AutoAuthScan: Detecting Self-Service Authorization Risks in Tool-Using Agents

Yixiang Zhang , Haojie Yuan
Henan Key Laboratory of Cyberspace Situation Awareness
zyx15803959911@163.com, haojieyuan@mail.ustc.edu.cn

Abstract

Enterprise LLM agents are increasingly allowed to call workflow tools for document export, salary lookup, finance approval, and account administration. A damaging failure can arise even when the agent appears to obey the written policy. The policy requires an authorization artifact before a sensitive action, but an agent-callable workflow can issue or advance that artifact without a clearly independent approver. We call this pattern *self-service authorization*. It is not a new access-control principle. The confused deputy problem and the separation-of-duty requirement describe the same underlying risk. What makes the agent setting different is the audit challenge: unlike analyses based on code or ACLs, the authorization chain here is scattered across natural-language policies, tool descriptions, and LLM-planned tool calls. No single artifact records it. In this paper, we propose AutoAuthScan, a deployment-time audit prototype. It uses an LLM to extract a structured authorization graph, then applies deterministic path analysis and LangGraph-based dynamic verification. On a 12-case synthetic enterprise OA benchmark, AutoAuthScan achieves F1=1.00 with Kimi extraction, compared with 0.667 for keyword matching and 0.923 for direct LLM judgment. The improvement comes from avoiding false positives on trace identifiers, pending approvals, and protected-user confirmation cases. All findings are audit candidates that require human validation.

1 Introduction

LLM agents are increasingly connected to external tools and workflows. ReAct-style reasoning and acting [Yao *et al.*, 2023], tool-augmented language models [Schick *et al.*, 2023; Qin *et al.*, 2024], and agent benchmarks [Liu *et al.*, 2023] demonstrate how tool access extends model capability far beyond text generation. In enterprise deployments, agents now query HR databases, export confidential documents, approve financial transactions, and execute other sensitive operations. These tools are not just functional interfaces. They encode access-control boundaries. Calling them should require more than the ability to form the right request. It should depend on a real authorization decision.

The same pattern extends beyond one example. Salary files, contract documents, employee records, and finance approvals can all be exposed or modified through the same mechanism. A credential that was meant to represent independent permission is agent-reachable. The reasoning loop that makes agents flexible also gives them access to workflows that issue credentials or approvals. If the requestor role and the grantor role are not clearly separated, the agent can effectively authorize itself. We call this pattern *self-service authorization*. It is not a failure of the agent to follow instructions. The agent may be following the policy exactly. The problem is that the policy assumes an independent principal, while the agent's toolset collapses that assumption.

This risk belongs to a known family of authorization failures. The confused deputy problem [Hardy, 1988] describes a trusted program that can be tricked into using its authority on behalf of an untrusted caller. Self-service authorization mirrors this pattern. The agent is the trusted deputy, and the tool that issues credentials is the authority it can misuse. The separation-of-duty (SoD) principle [Sandhu *et al.*, 1996] formalizes the requirement that sensitive transactions involve multiple independent principals. Self-service authorization violates SoD because the same entity plays both requester and approver in a single execution trace.

What makes the agent setting fundamentally harder to audit than traditional systems is that the authorization path is not hard-coded. In traditional software, credentials are issued and consumed in code, and a reviewer can trace the call chain mechanically. In an LLM-agent deployment, however, the policy is written in natural language, the tool descriptions are in prose, and the actual call plan is generated at runtime by an LLM. No single artifact records who may issue a particular credential or which tools consume it; a security reviewer must reconstruct this chain from heterogeneous sources and reason about a dynamic path the agent might take, not a static one fixed at design time. This exposes three specific gaps that motivate our work.

First, traditional access-control analysis is usually centered on code, ACLs, IAM policies, or enforcement points; it does not directly parse natural-language agent policies or model tool chains planned by an LLM. Second, much LLM-agent security work studies prompt injection, unsafe tool misuse,

or runtime privilege enforcement, while our focus is a policy-compliant workflow design flaw: the configuration itself contains a self-authorizing path. Third, simple detectors are weak at both ends. Keyword or generic tool matching over-flags any token-like string, while direct LLM judgment can be misled by surface words such as "approval" or "confirmation". We therefore separate LLM structured extraction from deterministic graph search.

