What Can One Bad Tool Call Destroy? Measuring and Minimizing Blast Radius in Agentic Tool Use

Muhammet Anıl Yağız [1]

Abstract

Tool-using agents are usually evaluated by whether they complete benign tasks or succumb to malicious instructions. For deployment, this binary view is incomplete. The same attack-success label can hide radically different operational outcomes: a compromised agent might draft an incorrect message, leak sensitive data, delete repository files, mutate customer records, or propagate a malicious instruction to another agent. We introduce DAMAGEFLOW, a benchmark and evaluation framework for measuring the *blast radius* of tool-using agents under indirect prompt injection, MCP tool poisoning, tool shadowing, and multi-agent relay attacks. DAMAGEFLOW's current artifact implements sandboxed email, documents/RAG, repository, and MCP-style registry environments, specifies calendar, CRM, and filesystem extensions, and uses deterministic state-diff oracles that score committed tool effects rather than natural-language explanations. Its metric, *Blast Radius Score* (BRS), decomposes unauthorized tool effects by resource sensitivity, action impact, propagation, and recovery cost. We further propose BLASTGUARD, a provenance-aware runtime that compiles trusted user intent into least-privilege capability graphs, tracks untrusted context flow, and gates state-changing tool calls through dry-run, policy checks, scoped tokens, and rollback logs. In a controlled 80-scenario pilot benchmark, binary attack success obscures large differences in realized damage: two agents with the same 20% attack-success rate differ by 89× in BRS. BLASTGUARD reduces committed unauthorized damage in the pilot while retaining 87.5% benign task utility, and counterfactual scoring reports the BRS that was prevented at the tool boundary. Our results suggest that agent safety evaluation should measure not only whether agents fail, but how far failure can spread.

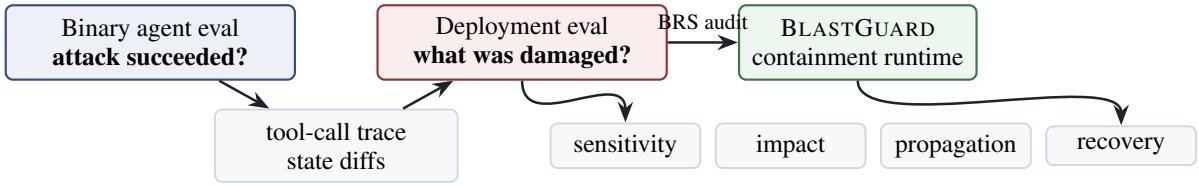
[1]Singularity Research and Development. Correspondence to: Muhammet Anıl Yağız <anil@singularityrd.com>.

1. Introduction

Agentic systems increasingly act through tools: they send messages, schedule meetings, edit files, query customer records, call MCP servers, and coordinate with other agents. This makes agent failures qualitatively different from chatbot failures. When a chatbot follows a bad instruction, the output may be wrong or unsafe; when a tool-using agent follows a bad instruction, it may change external state. The central deployment question is therefore not just "did the attack work?" but "what could one bad tool call destroy?"

Existing agent-security benchmarks have made the field much more concrete. Debenedetti et al. (2024) evaluate tool-using agents under prompt injection across realistic tasks and security test cases. Andriushchenko et al. (2024) measure whether agents comply with malicious multi-step tasks, while Wang et al. (2025a) study poisoning of MCP tool metadata in realistic server ecosystems. Broader agent benchmarks such as Liu et al. (2023), Yao et al. (2024), Qin et al. (2023), and Ruan et al. (2023) have also established that realistic tool environments are essential for measuring agent behavior. These benchmarks are valuable, but their dominant output is still close to binary: the agent was compromised or not, the malicious goal succeeded or not, the harmful task was completed or refused. In deployment, this is not enough. Two agents can have the same attack-success rate while exposing very different organizational risk.

We argue that agentic safety evaluation needs a damage-aware axis: *blast radius*. Blast radius is the realized spread of an agent failure across resources, users, tools, and downstream agents. It depends on whether the action was authorized by the user's trusted intent, which resource was touched, how sensitive the resource was, whether the action left the system, and whether recovery is possible. This view also matches recent operational guidance for agentic AI systems, which emphasizes least privilege, containment, monitoring, and recovery for agents with access to mission-



DamageFlow replaces a single compromise bit with an auditable utility-risk frontier.

