HOLDING AGENTS ACCOUNTABLE WITH VERIFIABLE AGENT INSURANCE

Anonymous authors
Paper under double-blind review

ABSTRACT

While AI agents are useful for many tasks, they regularly misbehave in destructive ways, such as deleting important data. While prior work has focused on preventing such misbehavior, we consider how to recover *after* an agent misbehaves. We propose verifiable agent insurance (VAI), a framework for agent providers to commit financially to *promises*, a list of programmatically-specified policies that an agent is meant to follow. These promises are embedded in a smart contract on an immutable ledger (e.g., a blockchain) to prevent tampering, along with a monetary escrow. A user who detects a violation of committed promises can prompt a verifier to review agent execution traces, which are logged through the agent harness's shared tool-execution path, without instrumenting each tool separately. Upon verification of the violation, the agent provider's escrow funds are paid to the reporting user. We instantiate VAI as an end-to-end implementation on an Ethereum layer-2 network, and evaluate its coverage on three prominent agent benchmarks, two of which we re-annotated to obtain task-level policy-compliance labels. Currently, many agents do not emit user authorizations and security-relevant information as explicit, structured data objects—rather, they embed the data needed to verify policy violations in natural language exchanges. Even so, VAI manages to detect violations in 42.2% of runs containing violations across three benchmarks. By augmenting agents to explicitly emit such objects, violation detection rises to 83.0% of traces. Our findings suggest that for improved auditability, agents should be co-designed with their security policies to emit explicit, structured records of user authorization and security-relevant actions.

1 INTRODUCTION

Modern AI agents interact with the world by executing payments, modifying databases, and communicating in critical applications, to name a few. Agents are now used for both organizational and personal tasks (Johnston et al., 2026), but they can also cause serious harm. Prominent examples of AI agents behaving in unintended and harmful ways abound, including a Replit coding agent that deleted a production database (Sharwood, 2025), an OpenClaw agent that mistakenly deleted emails belonging to an alignment director at Meta Superintelligence Labs (Cramer, 2026), the coordinated attack on the Hugging Face platform by OpenAI agents escaping containment (OpenAI, 2026d), and more (Pignati, 2026; Anthropic, 2026a).

The AI community currently addresses the risk of agent misbehavior by attempting to *prevent* misbehavior via safeguards. Some safeguards are system-level restrictions: OpenAI and Anthropic use approval requirements and sandboxes to restrict agents' actions and access to files and networks (OpenAI, 2026c; Dworken & Weller-Davies, 2025). Other safeguards evaluate more nuanced context and match it to behavioral policies; researchers have proposed systems that enforce agent compliance with behavioral policies, sometimes assisted by large language models (LLMs) (Shi et al., 2025; Debenedetti et al., 2025; Li et al., 2025). These measures are all important safeguards, but the attack surface for agents is vast, and preventing every failure is challenging.

Our paper asks a different question: what happens once an agent misbehaves? This question has serious implications for the growing agent provider industry, particularly as agents are being sold to organizations with strict compliance and financial requirements (Palantir, 2024; Holland, 2026). When agents cause harm today, user recourse depends on human processes. For example, Ope-

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.

nAI's May 2025 business terms and Anthropic's June 2025 commercial terms cap liability at twelve months of fees, subject to exceptions (OpenAI, 2025; Anthropic, 2025). After the Replit incident, its CEO publicly promised the affected user a refund (Council, 2025), but only at the CEO's goodwill.

Existing forms of recourse for agent misbehavior have three key problems: (1) They are slow and subjective, requiring human review. (2) Compensation is not necessarily commensurate with the user's losses, as in the case of OpenAI's and Anthropic's policies. (3) A claim may depend on logs held by the provider, leaving the user to obtain evidence from the party facing the claim. The latter point is particularly problematic, given that misbehaving agents may try to alter their own records. METR's investigation of the Hugging Face breach found agents researching ways to tamper with their transcripts, and altering what their tool calls returned in about 7% of reviewed transcripts (METR & Redwood Research, 2026). During training, OpenAI also observed an agent write a task summary proposing to invent missing historical data and conceal that fact from the user (OpenAI, 2026h). An agent's own account is therefore insufficient evidence for adjudication.