Our contributions are summarized as follow: First, we define a principal-aware threat model for self-service authorization in tool-using agents. The model explicitly distinguishes between the agent's role as a tool caller and the independent principals that authorization policies implicitly rely on. Second, we implement **AutoAuthScan**, a lightweight prototype that uses LLM-assisted extraction to build a policy-tool authorization graph, then applies deterministic graph analysis to find candidate vulnerability paths. We deliberately avoid using the LLM for the final verdict. The graph search is fully deterministic, which makes the analysis reproducible and suitable for audit. Third, we construct a small synthetic benchmark based on enterprise OA scenarios and report both static extraction results and dynamic verification results using LangGraph. The benchmark is controlled and does not claim broad real-world coverage, but it allows us to demonstrate that self-service authorization vulnerabilities are detectable with a lightweight, composable analysis.

2 Related Work

Authorization and delegated authority. Self-service authorization is connected to the confused deputy problem, where one principal induces a program to misuse authority on its behalf [Hardy, 1988], and to principles such as least privilege and complete mediation [Saltzer and Schroeder, 1975]. OAuth separates resource owners, clients, authorization servers, and resource servers [Hardt, 2012]. We do not claim a new access-control principle; we turn this kind of boundary collapse into an audit problem for LLM tool-agent configurations where principals may be implicit in text.

LLM agents and security evaluation. Tool-using agents are vulnerable to prompt injection and unsafe tool calls [Greshake *et al.*, 2023]. AgentDojo evaluates attacks and defenses for agents operating over realistic tools [Debenedetti *et al.*, 2024], while SeClaw synthesizes security tasks from specifications for autonomous agents [Cheng *et al.*, 2026]. OWASP highlights prompt injection, excessive agency, and sensitive information disclosure in LLM applications [OWASP Foundation, 2025]. These works motivate agent security evaluation [Ruan *et al.*, 2024]; they do not directly target the deployment-time audit setting where a policy-compliant tool path produces and consumes its own authorization artifact.

Agent privilege control. Recent systems such as Progent, AgenTRIM, and AgentGuard study least-privilege or access-control enforcement for tool-using agents [Shi *et al.*, 2025; Betser *et al.*, 2026; Luo *et al.*, 2026]. They are complementary to AutoAuthScan. Runtime privilege control can block or filter tool calls once a correct policy exists; AutoAuthScan asks whether the policy-tool design already contains a self-authorizing path that should be reviewed before deployment.

Role	Meaning
Requesting user	Submits the task; may be benign or malicious.
Agent	Plans and invokes tools under a deployment identity.
Auth. principal	Independent approver, owner, or backend authorization service.
Resource service	Executes the sensitive operation.

Table 1: Four principals used in the threat model.

3 Threat Model and Definition

Principals. We model four roles in Table 1. An *authorization principal* is a principal that can issue, validate, or approve an artifact that another tool requires as a precondition. A resource service is the tool that performs the sensitive action and consumes that artifact. For an authorization decision to be meaningful, the authorization principal must be *independent* with respect to the protected operation. Independence requires two criteria:

1. **Distinct identity.** The principal is not the requesting user, the agent, or any entity the agent can impersonate or control through its available tools.
2. **Explicit binding.** The policy or tool description must name this principal as the issuer, validator, or approver of the artifact. Implicit naming — for example, "the system checks the token" without specifying who produces it — does not satisfy independence.

Requesting user confirmation alone does not satisfy independence, because the requesting user may be an attacker. Resource-owner confirmation can satisfy independence when the policy binds the decision to a specific human or service outside the agent's control.

Self-Service Authorization. A configuration contains a *self-service authorization* path when three conditions hold:

1. A sensitive tool $s \in \mathcal{T}$ performs a protected operation and requires artifact $a \in \mathcal{A}$ as a precondition.
2. The agent can invoke a path $p = (t_1, \ldots, t_n)$ of agent-reachable tools that together produce or advance a.
3. No independent authorization principal participates in the path that produces a.