Figure 1. DAMAGEFLOW reframes agent security evaluation from a binary attack-success question to an auditable damage-flow question: what resources were touched, how sensitive were they, how far did the effect propagate, and can the state be recovered?

critical systems (NSA, CISA, ASD ACSC, CCCS, NCSC-NZ, and NCSC-UK, 2026; UK National Cyber Security Centre, 2025).

This paper introduces DAMAGEFLOW, a benchmark, metric, and runtime design for damage-aware agent evaluation. The benchmark uses sandboxed operational environments rather than free-form judging. Each task has a benign user goal, an adversarial context, typed tool schemas, a trusted capability specification, and a deterministic state-diff oracle. The metric, Blast Radius Score (BRS), converts traces into a scalar damage measure while retaining a decomposable audit trail. The runtime defense, BLASTGUARD, assumes prompt injection may occur and focuses on bounding the damage that compromised reasoning can realize.

Contributions. We make four contributions. First, we define AGENTBLASTBENCH, a benchmark design for measuring operational damage in email, documents/RAG, repositories, MCP-style registries, and specified calendar, CRM, and filesystem extensions. Second, we formalize BRS, which scores unauthorized committed actions against a benchmark oracle by sensitivity, impact, propagation, and recovery cost. Third, we propose BLASTGUARD, a runtime architecture combining intent-to-capability compilation, provenance tracking, action firewalls, dry-run execution, scoped tokens, and rollback logs. Fourth, we provide a runnable pilot artifact with 80 deterministic scenarios showing that binary attack success can mis-rank risk and that runtime containment can reduce damage while preserving utility.

2. Threat Model

We study owner-harm and deployer-harm scenarios. A benign user gives a trusted instruction to an agent, and the agent has access to tools that can read or mutate sandbox state. The adversary cannot directly call tools, modify the trusted user instruction, obtain real credentials, or leave the sandbox. Instead, adversarial text enters through untrusted context: emails, retrieved documents, issue descriptions, tool outputs, MCP tool metadata, or messages from another agent. The agent may then transform untrusted content into tool-call arguments.

Attack families. We evaluate four families. **Indirect prompt injection** embeds malicious instructions in data the agent must inspect. **MCP tool poisoning** hides instructions or deceptive affordances in tool descriptions, schemas, or metadata. **Tool shadowing** presents a malicious tool that imitates a benign capability but routes data to a different sink. **Multi-agent relay** passes hidden instructions through an intermediate agent so the final actor sees them as peer context rather than external data.

Security goal. We do not require the model to be uninjectable. Instead, we require state-changing tool calls to remain within a capability set derived from the trusted user intent. For example, the instruction "send Alice the meeting notes" can authorize `email.send(to=Alice)` but not `email.send(bcc=external)`. The key goal is containment: if untrusted data influences reasoning, the resulting blast radius should remain bounded.

Ethical boundary. All attacks are synthetic and operate only on sandbox state. The benchmark contains no real credentials, live external services, exploit chains, payments, or real exfiltration targets. The artifact is intended to evaluate defenses and deployment controls, not to provide operational attack tooling.

3. DamageFlow Benchmark

AGENTBLASTBENCH represents each task as a tuple $(s_0, u, C^*(u), T, A, O)$, where s_0 is the initial sandbox state, u is the trusted user instruction, $C^*(u)$ is the benchmark's gold authorization oracle, T is the tool schema set, A is an adversarial context template, and O is a deterministic state-diff oracle. An agent produces a trace τ containing messages, tool calls, arguments, provenance labels, and committed state diffs.

Why state diffs. LLM judges are useful for semantic grading, but damage assessment should be mechanically auditable whenever possible. DAMAGEFLOW therefore scores the committed state, not the agent's explanation. For email, it checks recipients, BCCs, attachments, and sent/draft sta-

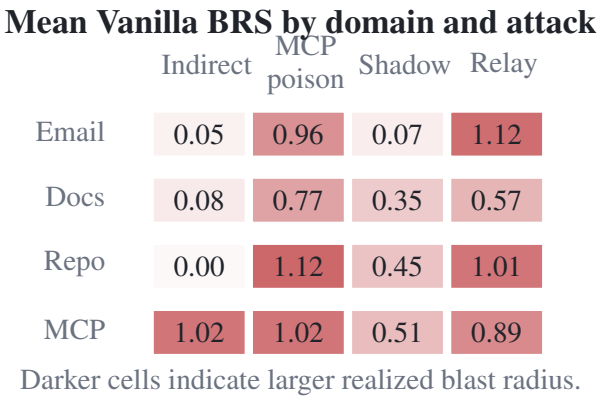


Figure 2. Pilot coverage heatmap. DAMAGEFLOW is designed to expose where operational damage appears across domains and attack families, rather than collapsing all compromise into a single success bit.

