike N-S, the counted operations cross agent boundaries.

N-X: Inconsistent shared-budget enforcement. Multiple agents collectively exceed an intended joint repetition budget, so that the cap fails and the shared resource is overspent or exhausted, even though their individual admission checks permit execution. The agents may be distinct roles or concurrent executions of the same agent, since each autonomous π-loop counts as one agent. This can arise from non-atomic check-and-update operations or from treating a shared budget as independent local allowances. The defining condition is inconsistent enforcement of a joint bound; a cycle in the call graph is not required.

K-S: Excessive local fan-out. A single-agent decision initiates more concurrent operations than the intended concurrency bound allows. This concerns the width of a single decision, regardless of how often it repeats.

K-M: Excessive inter-agent fan-out. An agent decision initiates downstream agents without an effective bound on aggregate concurrency. Nested fan-out can multiply the number of concurrent executions when downstream agents independently initiate further calls.

V-S: Excessive per-call volume. An operation processes data without an effective size bound, such as reading an oversized external response into an agent's execution context.

V-M: Unbounded inter-agent payload growth. Messages or accumulated context cross agent boundaries without effective size constraints. Appending histories or combining upstream outputs can enlarge downstream payloads. Repeated transmission of a fixed-size payload increases aggregate processed volume, but does not by itself establish this per-call volume failure.

D-S: Unbounded local waiting. An operation can occupy an agent indefinitely because an effective timeout or cancellation mechanism is absent, for example, when waiting on an unresponsive external service.

D-M: Unbounded inter-agent waiting. An agent waits for another agent without an effective duration bound or escape mechanism. This type covers inter-agent waiting without a circular dependency.

D-X: Circular inter-agent waiting. Two or more agents form a cycle of dependencies, each waiting for an activation, message, or approval that only another agent in the cycle can produce, with no effective timeout, cancellation, or other mechanism that breaks the cycle. Unlike D-M, progress is prevented by mutually dependent waits, and the service either blocks indefinitely or is terminated without completing its task. The relevant cycle is in the wait-for relation; a call-graph cycle alone does not establish D-X. The `GraphFlow` example in Section 5 is an instance: the mutual activation dependency leaves the workflow unable to proceed, and the scheduler ends it prematurely without running the agents in the cycle.

Ret-S: Missing local resource release. An agent retains allocated resources beyond their intended lifetime because release is absent or ineffective, allowing retained resources to accumulate.

Ret-M: Missing release across agent boundaries. Resource ownership or cleanup responsibility is lost or inconsistently assigned during an inter-agent transfer, leaving resources retained beyond their intended lifetime.

P-S: Excessive local retry rate. An agent retries a failing dependency without effective pacing, allowing repeated attempts to impose excessive resource demand. A constant retry interval is not inherently unsafe; the failure depends on whether the resulting rate is adequately bounded.

P-M: Cross-agent retry amplification. Retry policies at multiple agent boundaries interact, allowing an upstream retry to trigger additional downstream retry sequences. Without coordinated pacing or retry budgets, these policies amplify request volume and resource demand across the call chain.

B Discussion and Limitations

Taxonomy Coverage

The fourteen risk types characterize possible resource-bound failures under our model; they are not all empirically confirmed in the evaluated repositories.

Dataset and Analysis Limitations

Our dataset covers diverse agent architectures but is not a statistically representative sample of deployed systems. Detection coverage depends on manually selected scan scopes, heuristic regex-plus-AST matching, and LLM-based interpretation. The scanner may miss unfamiliar framework conventions or resource flows that require deeper interprocedural analysis. LLM outputs may also contain incorrect source locations or unsupported interpretations. Source review and PoC validation strengthen the evidence for reported findings, but do not guarantee complete detection or eliminate selection bias.

Evaluation Scope

Recall is measured against a benchmark rather than exhaustive ground truth. The benchmark consists of the 16 risk points identified by manual analysis of a random 20% sample (314 edges) of the 1,570 resource-flow edges extracted by MADScan. Because the sample is not drawn from MADScan's reported findings, it is not restricted to risks that MADScan detects. However, it covers only edges within the scan scope, leaving risks in code outside that scope unexamined. To check the scanner's coverage, we also inspected the code around each of the 314 sampled edges and found no resource-flow edge that the scanner had missed. With only 16 risk points, each missed risk changes recall by 6.3 percentage points, so both recall and baseline comparisons should be read as estimates on a small sample and may not generalize.

C Dataset Details

Our dataset contains 26 open-source agent systems, combining repositories from the AgentDoS evaluation (Luo et al., 2026) with additional systems selected to broaden architectural coverage. Table 8 lists the evaluated systems.

Table 8: The 26 agent systems evaluated in this study.