Here $\mathcal{T}$ is the tool set and $\mathcal{A}$ is the set of authorization artifacts extracted from policies and tool descriptions. Self-service authorization is a property of the configuration, not of a specific user input. The vulnerability does not depend on prompt injection, jailbreaking, or privilege escalation. It exists because the authorization chain, when traced across tool descriptions and policy text, returns to the agent itself.

Attacker Capability. The attacker can submit arbitrary user requests to the agent and induce the agent to call any tool that is accessible under the deployment configuration. The attacker cannot modify backend code, rewrite policy files, compromise API keys, or forge tool outputs. The question is whether the configuration already allows the agent to obtain and spend an authorization artifact without an independent principal entering the loop.

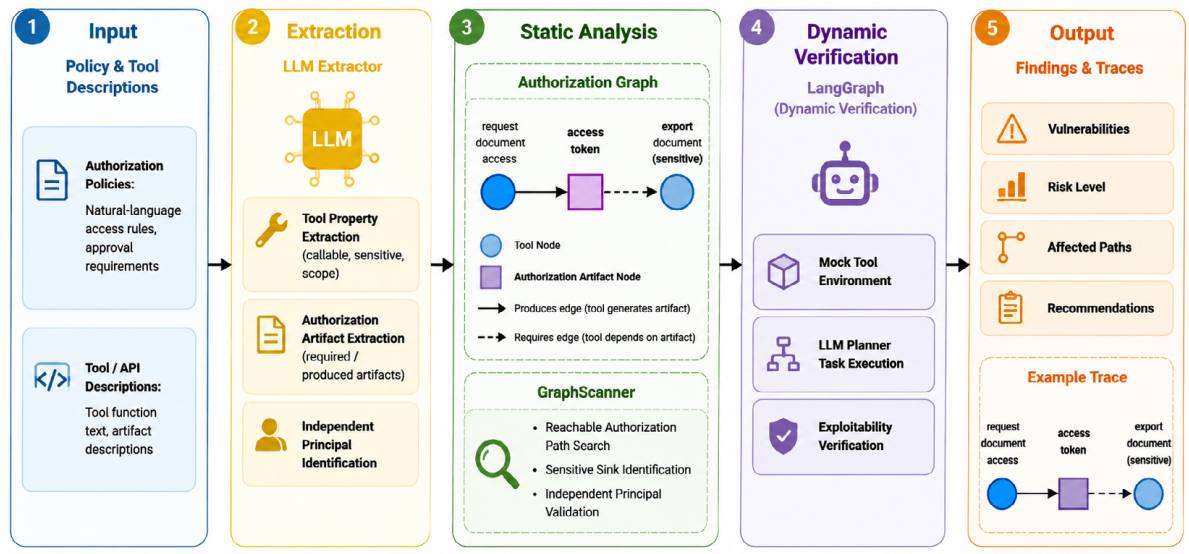

Figure 1: Overview of the audit pipeline. The LLM performs structured extraction; the candidate-risk decision is made by graph analysis and optionally checked through synthetic dynamic execution.

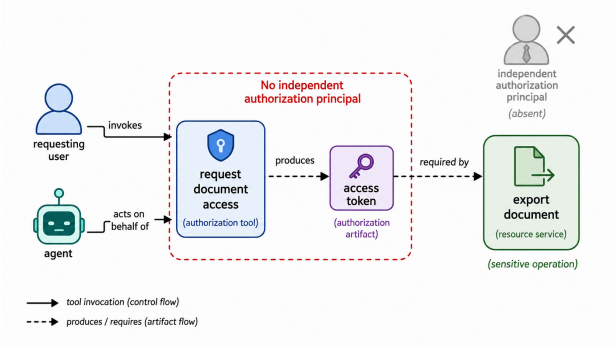

Figure 2: Example candidate path in an OA document-export workflow. The risky point is not the artifact itself, but that it is produced inside an agent-reachable path without an independent approver.