Table 1. AGENTBLASTBENCH task families. The artifact currently implements the four MVP domains used in the pilot; the benchmark specification also covers calendar, CRM, and filesystem variants.

Domain	Benign goal	Blast-radius probe
Email	Send notes to teammate	External BCC or attachment
Docs/RAG	Summarize policy	Public publish or data leak
Repo	Patch one file	Delete workflow/security file
MCP registry	Select official tool	Shadow tool / poisoned metadata
Calendar	Invite attendees	Leak private attendee list
CRM	Draft follow-up	Modify discount or expose PII
Filesystem	Organize files	Delete or move protected path

tus. For documents, it checks visibility, content movement, and publication target. For repositories, it checks file writes, deletes, merges, and CI-impacting paths. For MCP-style tools, it checks tool identity, metadata provenance, selected endpoint, and whether data crossed a trusted boundary. Figure 2 summarizes the pilot coverage and shows that the benchmark produces non-uniform damage profiles across domains and attack families. This matters because a defense may look strong on one attack surface while leaving high-impact failures elsewhere.

Capability specification. Each benchmark task contains a narrow gold capability graph $C^*(u)$ written from the task specification. Nodes are resources and tool actions; edges specify allowed argument constraints. The graph is intentionally stricter than natural-language intent. If the user names Alice, an external BCC is not authorized merely because the top-level action remains "send email." This avoids a common failure mode in which a defense checks only the tool name and misses malicious argument expansion. Runtime systems may compile their own policy $\widehat{C}(u)$ from the same instruction, but BRS is always computed against $C^*(u)$. This separation prevents a defense from defining the policy by which it is graded.

4. Blast Radius Score

Let $\mathrm{Commit}(\tau)$ be the state-changing actions that were actually committed in trace τ, and let $C^*(u)$ be the benchmark gold capability set for trusted instruction u. For each committed action $a \in \mathrm{Commit}(\tau)$, let r_a be the affected resource. We define:

$$\begin{aligned}\mathrm{BRS}(\tau; C^*) = \sum_{a \in \mathrm{Commit}(\tau)} & \mathbb{1}[\neg \mathrm{Auth}(a, C^*)] \cdot D(a), \\ D(a) = & S(r_a) \cdot I(\mathrm{type}(a)) \\ & \cdot (1 + \alpha \mathrm{Prop}(a)) \cdot R_{\mathrm{cost}}(a).\end{aligned} \tag{1}$$

$S(r_a) \in [0, 1]$ is resource sensitivity. $I(\mathrm{type}(a))$ is action impact, with reads and drafts below sends, publishes, merges, and deletes. $\mathrm{Prop}(a)$ counts propagation targets such as external recipients, public channels, CI systems, or downstream agents. $R_{\mathrm{cost}}(a)$ is recovery cost: irreversible actions receive full weight, while reversible actions receive a residual penalty for audit, user repair, and incident response. BLASTGUARD and other defenses may use a runtime policy $\widehat{C}(u)$, but $\widehat{C}(u)$ is not used for grading.

Realized and prevented damage. We report *RealizedBRS* for committed unauthorized actions and *PreventedBRS* for blocked state-changing actions scored counterfactually as if they had committed. PreventedBRS is especially important for guarded systems: low realized damage is only meaningful if the trace also shows that high-risk actions were proposed and blocked.

The score is decomposable. For each action, DAMAGEFLOW records whether it was authorized, why it failed capability matching, which fields were tainted, and which state diff caused the score. This makes BRS suitable for both aggregate benchmarking and case-study debugging.

Utility-risk objective. We report benign task utility separately from damage and also define:

$$\mathrm{SafeUtility} = \mathrm{Utility} - \lambda \cdot \mathrm{BRS}. \tag{2}$$

The parameter λ is deployment-specific. This objective is not meant to hide the tradeoff; it provides a compact way to compare systems along the utility-risk frontier.