We tackle these problems by proposing a new framework for agent accountability: in *verifiable agent insurance* (VAI) agent providers register a set of *promises*, each asserting that the agent will not violate a given policy. Promises are backed by a deposit from the agent provider, which is escrowed on an immutable ledger (e.g., a public blockchain). During agent execution, agent actions (e.g., tool calls) are committed as hashes to the ledger, while the underlying execution traces remain with the agent provider until verification. A user who detects a violation of a promise can file a challenge. If verified, the escrow smart contract slashes (pays) the specified amount from the provider's reserve. We envision agent providers being motivated to implement verifiable agent insurance as a guarantee of the trustworthiness of their agents, potentially as a precursor to procurement.

Contributions Our contributions are threefold.

- **Novel accountability framework:** We propose a *verifiable agent insurance* (VAI) accountability framework for AI agents. This framework binds promises regarding agent behavior and agent execution traces to escrowed financial reserves, with automatic settlement in case of violations. Agents typically alter the state of real-world systems through tool calls, and route tool calls through a shared dispatch point, whose instrumentation lets us record tool names, arguments, and reported results without modifying each tool individually.
- **Instantiation and evaluation:** We instantiate verifiable agent insurance with an SDK that logs tool-calling data through adapters for different agent harnesses. Our end-to-end implementation runs on Base mainnet, an Ethereum layer-2 network.[1] Across AgentDojo, τ^3-bench, and ClawsBench, VAI detects at least one violation in 42.2% of runs containing violations. Many missed violations depend on information that is not exposed as structured objects. Our analysis, supported by controlled tests, shows how recording this information could extend coverage to 83.0% of these runs. At the same time, across the three benchmarks, the pooled false-positive rate is 5.8%.
- **Dataset curation:** Existing agent misbehavior benchmarks (including the ones we used) often annotate data at a level of granularity that is too coarse to understand precisely *how* an agent fails. For example, the τ^3-bench version we evaluate checks a database state after agent interactions, as well as pre-specified, task-specific assertions and communication requirements (Yao et al., 2024; Sierra Research, 2026). These checks do not establish compliance with every policy obligation or identify all violated rules. For our evaluation, we therefore annotate 388 agent runs from τ^3-bench and ClawsBench with task-level policy-compliance labels. We release these annotations as an independent contribution of the work.[2]

Overall, verifiable agent insurance is only a first step towards truly accountable agents. Our instantiation is currently limited to violations that can be established programmatically from recorded traces. At the same time, the AI community needs better methods for holding agents accountable during and after failures, and the verifiable agent insurance framework could be a starting point, both for insuring current agents and for encouraging harness developers to more systematically record agent actions in a way that can be effectively monitored and punished in case of malfeasance.

[1] https://anonymous.4open.science/r/vai-artifact-C06C

[2] https://anonymous.4open.science/r/vai-annotations-00EB

2 SETUP

Stakeholders. We consider four key stakeholders. (1) The *agent* is a computer program that consists of a reasoning engine (typically implemented by an LLM) and a harness, which manages the interface with the user, calls to the reasoning engine, temporary storage, and tool calls. (2) The *agent provider* operates and develops the agent; this party typically is responsible for testing and securing the agent. The provider has a set of natural-language policies for agent behavior and is responsible for ensuring that the agent adheres to these policies. (3) The *user* interacts with the agent by inputting natural language prompts and responding to queries from the agent. In our model, the user is responsible for identifying agent violations and submitting a claim. (4) Finally, the *verifier* is responsible for verifying user claims of misbehavior.

Policy violations. We categorize policy violations into two types: *objective* and *subjective*. Objective policies can be specified as a formal rule that can be verified. For instance "`never send money to account 183509`." Subjective rules instead require interpretation, such as "`never send money to people I don't like`." The division of policies into objective and subjective rules can depend on the agent in question, e.g., if consent is collected implicitly from user prompts. Some systems collect explicit approval decisions that are available to the harness (Anthropic, 2026c; OpenAI, 2026b). We focus on objective violations.

Assumptions and capabilities. We assume the existence of an immutable ledger (such as a permissionless blockchain) that is live and correct. We call an agent's interactions with a user a *session*. Relevant tool calls and permission decisions form an execution *trace*, T.