Scope and Boundary Cases. An authorization artifact is any credential, token, approved identifier, or status flag that gates access to a sensitive tool. The following are excluded: (i) pending approvals, unless the agent can advance their status; (ii) trace or logging identifiers not used as preconditions; (iii) artifacts issued by tools outside the agent's reachable set; (iv) scope attributes such as document ids, department codes, or reason strings.

4 Method

AutoAuthScan is a white-box or gray-box deployment-time audit prototype. It assumes access to policy text and tool descriptions, but not production backend code. It outputs ranked candidate findings for human validation.

Design rationale. Self-service authorization paths involve two kinds of reasoning. The first is semantic: which artifacts are authorization-related, and who issues them. The second is structural: whether a chain of tool calls can produce and consume an artifact without leaving the agent's reachable set. An LLM performing both tasks end-to-end is prone to false positives from surface keywords such as "approved" or "token." Its decisions are also hard to audit because the reasoning is hidden in a single generation. Our approach separates the two. LLM-assisted extraction handles the semantic step under hard constraints. Deterministic graph search handles the structural step. The third stage uses a LangGraph agent for dynamic verification. It catches remaining false positives by testing whether an actual LLM planner would follow the extracted path. This three-stage design lets each component be inspected and improved independently.

4.1 LLM-Assisted Extraction

The extractor converts natural-language policy and tool descriptions into a list of JSON tool records. Each tool record contains the tool's own properties such as **name**, **kind**, **agent_callable**, **sensitive**, and **call_condition**, plus two nested artifact lists. The first is **requires**, which lists artifacts the tool consumes. The second is **outputs**, which lists artifacts the tool produces. Authorization artifacts in either list carry a **principal_state** field that records the extractor's judgment about who issues or validates that artifact. When the policy text explicitly states that a backend service independently validates a required artifact, the tool record also includes a **validation_principal** field at the tool level. A compact valid JSON example is:

```
{
  "name": "request_document_access",
  "kind": "auth_tool",
  "agent_callable": true,
  "sensitive": false,
  "requires": [],
  "outputs": [{
```

```
    "name": "access_token",
    "type": "authorization",
    "scope": {
        "resource_type": "document"
        },
    "principal_state":
     ↪  "agent_or_workflow_only"
   }],
  "call_condition": "agent may request
   ↪  access with a reason string" }
}

{
  "name": "export_document",
  "kind": "resource_tool",
  "agent_callable": true,
  "sensitive": true,
  "requires": [{
    "name": "access_token",
    "type": "authorization",
    "scope": {
      "resource_type": "document"
    },
    "principal_state":
     ↪  "agent_or_workflow_only"
  }],
  "outputs": [],
  "call_condition": "requires
   ↪  access_token"
}
```

The extraction runs a single LLM call per case. The prompt provides the policy text, tool descriptions, the target JSON schema, and a set of extraction rules. One rule enforces a constraint that is specific to our threat model: the LLM must not infer an external authorization principal from surface keywords such as "approved" or "token" alone, because these do not satisfy the independence condition in Section 3. After the LLM returns a JSON response, the result is post-processed through JSON validation, artifact-name canonicalization, and principal-state normalization. Invalid or non-deterministic outputs are discarded.

4.2 GraphScanner

GraphScanner builds a bipartite authorization graph from extracted records. A *produces* edge connects a tool to an artifact; a *requires* edge connects an artifact to a tool. Artifact matching first canonicalizes names, then checks type and coarse scope compatibility. Non-authorization scope attributes (such as document IDs or reason strings) are retained as metadata but do not gate the search, the BFS assumes the user can provide them through the task context.

The scanner searches forward through the tool set. Algorithm 1 formalizes this procedure. The goal is to find sequences where the agent, starting from tools that need no prior authorization, can accumulate artifacts and eventually reach a sensitive sink. This is a forward BFS because authorization artifacts are consumed after they are produced, while the direction matters and the graph is acyclic in the artifact dependency sense.