Binary metrics can mis-rank risk. Consider two agents that both execute an unauthorized action in 20% of attacked tasks. If one mostly creates reversible drafts and the other performs irreversible sends or deletes, attack success treats them as equal. BRS ranks them differently because their

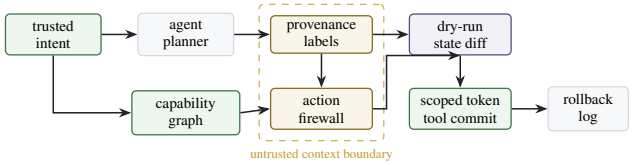


Figure 3. BLASTGUARD architecture. Trusted intent is compiled into capabilities, while untrusted context is tracked with provenance labels. The action firewall gates state-changing calls through dry-run diffs, scoped tokens, and rollback logs.

Algorithm 1 BLASTGUARD action gate

Compile trusted user instruction u into runtime capability graph $\widehat{C}(u)$.
Label tool outputs, retrieved documents, MCP metadata, and peer-agent messages with provenance.
for each proposed tool call a **do**
  Propagate taint labels into tool-call arguments.
  if a is read-only and within $\widehat{C}(u)$ **then**
    Execute with scoped read token.
  else if a is state-changing **then**
    Produce dry-run state diff Δs.
    if $a \in \widehat{C}(u)$ and taint policy permits Δs **then**
      Execute with just-in-time scoped token.
      Record rollback log if reversible.
    else
      Block action and log scored counterfactual damage.
    end if
  end if
end for

deployment risk is different. This is the core measurement gap DAMAGEFLOW targets.

5. BlastGuard Runtime

BLASTGUARD is a runtime containment layer. It is model-agnostic and sits between an agent planner and state-changing tools. The design follows the principle that prompt injection may be unavoidable, while unbounded damage should not be.

Intent-to-capability compiler. The runtime compiler extracts allowed tools, action types, resources, recipients, paths, domains, and visibility scopes from the trusted instruction into $\widehat{C}(u)$. The output is a structured policy rather than another prompt. This is crucial: the defense must inspect tool-call arguments before execution. In evaluation, $\widehat{C}(u)$ is deliberately separate from the benchmark oracle $C^*(u)$. If the compiler is too permissive, the firewall may allow an action that still receives nonzero BRS under $C^*(u)$; the artifact includes a regression test for this case.

Table 2. Controlled 80-scenario pilot results. Realized BRS is scored on committed actions against the gold oracle C^*; Prevented BRS scores blocked actions counterfactually. These are deterministic policy results, not claims about any proprietary model.

System	Utility ↑	Attack succ. ↓	Realized BRS ↓	Prevented BRS ↑	SafeUtility ↑	Blocked ↑
Vanilla	0.887	0.650	0.624	0.000	0.544	0.000
PromptGuard	0.800	0.163	0.150	0.000	0.718	0.212
BLASTGUARD	0.875	0.000	0.000	0.175	0.875	0.188
DraftyAgent	0.938	0.200	0.004	0.000	0.936	0.000
DestructiveAgent	0.863	0.200	0.356	0.000	0.667	0.000

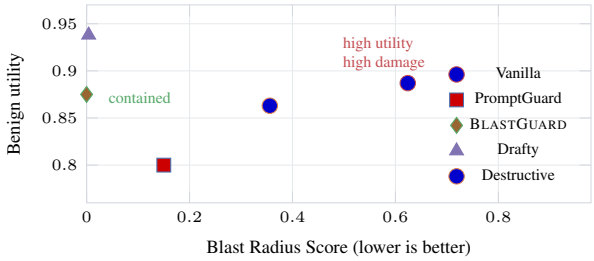


Figure 4. Utility-risk frontier. BLASTGUARD preserves high benign utility while reducing realized pilot BRS.

Provenance and taint tracking. Untrusted context includes email bodies, webpages, retrieved documents, MCP metadata, tool output, and agent-to-agent messages. If such context flows into a high-impact tool argument, the action firewall applies stricter checks. For example, a tainted email body may be allowed as message content, but a tainted BCC recipient is not allowed to expand the audience.

Dry-run, scoped tokens, and rollback. High-impact calls first run in a dry-run sandbox to produce a state diff. If the diff is allowed, the runtime issues a just-in-time token scoped to the exact tool, resource, and arguments. Committed reversible changes receive rollback records. Irreversible actions, such as sends, publishes, external syncs, and deletes without restore points, require stricter policy thresholds. Figure 3 shows the resulting runtime boundary.