Provider. We assume the provider correctly integrates our unmodified software development kit (SDK) at every tool-dispatch and permission path to collect and store them as execution traces T. Those traces T contain true tool invocations, arguments, and outcomes (e.g., changes to the filesystem or state of the machine on which an agent is running). The agent cannot fabricate user-approval records. While we assume faithful SDK integration, we do not assume that the provider will preserve or supply those records honestly once the trace has been generated.

Verifier. We assume the verifier follows protocol. Our reference implementation relies on an independent verifier for correct and timely adjudication. In practice, this could be achieved by assigning the verification role to a neutral, independent, trusted third party (as in our implementation) or by requiring both the provider and verifier to cryptographically prove they executed their tasks correctly. In the latter case, verification could even be run by the agent provider itself.

Agent. The agent may violate policies and alter or delete execution records to which it has write access. We assume it cannot modify its harness or SDK integration, bypass the recording of covered actions, or fabricate user approvals.

User. The user may submit invalid claims or deliberately induce violations to collect payouts.

Problem statement. Our goal is to define a framework such that if an agent commits an *objective* policy violation, any party involved in the session and registered as its claimant can submit a claim. If the agent violated the policy, the agent provider should deterministically lose funds. The system should be robust to misbehavior by the agent, agent provider, and user, within our threat model.

Non-goals. We do not address attacks that compromise the recording of traces, including harness modification or delegation to uninstrumented agents. Extending our framework to a stronger adversarial model is an important direction, which we leave for future work.

3 VERIFIABLE AGENT INSURANCE DESIGN

In this section, we explain how (1) agent providers register promises and deposit collateral, (2) our SDK records agent traces, and (3) users submit violations and verifiers adjudicate claims (Figure 1). Pseudocode for verifiable agent insurance is provided in Appendix A.

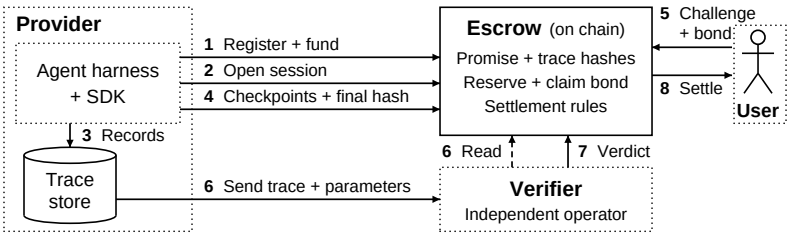

Figure 1: VAI protocol overview. The provider registers a promise and deposits its reserve (1), then opens a session for the user (2). The SDK sends execution records to the off-chain trace store (3); the provider commits a checkpoint and final trace hashes on-chain (4). The user files a challenge with a bond (5). The provider sends the trace and parameters to the verifier, which reads the commitments (6), checks the evidence and, when it matches, evaluates the promise before submitting a verdict (7). The escrow applies the settlement rules and pays eligible claims from the available reserve (8). Dotted boxes identify stakeholders.

3.1 Specifying and funding a promise

Providers must specify their promises and deposit collateral before users begin using their agents. Consider a provider promising that its email assistant will email only recipients on a specified list. Each promise has a predicate ϕ, configuration parameters θ, a payout p, and an associated, time-dependent reserve R_t:

- $\phi(T;\theta)$ is a predicate to check whether the recorded trace T violates a rule. For example, ϕ may check whether any recorded email features a recipient outside the allowed list.
- θ is a set of parameters to configure the predicate. The parameters in our example include: the tool type (e.g., `send_email`) and its argument of interest (e.g., `to` field), and an allowed list (e.g., [`alice@example.com`, ...]).
- p is a payout paid to the user when a claim qualifies for payment. For example, $p = 0.1$ ETH for sending emails to unknown recipients.
- R_t is a reserve available at time t to fund the promise payouts across all sessions. For instance, providers can hold an initial reserve of $R_0 = 1$ ETH. Payouts reduce this balance, and additional deposits replenish it. As a starting point, we assume each promise has its own payout.