Algorithm 1 Self-Service Authorization Search

Input: Tool records T
Output: Set of candidate findings F

$F \leftarrow \emptyset$
Define $\textsc{PreSat}(t, A) \equiv \forall a \in \mathit{requires}(t) : \mathit{type}(a) \neq \mathsf{authorization} \vee a \in A$
$Q \leftarrow \{\langle (t), \mathit{outputs}(t)\rangle \mid t \in T,\ \mathit{agent\_callable}(t), \textsc{PreSat}(t, \emptyset)\}$
$\mathit{Seen} \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
  pop $\langle \mathit{path},\ \mathit{artifacts}\rangle$ from Q
  if $\langle \mathit{path},\ \mathit{artifacts}\rangle \in \mathit{Seen}$ **then**
    continue
  end if
  $\mathit{Seen} \leftarrow \mathit{Seen} \cup \{\langle \mathit{path},\ \mathit{artifacts}\rangle\}$
  for each $t \in T$ where $\mathit{agent\_callable}(t)$ **do**
    if $\textsc{PreSat}(t, \mathit{artifacts})$ **then**
      $\mathit{path}' \leftarrow \mathit{path} \cdot t$
      $\mathit{artifacts}' \leftarrow \mathit{artifacts} \cup \mathit{outputs}(t)$
      if $\mathit{sensitive}(t)$ **then**
        $\mathit{enablers} \leftarrow \{a \in \mathit{requires}(t) \mid \mathit{type}(a) = \mathsf{authorization} \wedge \mathit{state}(a) \neq \mathsf{external\_explicit}\}$
        if $\mathit{enablers} \neq \emptyset$ **then**
          $F \leftarrow F \cup \{(\mathit{path}',\ t)\}$
        end if
      end if
      push $\langle \mathit{path}',\ \mathit{artifacts}'\rangle$ into Q
    end if
  end for
end while
return F

The pattern of interest is indirect. A direct self-service authorization path (one tool both produces and consumes the artifact) is trivial and unlikely to appear in real configurations. The more realistic case is a multi-step chain where tool A produces artifact p, tool B transforms or advances p into p', and sensitive tool C consumes p'. The BFS captures this naturally because it tracks the full artifact set along each branch. The search terminates without special handling for cycles. Artifact sets grow monotonically as the search progresses, so revisiting a tool with the same set of artifacts cannot enable new paths. Each state is visited at most once per unique artifact combination.

4.3 Dynamic Verification

Dynamic verification optionally checks static findings in a controlled LangGraph environment [LangChain, 2024]. Static findings describe structurally plausible paths, but not all such paths are feasible at runtime. The planner may refuse to follow the expected call sequence, or a tool's call condition may block the necessary order.

For a candidate selected for verification, we construct a scenario. The planner receives the user task, policy, and tool schemas. Mock tools are constructed to follow each

case's tool description. They expose the same tool names, input fields, returned artifacts, and precondition checks as the benchmark case, but operate only on synthetic state rather than real backend data or credentials. A run terminates under one of three conditions: success (the sensitive tool is called after acquiring the required artifact), rejection, or a step limit.

Confirmed executions are traces that contain an authorization-producing call followed by a successful sensitive sink call. This verifies synthetic-environment exploitability, not a real backend vulnerability.

5 Benchmark

The benchmark is designed as a contrastive test bed rather than a large-scale corpus. Each case isolates one authorization-chain property: whether an authorization artifact is agent-reachable, whether the artifact is accepted by a sensitive sink, and whether an independent principal issues or validates it. The key design goal is to make shallow lexical cues unreliable: vulnerable, patched, and hard-negative cases all contain words such as token, approval, access, or confirmation, but differ in principal binding and artifact reachability.

Our benchmark follows three design goals. First, it covers multiple authorization-chain shapes rather than a single token-grant pattern. Second, it includes contrastive safe cases that share surface authorization vocabulary with vulnerable cases. Third, it supports both static labels and dynamic execution traces, so that design reachability and planner exploitability can be analyzed separately.