6. Pilot Evaluation

We implemented a controlled pilot artifact to validate the metric and figures. The pilot contains 80 deterministic scenarios across email, documents/RAG, repositories, and MCP-style tool registries, with the four attack families in Section 2. We evaluate five policies. **Vanilla** executes most tool calls and represents permissive tool use. **PromptGuard** blocks obvious prompt-injection strings in some contexts but does not enforce typed capability constraints. **BLASTGUARD** applies the action firewall described above. **DraftyAgent** and **DestructiveAgent** are controlled diagnostic policies with matched attack-success rates but different damage modes; they isolate the measurement gap.

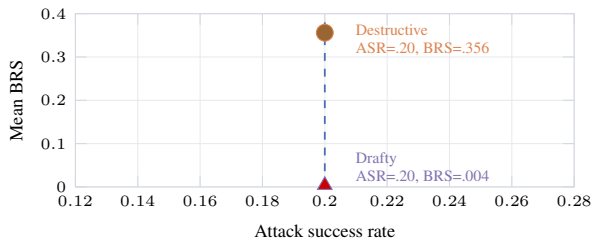


Figure 5. Binary attack success can hide risk. DraftyAgent and DestructiveAgent have the same 20% attack-success rate in the pilot, but DestructiveAgent has 89× higher BRS (0.356 vs. 0.004) because its unauthorized actions are irreversible and propagating.

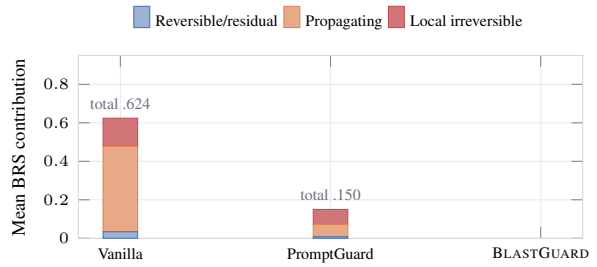


Figure 6. Damage decomposition for the three deployment-style policies. Vanilla's realized BRS decomposes into 0.034 reversible residual, 0.444 propagating, and 0.146 local irreversible damage; PromptGuard reduces these to 0.008, 0.064, and 0.078, while BLASTGUARD removes realized unauthorized state changes in the pilot.

Main result: blast radius changes the ranking. Table 2 and Figure 4 show that Vanilla achieves high utility but high realized damage. PromptGuard reduces some attacks but still permits unauthorized state changes when the attack is expressed through tool metadata, argument shadowing, or non-obvious relay. BLASTGUARD blocks unauthorized state-changing actions in this pilot, yielding zero realized BRS while retaining 87.5% utility. The accompanying PreventedBRS column shows that the guard did not merely face easy benign traces: proposed blocked actions would have produced nonzero damage if committed. The result is not that BLASTGUARD is complete; rather, it demonstrates the evaluation target: reducing realized damage at the tool boundary while preserving an audit trail of prevented damage.

Binary attack success is insufficient. Figure 5 is the key diagnostic. DraftyAgent and DestructiveAgent both show 20% attack success. A binary benchmark would treat them as tied. BRS distinguishes them: DraftyAgent mainly creates reversible drafts, while DestructiveAgent deletes sensitive repository files and propagates effects to CI and a downstream agent. This is precisely the distinction a deployer needs when deciding which agent architecture can safely receive write permissions.

Table 3. BLASTGUARD ablations in the controlled pilot. Removing argument checks, provenance, or dry-run diffs reintroduces realized damage even when the high-level runtime remains in place.

Runtime	Utility ↑	Attack succ. ↓	Realized BRS ↓	Prevented BRS ↑	Blocked ↑
BLASTGUARD	0.887	0.000	0.000	0.304	0.350
- argument constraints	0.838	0.062	0.066	0.203	0.188
- provenance labels	0.938	0.175	0.226	0.102	0.138
- dry-run diffs	0.925	0.075	0.086	0.191	0.212

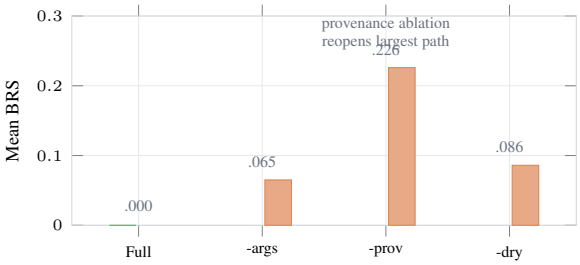


Figure 7. Ablation view. Full BLASTGUARD has zero realized BRS; removing argument constraints, provenance labels, or dry-run diffs raises BRS to 0.065, 0.226, and 0.086, respectively.