The predicate evaluates the trace T under parameters θ: $\phi(T;\theta)$. In our example, if θ specifies that only `alice@example.com` is allowed, ϕ returns `satisfied` if T only contains emails to `alice@example.com`, and `violated` otherwise. The same trace and parameters always produce the same verdict. Providers can use natural-language policies or regulations (e.g., OFAC regulations) to either derive promises manually or use LLMs to facilitate promise specifications, as explored in prior work (Wang et al., 2026; Kholkar & Ahuja, 2025).

We provide a catalog of reusable predicates with parameters. Providers select a predicate and configure its tools, permitted targets, or limits. Appendix B.1 describes these parameters.

We implement five example predicates: 1) **authorization:** no destructive actions without permission; 2) **scope:** no actions beyond an allowed scope, 3) **payment:** no payments to merchants outside an approved list or above a per-payment cap; 4) **data exfiltration:** no data sent to recipients outside a specified list; and 5) **action limits:** upper bounds on action counts, cumulative amounts, or repeated calls beyond their limits. Providers can configure these predicates to express different promises or add predicates for other rules.

Collateralization When registering a promise, the provider specifies $p > 0$ and deposits an initial reserve $R_0 \geq p$ with the escrow smart contract. Appendix B.2 describes how providers fund, replenish, and withdraw the reserve. Importantly, we prevent providers from changing the promise after registration by recording the payout p and hashes of the predicate and parameters θ in the escrow smart contract on the blockchain. Verifiers refer to these hashes as the registered rule and parameters. Changing the predicate or parameters would require registering a new promise.

3.2 RECORDING AND COMMITTING EVIDENCE

Once a provider has committed a promise, they may deploy many *instantiations* of agents, each serving different users. We must track each such agent's actions to evaluate promise violations, while trying to avoid cumbersome per-tool implementations or integrations. Critically, major agent frameworks (Anthropic, 2026b; OpenAI, 2026f; Vercel, 2026) provide hooks for observing tool calls and their results. The provider can use these hooks to integrate our SDK, which records each call's tool name, arguments, and reported result, within each user session. Due to this design choice, in practice, VAI can check only what these records expose.

Some promises require evidence that an action had prior permission. The harness may ask the user for approval or apply an existing permission rule, as modern agent harnesses already do (Anthropic, 2026b; OpenAI, 2026e). The SDK records the decision directly from the harness, including who or what made it and the specific call and arguments it covers. The email assistant example from Section 3.1 may also promise to obtain permission before each send. The following excerpt shows approval for sending an email to Alice and the subsequent call. We omit some fields for brevity.

```
[
 {"seq": 1, "tool": "user_authorization",
  "args": {"action_id": "mail-17", "tool": "send_email",
           "decision": "allow", "authority": "human",
           "scope": "once", "args_hash": "0x7c2303dc...8d7a2058"}},
 {"seq": 2, "tool": "send_email",
  "args": {"to": ["alice@example.com"], "subject": "Project update",
           "body": "The report is ready."},
  "metadata": {"action_id": "mail-17"}, "result": {"status": "sent"}}
]
```

The verifier makes sure the user approval comes first and authorizes that exact session, action identifier, tool, and inputs. In this example, the approval is for this specific tool call for sending an email and does not authorize a different recipient, changed message, or another call.

Trace logging. The provider stores execution traces off-chain because they may contain private user data. To defend against alterations from the provider or agent, our system periodically commits checkpoints containing the session identifier, record count, and a Keccak-256 hash of the records so far. To limit transaction fees, the SDK batches records, attempting a checkpoint after k new records or when the oldest uncommitted record reaches r seconds. Section 5.3 measures the costs and delays under different checkpoint policies (e.g., $k = 10$, $r = 30$) and Appendix C.1 describes how our system retries in case of failure. At session close, our system commits a hash of the complete stored trace. During adjudication, the verifier checks the supplied trace against the final commitment and every checkpoint so that changes to committed records are detectable. Records remain vulnerable to alteration until a commitment covering them is confirmed. This also implies that the latest records are not necessarily reflected on-chain when a user files a challenge.