Agent system	Agent system
Open Deep Research	Ragflow
DB-GPT	Quivr
deep-research-agent(tarun7r)	Langchain-Chatchat
Gpt Researcher	Khoj
MetaGPT	Kotaemon
Agentscope	Owl
Autogen	MaxKB
OpenAI Swarm	SuperAGI
Agency Swarm	ChuanhuChatGPT
Deer-flow	Agent-zero
DeepResearchAgent(SkyworkAI)	Bisheng
AutoGPT	TaskWeaver
Dify	Langflow

D Pipeline Implementation Details

MADScan uses separate analysis scripts with shared configuration and a common entry point for per-repository or dataset-wide execution. Scan scopes are manually specified after inspecting repository layouts. Candidate localization combines regular expressions with Python AST matching rather than full points-to or taint analysis. LLM outputs follow structured schemas, with predefined labels for risk types. The default model is `gpt-4.1-mini`, with an optional larger model configurable per repository.

Candidates rejected by PoC validation. PoC testing rejected three candidates who passed source review because their cited code omitted relevant behavior elsewhere: (1) A shell-execution dependency enforced a finite default timeout; (2) A parser bug discarded multi-digit identifiers, limiting the tested fan-out to nine; and (3) An apparently uncounted cyclic method was paired with another method counted by the framework's loop guard. These cases show why accurate local citations alone do not establish an unbounded execution path. The parser behavior is an incidental restriction in the tested version, not a reliable resource-control mechanism.

E Case Study Details

This appendix provides source evidence and reproduction details for the cases in Section 5. Code excerpts omit unrelated statements.

E.1 Case I: Circular Activation Dependencies in AutoGen

Source and mechanism. In `_digraph_group_chat.py` at commit `027ecf0`, incoming edges default to a target-specific activation group with condition `all` (lines 25–82). A node becomes ready only after all parents in that group have executed (lines 277–299 and 392–424). An empty ready queue is treated as a normal completion (lines 473–495).

For $S \to A \to B \to A$ with a conditional exit $B \to C$, executing S satisfies only one of A's two incoming dependencies. A still requires B, while B requires A. Neither becomes ready, and the scheduler reports "Digraph execution is complete." The graph is defined by the user or workflow author, and the default configuration triggers the failure without any further attacker input.

Reproduction and control. We use fixed-output agents in AutoGen 0.7.5, without LLM or network calls. Agent B always outputs `continue`; the exit edge requires `stop`. Both configurations use `MaxMessageTermination(12)`. The control changes only the back-edge activation group:

```
# Default
g.add_edge(b, a, condition="continue")
```

```
# Control
g.add_edge(b, a, condition="continue",
           activation_group="loop",
           activation_condition="any")
```

Table 9: GraphFlow execution results. Counts include the user message.

Configuration	Observed sequence	Termination
Default	`user, S`	Reported completion
Control	`user, S, A, B, ...`	12-message cap

This is a configuration-induced circular activation dependency, an instance of D-X. The workflow is terminated prematurely after S, and neither A nor B ever runs, so the requested work is never performed; the failure appears as premature termination rather than indefinite blocking.

E.2 Case II: Concurrent Budget Overspending in AutoGPT

Source and mechanism. In CoPilot's `rate_limit.py` at commit `bbf396a`, `check_rate_limit` checks recorded usage before execution (lines 638–705), while `record_cost_usage` adds the actual cost afterward (lines 806–870). The source explicitly describes the former as a soft check that parallel turns can pass against the same usage snapshot. Although accounting updates are atomic, admission and accounting are separated by the execution of a turn.

Reproduction. The harness loads the original checking and accounting functions, stubs unrelated dependencies, and substitutes an in-memory Redis implementation. Each simulated turn checks admission, waits 0.2 seconds, and then records its configured cost:

```
async def one_turn():
    await rl.check_rate_limit(
        user_id, daily_cap, weekly_cost_limit=-1)
    await asyncio.sleep(0.2)
    await rl.record_cost_usage(user_id, turn_cost)

await asyncio.gather(*(one_turn() for _ in range(10)))
```

With a $1.00 daily cap and ten concurrent turns costing $0.30 each, all ten pass admission. Recorded usage reaches $3.00: three times the cap, exceeding it by $2.00. These costs are test inputs, not actual LLM charges.

The result demonstrates concurrency-amplified overspending under documented soft-limit semantics. Each turn is an autonomous agent loop, so under our model the ten concurrent turns are ten agents sharing one budget, which matches N-X.

Additional component-level reproductions. Table 10 summarizes related check–act failures. These results establish concurrent enforcement failures, not necessarily multi-agent-exclusive triggers.

Table 10: Additional component-level quota and concurrency tests.

Finding	Bounded quantity	Cap	Observed result
AutoGPT P10-1	User executions	3	10/10 admitted
dify N2	App requests	5	19–20/20 admitted
deer-flow P10-1	Sandbox replicas	3	10 created
khoj P10-1	User requests	3	10/10 admitted
kotaemon P10-1	Demo request rate	3	10/10 admitted