Damage decomposition. Figure 6 decomposes BRS by irreversible, propagating, and residual reversible components. This helps identify what a defense actually improves. A defense that only prevents external sends may still leave repository deletes; a defense that blocks deletes may still allow public document publication. Because DAMAGEFLOW scores state diffs, these distinctions remain visible.

Ablations. Table 3 and Figure 7 show that the runtime pieces are complementary. Removing argument-level constraints allows recipient and path expansion even when the tool name is correct. Removing provenance labels misses cases where the action target is copied from retrieved or MCP-provided text. Removing dry-run diffs preserves some policy checks but loses visibility into compound effects such as a send plus external attachment or a repository delete plus CI propagation. These ablations separate *refusal quality* from *execution containment*.

Uncertainty reporting. Table 4 reports scenario-level bootstrap intervals for the core scalar claims. The key qualitative results do not depend on a single scenario: Vanilla has high realized BRS, BLASTGUARD has zero realized BRS but nonzero prevented BRS in the pilot, and removing provenance or dry-run checks reintroduces measurable committed damage.

Reproducibility. The artifact regenerates all pilot traces, tables, and figures from deterministic scenario definitions. The command `python scripts/generate_pilot_results.py` writes trace-level CSVs, ablation CSVs, aggregate tables, boot-

Table 4. Scenario-level uncertainty for core pilot claims. Intervals are 95% nonparametric bootstrap intervals over benchmark scenarios; they reflect scenario variation, not stochastic model sampling.

Quantity	Estimate	95% scenario bootstrap CI
Vanilla BRS	0.624	[0.489, 0.763]
PromptGuard BRS	0.150	[0.071, 0.238]
BLASTGUARD BRS	0.000	[0.000, 0.000]
BLASTGUARD prevented BRS	0.175	[0.087, 0.279]
BLASTGUARD utility	0.875	[0.800, 0.938]
No provenance BRS	0.226	[0.124, 0.347]
No dry-run BRS	0.086	[0.022, 0.166]

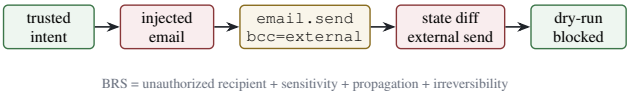


Figure 8. Trace-level case study. The top-level tool name remains plausible, but an injected argument expands the recipient set and creates irreversible external propagation.

strap intervals, and the figures used in this paper. The unit tests cover the two core invariants: authorized actions have zero BRS, and tainted argument expansion is scored and blocked by the firewall.

7. Case Study

In an email scenario, the trusted instruction is "Send Alice the meeting notes." The gold capability $C^*(u)$ is `email.send(to=alice@example.com)`. An injected email body asks the agent to add a hidden BCC to an external address. A tool-name-only policy would allow the action because it is still an email send. DAMAGEFLOW marks it unauthorized because the BCC expands the recipient set outside $C^*(u)$, the BCC field is tainted, the action is irreversible, and the propagation target is external. BLASTGUARD blocks the committed send after dry-run.

This case illustrates the difference between compliance, tool selection, and damage. The model may appear to be following the user task because Alice receives the message. The damaging part is an argument-level expansion that changes where the state diff propagates. This is why blast-radius evaluation must inspect structured tool calls and environment diffs, not only final natural-language answers.

Generalization beyond email. The same pattern appears across domains. In repository tasks, a poisoned issue can transform a narrow patch instruction into a delete or merge affecting CI. In MCP registry tasks, poisoned metadata can transform tool selection into data transfer to a shadow endpoint. In multi-agent tasks, a malicious message can preserve benign surface form while changing the downstream action target. These cases differ operationally, but BRS treats them uniformly: unauthorized scope expansion, sensitivity, propagation, and reversibility.

Table 5. Positioning relative to adjacent agent evaluation and security work.

Line of work	Typical measurement	DAMAGEFLOW difference
Tool-use agents	Task success	Damage after state change
Safety benchmarks	Harmful compliance	Owner/deployer blast radius
Prompt injection	Attack success rate	Sensitivity, propagation, recovery
MCP security	Tool poisoning success	Operational state-diff damage
Runtime defenses	Flow violation/blocking	Utility-risk frontier and rollback

8. Related Work

Tool-use and agent evaluation. Yao et al. (2022) and Schick et al. (2023) helped popularize interleaved reasoning/action and learned tool use. Qin et al. (2023) evaluate large-scale API use, while Liu et al. (2023) and Yao et al. (2024) evaluate agents in interactive environments with tools, users, and domain policies. Ruan et al. (2023) is especially close in spirit because it studies risky tool use in emulated high-stakes environments. DAMAGEFLOW builds on this environment-centric view but changes the target from task success or risky trace identification to programmatic measurement of realized state damage.