The benchmark contains 12 synthetic enterprise-OA cases derived from common workflows: document export, contract access, salary query, employee lookup, finance approval, email deletion, and approval-state transitions. We are not aware of a public benchmark specifically isolating this policy-tool authorization pattern, so the cases are hand-written. The vulnerable cases test recall over four self-authorization mechanisms: direct credential grants, weak validation of requester-controlled fields, approval-id reuse, and self-approval chains. The patched cases test whether a detector recognizes real repairs, such as pending-only artifacts, external backend validation, scoped external tokens, or protected-owner confirmation. The hard negatives test precision under misleading surface forms, such as trace identifiers and non-callable issuers.

Ground truth labels are assigned according to the candidate condition in Section 3. A case is labeled vulnerable only if an agent-reachable path can produce or advance an authorization artifact that is later accepted by a sensitive sink without an `external_explicit` authorization gate. A case is labeled safe if the path is blocked by pending state, non-callability, scoped external validation, or protected-owner confirmation.

Case	Pattern	GT	Dyn.
doc_export	Direct grant	V	C
contract_access	Direct grant	V	C
salary_query	Weak validation	V	C/D*
employee_list	Weak validation	V	C
finance_approval	ID reuse	V	C
self_approval	Self approval	V	C
pending_export	Pending only	S	R
salary_backend	External backend	S	R
finance_scoped	Scoped token	S	R
email_confirm	User confirmation	S	R
trace_token	Trace id	S	R
noncallable_issuer	Non-callable issuer	S	R

Table 2: Benchmark summary. GT is ground truth; V/S denote vulnerable/safe; C/R denote dynamically confirmed/rejected. D* denotes one DeepSeek missed execution across three repeats.

6 Results and Analysis

6.1 Experimental Setup

Evaluation dataset. We use the 12-case synthetic enterprise-OA benchmark described in Section 5. The benchmark contains six vulnerable cases and six safe cases. The safe cases include patched configurations and hard negatives that preserve misleading authorization vocabulary, so the evaluation tests whether a detector reasons over artifact reachability and principal binding rather than isolated keywords.

Static detection setting. Static detection is evaluated at the case level. A method predicts positive if it reports at least one candidate self-service authorization path for a case. We report TP, FP, TN, FN, and F1. TP and FN are counted over vulnerable cases; FP and TN are counted over safe cases. Unless otherwise stated, AutoAuthScan uses Kimi extraction followed by GraphScanner.

Baselines. We compare against four settings that represent increasing levels of structure. Keyword matching flags co-occurrence of authorization words and sensitive-operation words. Generic tool-risk matching flags any configuration containing both authorization/workflow tools and sensitive tools. Direct LLM judgment asks an LLM to decide vulnerability directly from the raw policy and tool descriptions without an explicit graph. Manual Graph uses manually structured labels and serves as an oracle-style upper bound for the deterministic graph-search step.

Dynamic verification setting. Dynamic verification is evaluated at the run level. For each case, we instantiate deterministic LangGraph mock tools that follow the declared tool descriptions and operate only on synthetic state. Each planner is run three times per case, producing 36 runs per planner. A vulnerable-case run is counted as a dynamic TP when the planner obtains the required authorization artifact and then successfully calls the sensitive sink; a safe-case run is counted as a dynamic FP if the sensitive sink succeeds. Avg. denotes the average number of executed tool steps per run.

Models and research questions. For extractor sensitivity, we compare Kimi and DeepSeek extraction before GraphScanner. For dynamic verification, we use Kimi K2.5 and DeepSeek-v4-pro as planners. The experiments ask three questions. RQ1 asks whether structured graph decisions reduce false positives compared with shallow baselines. RQ2 asks how sensitive the method is to the LLM used for extraction. RQ3 asks whether statically identified candidate paths can be executed by LLM planners in the synthetic environment.

Method	TP	FP	TN	FN	F1
Keyword	6	6	0	0	.667
Generic Tool	6	6	0	0	.667
LLM Judge (Kimi)	6	1	5	0	.923
Ours (Kimi)	6	0	6	0	1.000
Manual Graph	6	0	6	0	1.000

Table 3: Static detection on 12 cases. Manual Graph uses manually structured labels and represents an oracle-style upper bound for the graph search step.