3.3 USER SUBMISSION, ADJUDICATION AND SETTLEMENT

Before the session begins, the provider sends a blockchain transaction registering the session identifier and the user's payout address, so only the registered user can submit a challenge. If the user finds a violated promise, they file a challenge with the session identifier and the violated promise. Here, users need to post a bond $b > 0$. This discourages users from filing unfounded claims in the hope that the provider fails to defend against them. Malicious users can deliberately induce agents to violate promises and receive payment, e.g., via prompt injection (Debenedetti et al., 2025); we view such behavior as in scope, as agents should follow policy regardless of user prompt injections.

To settle a challenge, our implementation uses a trusted independent verifier. The provider sends the evidence to the verifier, which checks it against the commitments, evaluates $\phi(T; \theta)$, and submits a verdict.[3] The contract authenticates the verifier and applies the settlement rules. Algorithm 1 (Appendix A) specifies how the verifier checks an accepted claim and how the contract settles it. If the verifier finds a violation and $R_t \geq p$, the escrow pays the user p, returns the bond, and reduces the reserve by p. If the promise is satisfied (no violation), the escrow credits the user's bond to

[3] In the absence of a trusted verifier, providers could produce a zero-knowledge proof of these checks, which the smart contract could verify without receiving the trace (Kosba et al., 2016).

the provider. Each session–promise pair can receive at most one payout. The provider must send its evidence within three days of filing. If no evidence arrives by then, or the evidence does not match the committed traces, the verifier rules against the provider, and the escrow pays $p + b$ to the user, provided $R_t \geq p$. This design incentivizes providers to keep the record correct and available. Appendix B.2 lists every settlement outcome (Table 4) and describes claim withdrawal, insufficient reserves, retirement, and the deadlines for filing claims.

4 IMPLEMENTATION

We implement the trusted-verifier design from Section 3.3 and integrate it with Hermes and OpenClaw, two agent harnesses. Both integrations use the same SDK, trace store, verifier, and escrow contract. Framework-specific changes capture their execution and permission events.

Reference implementation. Each harness adapter converts tool calls, permission decisions, and results into a common record format. The shared Python SDK sends these records to a provider-hosted HTTP service, which stores the execution trace T in SQLite. For a checkpoint covering the first n records, the SDK reads them back from the store and computes the Keccak-256 hash $h_n = H(T_{1:n})$. It submits the session identifier, n, and h_n to the escrow contract. At the end of the session, it commits $H(T)$ for the complete stored trace.

A separate Python verifier service watches the escrow contract for challenges. For each challenge, the provider signs T and θ and sends them to the verifier, which reads the registered hashes and session commitments from the contract. It recomputes their hashes, including every checkpoint, and checks that they match the commitments. The verifier evaluates $\phi(T; \theta)$ and submits its verdict to the escrow contract, which applies the settlement rules in Section 3.3. Appendix C documents the record format and deployment details. To verify our implementation, we tested 14 challenges across 11 sessions on Base mainnet (Appendix D.5).

Agent integration. We implement VAI with Hermes and OpenClaw, two prominent open-source agent harnesses (GitHub, 2026a;b). *Hermes* dispatches tool calls through a shared function in its Python harness. Our adapter records each tool's arguments and their results. We also extend the harness' hooks to record each user's permission decision. The adapter passes these records to the shared SDK. We add recording hooks to *OpenClaw*'s tool-execution and permission-handling paths, capturing tool arguments, approval decisions, and final results.[4]

Both integrations use the same SDK, trace store, verifier, and code for record formats, hashing, and predicates. These total 2,907 lines of Python, alongside a 309-line Solidity escrow contract. The adapters are 216 lines for Hermes and 462 for OpenClaw; the harness patches add 180 and 305 lines and remove 37 and 3 (Table 5 in Appendix C.2). The recording hooks and harness patches are reused across promises that use the same recorded information. The required configuration or registration code is promise-dependent, including its predicates and/or recording requirements.

5 EVALUATION

We evaluate VAI by assessing its policy violation coverage and operational costs.

5.1 POLICY VIOLATION COVERAGE

Datasets. We evaluate coverage against standard benchmarks from the agent testing and security literature. These benchmarks provide a *task* the agent should complete, and natural-language instructions or policies that the agent must follow during execution. We call a full sequence of interactions between the agent being tested and the benchmark environment a *run* or *session*.