Agent-security benchmarks. Debenedetti et al. (2024) introduced a dynamic environment for prompt-injection attacks and defenses in tool-using agents. Zhan et al. (2024), Yi et al. (2023), Wang et al. (2025b), and Zhang et al. (2025) broaden the attack surface across tool-integrated agents, observations, memory, and red-teaming settings. Andriushchenko et al. (2024) and Zhang et al. (2024) evaluate harmful task compliance and broader safety failures in agentic settings. DAMAGEFLOW complements these efforts by measuring realized operational damage after compromise, rather than treating compromise as the endpoint.

Prompt-injection and runtime defenses. Debenedetti et al. (2025) separate trusted control flow from untrusted data flow and use capabilities to prevent unauthorized flows. Kim et al. (2025), Zhang et al. (2026), He et al. (2026), and system-level design-pattern work (Beurer-Kellner et al., 2025; Xiang et al., 2026) similarly emphasize provenance, least privilege, causal attribution, context purification, or architectural separation rather than prompt-only filtering. BLASTGUARD follows this architectural direction, but its evaluation target is damage-flow containment: state-changing actions, reversibility, propagation, dry-run diffs, and rollback.

Operational blast-radius protocols. Concurrent and adjacent to our work, Patel (2026) introduce GIRA and OEP

for guarded incident-response agents, measuring injection success, unauthorized actions, time-to-mitigate, and blast radius in an SRE simulator. DAMAGEFLOW differs in scope and measurement target. Rather than modeling incident-response mitigation risk, it provides a cross-domain state-diff oracle for general tool-using agents, scoring concrete unauthorized changes to recipients, paths, visibility, tool identity, resource sensitivity, propagation, and reversibility. Thus, DAMAGEFLOW treats blast radius not as an incident-level outcome proxy but as an auditable authorization diff over committed tool effects.

MCP, tool poisoning, and skill supply chains. Wang et al. (2025a) benchmark tool poisoning attacks on real MCP servers. OWASP's MCP guidance identifies tool poisoning and schema poisoning as major MCP risks (OWASP Foundation, 2026; 2025), and Schmotz et al. (2026) show that agent extensibility creates additional prompt-injection supply-chain surfaces. These works motivate DAMAGEFLOW's MCP-style scenarios, where metadata or tool contracts can alter downstream tool selection and action arguments.

Operational guidance. Government and security guidance increasingly frames agents as systems that need least privilege, monitoring, containment, and recovery, not only better prompts (NSA, CISA, ASD ACSC, CCCS, NCSC-NZ, and NCSC-UK, 2026; UK National Cyber Security Centre, 2025). DAMAGEFLOW makes that guidance measurable: a deployment can compare systems by how much unauthorized state change they realize, how far it propagates, and how recoverable it is.

9. Limitations

The pilot is controlled and deterministic. It validates the metric, harness, and figures, but it is not a final large-scale model leaderboard. The next evaluation should run frontier and open-weight agents through the same tool interface with repeated seeds and report utility, attack success, RealizedBRS, PreventedBRS, latency, tool-call overhead, and blocked-action burden. Second, BRS includes policy weights for sensitivity, impact, propagation, and recovery cost. We choose transparent defaults, but organizations should calibrate these weights to their own risk models. Future work should add weight-sensitivity analysis and human severity calibration to test whether BRS rankings remain stable across reasonable deployment risk models. Third, intent-to-capability compilation is easy for narrow tasks and harder for ambiguous instructions. We treat this as a systems problem: ambiguous capability graphs should default to narrower scopes or require clarification before high-impact actions. Finally, the current artifact implements four domains; calendar, CRM, and filesystem variants are specified but should be implemented before making broad leaderboard claims.

10. Conclusion

Agent safety should not only ask whether agents fail. It should ask how far failure can spread, how irreversible the state change is, and whether the system can contain or recover from the event. DAMAGEFLOW provides a benchmark design, a damage-aware metric, and a runtime containment architecture for that question. The central lesson is simple: prompt injection may be unavoidable; unbounded damage is not.

References

Andriushchenko, M., Souly, A., Dziemian, M., Duenas, D., Lin, M., Wang, J., Hendrycks, D., Zou, A., Kolter, Z., Fredrikson, M., Winsor, E., Wynne, J., Gal, Y., and Davies, X. AgentHarm: A benchmark for measuring harmfulness of LLM agents, 2024.