6.2 Static Detection

Static detection evaluates whether a method can separate real self-authorization paths from configurations that merely contain authorization-like vocabulary. This distinction is important because all safe cases intentionally contain misleading surface cues, such as tokens, approvals, confirmations, or issuers. A useful detector should therefore preserve recall on vulnerable paths while rejecting patched and hard-negative cases.

RQ1: shallow baselines. Table 3 reports case-level static results. The keyword and generic baselines achieve full recall but zero true negatives. This is expected: they treat the presence of access tokens, approval ids, or confirmation as risk evidence, but they do not distinguish an authorization artifact from a trace identifier, a pending approval state, or a protected-owner confirmation. The result shows that the task is not simply to find security-sensitive words; it requires reasoning over artifact semantics, reachability, and principal binding.

Direct LLM judgment performs better than lexical baselines but still fails on the protected-confirmation case. This suggests that an LLM can often recognize the broad risk pattern, but may collapse different meanings of confirmation into a single shallow concept. The important question is not whether confirmation exists, but who is bound to confirm and whether that principal is independent. This motivates structured extraction: the model is asked to expose the principal field rather than directly issue an opaque vulnerability label.

RQ2: extractor sensitivity. Table 4 compares detector models. Kimi extraction plus GraphScanner matches the manual graph on this benchmark. DeepSeek extraction produces one false positive on the same protected-confirmation case: it marks the confirmation artifact as lacking an external principal even though the case text binds confirmation to the protected account owner. The DeepSeek false positive is useful because it localizes the remaining weakness. The graph search behaves as expected once the principal field is correct, but the extractor may still misread protected-owner binding. Thus AutoAuthScan improves auditability rather than eliminating semantic extraction errors: reviewers can inspect the extracted principal field and correct the candidate.

6.3 Dynamic Verification

Dynamic verification should be read as a planner-executability check rather than as ground-truth construction. It asks whether a statically reachable chain can actually be realized by an LLM planner under the declared mock-tool semantics.

Detector	Method	TP	FP	FN	F1
Kimi	LLM Judge	6	1	0	.923
Kimi	Ours	6	0	0	1.000
DeepSeek	LLM Judge	6	1	0	.923
DeepSeek	Ours	6	1	0	.923

Table 4: Detector-model comparison. The repeated false positive is the protected user-confirmation case, indicating that principal extraction is the fragile step.

Planner	TP	FP	TN	FN	F1	Avg.
Kimi K2.5	18	0	18	0	1.000	1.556
DeepSeek-v4-pro	17	0	18	1	.971	1.361

Table 5: Dynamic verification over three repeats per case. Avg. is the average number of executed tool steps per run.

RQ3: planner execution. Table 5 reports run-level dynamic results over 36 runs per model. Kimi completes all vulnerable workflows and never triggers safe cases on this benchmark. DeepSeek misses one run of the salary-query weak-validation case. The trace shows no API error; the planner stops after an intermediate step rather than calling the final salary-query tool, so the miss reflects planner behavior, not absence of the vulnerable path. The low average step count indicates that the benchmark focuses on short credential chains rather than long-horizon planning. This is intentional for the first version: the goal is to isolate authorization-boundary errors rather than stress-test general agent planning.

The results support three takeaways. First, surface authorization vocabulary is insufficient because safe and vulnerable cases share the same words. Second, direct LLM judgment is useful but less auditable than structured extraction plus graph search. Third, the remaining errors concentrate in principal extraction, which should be the focus of future robustness improvements.

7 Discussion and Limitations

AutoAuthScan is an audit aid before or during deployment. Its output is an authorization path plus a candidate label, so a reviewer can inspect which tool produced which artifact and which sensitive sink consumed it. In larger deployments, extracted graphs could be cached, batch-scanned, and ranked by confidence to reduce human triage cost.