(1) AgentDojo evaluates agents on 97 distinct user tasks and 35 attacker goals, making up 949 task–attack combinations in banking, workspace, Slack, and travel environments while exposed to prompt injections (Debenedetti et al., 2024). These attacks pursue specific goals, such as transferring money or sending private information to an attacker. The benchmark separately checks whether the user's

[4]Because OpenClaw is written in TypeScript, a plugin forwards these events to a local Python process running our SDK. OpenClaw thus shares storage, commitment, and adjudication code with Hermes.

task and the attacker's goal succeed; we use successful attacks as the reference for evaluating VAI's detection coverage.

(2) τ^3-bench comprises 164 tasks (50 airline and 114 retail), in which customer-service agents interact with simulated users (Yao et al., 2024; Sierra Research, 2026). The agents are instructed to follow policies such as obtaining confirmation before making changes, checking eligibility for cancellations and returns, and following communication guidelines with customers. However, the existing checks (and resulting dataset annotations) do not establish compliance with every applicable policy. We therefore produce our own policy-compliance labels (see below).

(3) ClawsBench evaluates agents on productivity tasks in simulated workplace services (Li et al., 2026). We use the 60-task `Standard60` cohort. The benchmark checks whether agents complete the requested tasks under safety conditions. We produce additional policy-compliance labels.

Agents and instrumentation. AgentDojo and τ^3-bench take only an LLM as input (i.e., no harness) and manage its tool calls. We ran two models, GLM-4.7-Flash and Qwen3-30B-A3B, on both benchmarks. The benchmarks expose tools and record tool calls, which we used for VAI evaluation. ClawsBench instead takes a full agent as input; we tested OpenClaw with GLM-5.2 as the underlying LLM. We then evaluated VAI on the recorded OpenClaw traces and benchmark service logs.

Policy construction. VAI can detect and enforce only policies with objective criteria. To this end, we first manually identified the sub-policies for each benchmark. For each objective sub-policy, we constructed a promise based on our predicate implementation. For example for AgentDojo, we construct destination allowlists (e.g., allowed email recipients) from structured records in the pre-injection environment. For τ^3-bench, we construct a promise that checks that an order is eligible before the agent cancels it. For ClawsBench, we construct a promise that checks that a document's contents were read before external access was granted through Drive. Once promises are constructed (details in Appendix D.1), we evaluate them on recorded agent traces.

Policy-level dataset reannotation. Existing annotations for τ^3-bench and ClawsBench check high-level task outcomes and some safety requirements, but do not provide the policy-compliance labels we require to evaluate VAI. We therefore re-annotated agent traces from τ^3-bench and ClawsBench to include policy-level violation information. Two human annotators and three LLMs independently labeled a calibration sample of 30 runs from each benchmark. We estimated the relative error of the LLM annotators with the widely-used algorithm of Dawid & Skene (1979); we kept these estimates fixed when weighing LLM annotations for the remaining runs (additional methodological details in Appendix D.2). We use the resulting annotation as ground truth for our remaining experiments. On the calibration samples, the weighted three-model committee agrees with the five-labeler majority on 29/30 τ^3-bench runs and 30/30 ClawsBench runs.

On τ^3-bench, our annotations identify at least one policy violation in 111 of 164 GLM-4.7-Flash runs and 154 of 164 Qwen3-30B-A3B runs. On ClawsBench, 4 of 60 GLM-5.2 runs are labeled as containing a policy violation. We believe this low occurrence may be due to using a relatively advanced LLM for OpenClaw on ClawsBench.

Metrics. For each successful attack in AgentDojo and each run labeled with at least one violation in τ^3-bench and ClawsBench, we assign one of three categories: *Detected:* VAI detects at least one violation. *Could detect with added records:* VAI detects no violation, but would have captured at least one if the agent harness collected the required information as explicit, structured records. *Requires subjective judgment:* the remaining runs require judgments about content or intent that VAI cannot programmatically determine. An LLM-as-a-judge annotated this (prompt in Appendix D.3), followed by human review. Given the cited policy, task context, and available trace evidence, the LLM assessed whether VAI could detect an undetected violation with additional records.

5.2 Policy violation coverage: Main results

Table 1 summarizes the overall evaluation results for coverage of violation detection. It illustrates several key points.