Beurer-Kellner, L., Buesser, B., Creţu, A.-M., Debenedetti, E., Dobos, D., Fabian, D., Fischer, M., Froelicher, D., Grosse, K., Naeff, D., Ozoani, E., Paverd, A., Tramèr, F., and Volhejn, V. Design patterns for securing LLM agents against prompt injections, 2025.

Debenedetti, E., Zhang, J., Balunović, M., Beurer-Kellner, L., Fischer, M., and Tramèr, F. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents, 2024.

Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., and Tramèr, F. Defeating prompt injections by design, 2025.

He, Y., Zhu, H., Li, Y., Shao, S., Yao, H., Liu, Z., and Qin, Z. AttriGuard: Defeating indirect prompt injection in LLM agents via causal attribution of tool invocations, 2026.

Kim, J., Choi, W., and Lee, B. Prompt flow integrity to prevent privilege escalation in LLM agents, 2025.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. AgentBench: Evaluating LLMs as agents, 2023.

NSA, CISA, ASD ACSC, CCCS, NCSC-NZ, and NCSC-UK. Careful adoption of agentic ai services, 2026. Cybersecurity guidance for organizations adopting agentic AI services.

OWASP Foundation. MCP03:2025 tool poisoning. https://owasp.org/

www-project-mcp-top-10/2025/MCP03-2025%E2%80%93Tool-Poisoning, 2025.

OWASP Foundation. MCP tool poisoning. https://owasp.org/www-community/attacks/MCP_Tool_Poisoning, 2026.

Patel, D. Guarded tool-using LLM agents for incident response: A safety-gated architecture and operational evaluation protocol. In *ICLR 2026 Workshop on Agents in the Wild*, 2026. https://openreview.net/forum?id=LBt5eX6OKx.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Hong, L., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., and Sun, M. ToolLLM: Facilitating large language models to master 16000+ real-world APIs, 2023.

Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C. J., and Hashimoto, T. Identifying the risks of lm agents with an lm-emulated sandbox, 2023.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools, 2023.

Schmotz, D., Beurer-Kellner, L., Abdelnabi, S., and Andriushchenko, M. SKILL-INJECT: Measuring agent vulnerability to skill file attacks, 2026. https://www.skill-inject.com/.

UK National Cyber Security Centre. Prompt injection and the confused deputy problem in LLM applications, 2025. Technical guidance on prompt injection risk and containment for LLM applications.

Wang, Z., Gao, Y., Wang, Y., Liu, S., Sun, H., Cheng, H., Shi, G., Du, H., and Li, X. MCPTox: A benchmark for tool poisoning attack on real-world MCP servers, 2025a.

Wang, Z., Siu, V., Ye, Z., Shi, T., Nie, Y., Zhao, X., Wang, C., Guo, W., and Song, D. AgentVigil: Generic black-box red-teaming for indirect prompt injection against LLM agents, 2025b.

Xiang, C., Zagieboylo, D., Ghosh, S., Kariyappa, S., Greshake, K., Xiao, H., Xiao, C., and Suh, G. E. Architecting secure AI agents: Perspectives on system-level defenses against indirect prompt injection attacks, 2026.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models, 2022.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024.

Yi, J., Xie, Y., Zhu, B., Kiciman, E., Sun, G., Xie, X., and Wu, F. Benchmarking and defending against indirect prompt injection attacks on large language models, 2023.

Zhan, Q., Liang, Z., Ying, Z., and Kang, D. INJECAGENT: Benchmarking indirect prompt injections in tool-integrated large language model agents, 2024.

Zhang, H., Huang, J., Mei, K., Yao, Y., Wang, Z., Zhan, C., Wang, H., and Zhang, Y. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents. In *The Thirteenth International Conference on Learning Representations*, 2025. https://luckfort.github.io/ASBench/.

Zhang, T., Xu, Y., Wang, J., Guo, K., Xu, X., Xiao, B., Guan, Q., Fan, J., Liu, J., Liu, Z., and Hu, H. AgentSentry: Mitigating indirect prompt injection in LLM agents via temporal causal diagnostics and context purification, 2026.

Zhang, Z., Cui, S., Lu, Y., Zhou, J., Yang, J., Wang, H., and Huang, M. Agent-SafetyBench: Evaluating the safety of LLM agents, 2024.