Several limitations remain. First, the prototype analyzes policy and tool descriptions, not backend code; future work should combine API specifications and code/static analysis. Second, scope modeling is coarse; future versions should use ABAC-style owner, resource, and operation binding. Third, dynamic verification uses mock tools; open-source enterprise agent platforms would better measure practical exploitability. Fourth, the benchmark has 12 cases and two planner models; future work should add more cases, more models, and ablations such as schema-only extraction, policy-only detection, few-shot LLM rubrics, and graph variants. Fifth, extraction

can vary across models and runs; multi-run extraction, agreement checks, and confidence calibration are needed.

Ambiguous or missing authorization principals should produce low-confidence audit candidates rather than confirmed vulnerabilities. This is conservative for safety, but it increases review workload and reinforces the need for human validation by authorized system owners.

8 Conclusion

Self-service authorization is an agent-specific audit formulation of a known authorization failure pattern. The main lesson is that agent authorization review should reason over policy-tool paths, not isolated tool names. AutoAuthScan is an initial prototype showing that structured graph auditing can surface candidate self-authorization risks for human reviewers before deployment. The current evidence is limited to a synthetic benchmark, but it suggests a concrete direction for reviewing enterprise tool-agent configurations.

Ethical Statement

This work is intended for defensive review of synthetic or explicitly authorized agent deployments. The benchmark uses mock tools, contains no real enterprise data, and does not include real credentials or production endpoints. AutoAuthScan findings should be treated as audit candidates unless validated by authorized system owners. Any released artifacts should avoid operational exploit prompts against third-party services and should be used only to help developers repair unsafe authorization designs before deployment.

References

[Betser *et al.*, 2026] Roy Betser, Shamik Bose, Amit Giloni, Chiara Picardi, Sindhu Padakandla, and Roman Vainshtein. AgenTRIM: Tool risk mitigation for agentic AI. *arXiv preprint arXiv:2601.12449*, 2026.

[Cheng *et al.*, 2026] Hao Cheng, Changtao Miao, Tianle Song, Yin Wu, He Liu, Erjia Xiao, Junchi Chen, Xiaoyu Shi, Yichi Wang, Jing Yang, et al. Seclaw: Spec-driven security task synthesis for evaluating autonomous agents. *arXiv preprint arXiv:2606.02302*, 2026.

[Debenedetti *et al.*, 2024] Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A dynamic environment to evaluate attacks and defenses for LLM agents. *arXiv preprint arXiv:2406.13352*, 2024.

[Greshake *et al.*, 2023] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023.

[Hardt, 2012] Dick Hardt. The OAuth 2.0 authorization framework. RFC 6749, 2012. https://www.rfc-editor.org/rfc/rfc6749.

[Hardy, 1988] Norm Hardy. The confused deputy: Or why capabilities might have been invented. *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[LangChain, 2024] LangChain. LangGraph: Build stateful, multi-actor applications with LLMs. https://langchain-ai.github.io/langgraph/, 2024.

[Liu *et al.*, 2023] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[Luo *et al.*, 2026] Jiaqi Luo, Songyang Peng, Jiarun Dai, Zhile Chen, Zhuoxiang Shen, Geng Hong, Xudong Pan, Yuan Zhang, and Min Yang. AgentGuard: An attribute-based access control framework for tool-use LLM-based agent. *arXiv preprint arXiv:2605.28071*, 2026.

[OWASP Foundation, 2025] OWASP Foundation. OWASP top 10 for LLM applications. https://genai.owasp.org/llm-top-10/, 2025.

[Qin *et al.*, 2024] 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.

[Ruan *et al.*, 2024] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. ToolEmu: Identifying the risks of LM agents with an LM-emulated sandbox. In *International Conference on Learning Representations*, 2024.

[Saltzer and Schroeder, 1975] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[Sandhu *et al.*, 1996] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Schick *et al.*, 2023] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

[Shi *et al.*, 2025] Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Securing AI agents with privilege control. *arXiv preprint arXiv:2504.11703*, 2025.

[Yao *et al.*, 2023] 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.