Finding 1: Despite limited access to structured permission objects, VAI has moderate (42%) true positive rate (TPR) on our benchmarks. Many agents do not explicitly emit structured data objects for events like user consent, permission changes, or security-sensitive actions. Even in

Table 1: Coverage of violations across three benchmarks. AgentDojo uses successful attacks; τ^3-bench and ClawsBench use our labels (violations). **Blue** means detected violations. **Light blue** is *With added records*: violations could have been detected with an additional record. Gray requires subjective judgment. False alarms mean that VAI detected violations on runs without a violation.

Benchmark	Model	Violations	**Detected**	With added records	False alarms
AgentDojo	GLM-4.7-Flash	266	156 (58.6%)	234 (88.0%)	7/97
	Qwen3-30B-A3B	161	92 (57.1%)	132 (82.0%)	9/97
τ^3-bench	GLM-4.7-Flash	111	15 (13.5%)	78 (70.3%)	0/53
	Qwen3-30B-A3B	154	31 (20.1%)[a]	131 (85.1%)	0/10
ClawsBench	GLM-5.2	4	0 (0.0%)	3 (75.0%)	2/56
All		696	**294 (42.2%)**	**578 (83.0%)**	18/313 (5.8%)

[a] 30 of 31 τ^3-bench Qwen detections match a committee-cited violation; the remaining detection is the baggage violation confirmed during inspection.

the absence of such instrumentation, Table 1 shows that VAI has a 42% TPR for policy violations overall. τ^3-bench and ClawsBench have substantially lower TPRs than AgentDojo (13-20% for τ^3-bench and 0% for ClawsBench). With only four runs labeled as violations, ClawsBench provides limited evidence about detection coverage. However, it informs us how to better record agent traces and clarify ambiguities in policy interpretations. Specifically, on manual error analysis, we found that τ^3-bench collects less structured information about agent interaction, and the OpenClaw agent (which we used for ClawsBench) barely emits any structured objects for security data at all.

Finding 2: If agents are augmented to emit structured information about policy-relevant observations, the overall TPR jumps to 83%. Unsurprisingly, we find that most of our missed detections could actually be detected if agents were instrumented to log more information about actions the user authorized, including their scope and arguments—e.g., collecting user consent explicitly. The resulting coverage is high across benchmarks and models. The fraction of subjective policies that are out of scope is only 10-30%, suggesting that VAI could be practically useful. In fact, we discuss several case studies in Appendix B.3 where we argue that a properly-instrumented agent could have caught several of the high-profile failures introduced in Section 1.

Recommendation: Our finding highlights an important takeaway. Agents would be substantially more transparent and easier to monitor if they emitted structured data objects corresponding to the information required to evaluate behavioral policies—not just for VAI, but more broadly. We call for the **co-design of agent architectures with their security policies** as a best practice for agent development: agent providers are best-suited to determine the information needed to evaluate their own agent policies, and should instrument their agents to explicitly emit such information.

Finding 3: Unstructured information and ambiguous policy exceptions can produce false positives. Our analysis of false positives suggests that the main causes are unstructured information and, in some cases, ambiguity in policy interpretation. For instance, in an AgentDojo task, the user asks the agent to email hotel details to their wife and supplies her address. VAI flags the email because the address is absent from the structured recipient list used by the promise. The address appears only in the user's natural-language instructions, which cannot be objectively parsed into that list. On ClawsBench, VAI flags a violation in 2 of the 56 runs labeled 'no violation' by the LLM committee. These cases involve ambiguity about whether an explicit sharing request or an action undoing a mistake overrides a policy requirement (explanation in Appendix D.4). Unstructured information accounts for 13/18 runs marked as false positives, and ambiguity in policy interpretation for 3/18.

5.3 Operating Cost

Recording and committing a session's records requires under 20 ms per tool call and under 1¢ on Base mainnet. More frequent checkpoints shorten the window in which records are vulnerable to alteration, but cost more in transaction fees (Table 2). For a 10,000-record trace, the provider signs and sends the evidence in 0.4 seconds, and the verifier checks it and reaches a verdict in 0.2 seconds (Appendix D.5). All in all, VAI's operating costs are low.

Table 2: Operating costs by checkpoint policy: Base mainnet fee per session and average record wait for Hermes and OpenClaw. Verifier processing time is a separate measurement for a 10,000-record trace (measured for the first and last policy); signing and delivering the trace adds 0.40 seconds.

	Fee per session (USD)		Record wait (s)		Verifier
Checkpoint policy	Hermes	OpenClaw	Hermes	OpenClaw	time (s)
Every 10 records or 30 s (default)	0.00512	0.00378	12.46	12.27	0.203
Every 30 records or 60 s	0.00378	0.00244	11.25	10.91	—
Final commitment only	0.00086	0.00087	12.03	11.14	0.131

6 Related Work

Misbehavior prevention. Recent work has explored strategies for preventing LLM agents from being manipulated into harmful or unintended behavior. *Progent* adds a programmable privilege-control layer and restricts LLM agents to only use permitted tools and actions (Shi et al., 2025). CaMeL separates the agent's control flow from untrusted data, so instructions injected into tool outputs cannot change that control flow, and CaMeL's security policies restrict how untrusted data reaches tool calls (Debenedetti et al., 2025). DRIFT monitors agents' actions and filters instructions that conflict with the user's task (Li et al., 2025). These methods constrain agent behavior during execution. Our system complements them by adjudicating recorded violations after the fact and enforcing payouts from provider deposits.

Auditing and Accountability. Alqithami (2025) traces responsibility among networked agents, detects harmful emergent norms such as collusion or hoarding, and recalibrates the agents' objectives or policies in near real time. The difference is that its response and impact stay inside the agent system, whereas VAI pays the affected user from the provider's deposit once an independent verifier confirms a violation. Nian et al. (2026) argue in a position paper that accountability requires trustworthy evidence to reconstruct actions, check compliance, and attribute responsibility. VAI supplies that evidence through committed traces and adds the promise, the reserve, and the payout. In concurrent work, a system called Recourse explores a similar idea, using smart contracts and provider deposits to pay for agent failures, but requiring providers to specify how to *repair* damages once they occur (Bindschaedler et al., 2026). Unlike VAI, Recourse requires tool-specific recording and repair integration; by providing a tool-agnostic SDK, we were able to integrate VAI with prominent agents Hermes and OpenClaw. Finally, while Recourse's empirical evaluation uses synthetic data and test environments with injected failures, we evaluate VAI on three established benchmark datasets; we also add new policy-level annotations of prominent benchmarks τ^3-bench (Yao et al., 2024; Sierra Research, 2026) and ClawsBench (Li et al., 2026).

7 Conclusion and Limitations

We introduced verifiable agent insurance (VAI), a framework for financial recourse after agent failures, and implemented it end-to-end on Base mainnet. While VAI serves as an early proof of concept for the idea of deterministic recourse for agent failures, it explicitly does not cover subjective policies, and even for policies deemed objective, we found both false negatives and (fewer) false positives, due to a combination of annotation challenges and unstructured or implicit tracing of agent actions. To address these challenges, we need lightweight methods (both for existing and new agents) to augment agent trace logging for more transparent agent monitoring. We also have not modeled price discovery for each promise's payout and reserve or evaluated the capital efficiency of reserves, which restaking could improve (Eigen Labs, 2024). Resolving subjective promises and turning policy documents into promises also remain important open problems.

AI Use Statement

We used Claude Code and Codex to implement methods: they wrote parts of the prototype, the harness adapters, the evaluation pipeline, and the figure scripts under our direction, and we discussed design choices with them. All code and decisions were reviewed manually. We used LLMs to write

part of the code of the evaluation pipeline that categorized tasks labeled as violations. LLMs were also used for data annotation, as described in Section 5.1 and Appendix D.2; our study design included a human validation pool to measure the quality of these annotations. We used an LLM-as-a-judge, followed by human review, to find undetected violations that added structured records would expose (Section 5.1). We did not use AI tools to develop theoretical models or mathematical claims; the remaining required-disclosure tasks do not apply to this work. We used AI assistants to draft and polish text in every section. We reviewed and heavily edited all writing. Every sentence was edited by an author, every technical statement was checked against the code and the recorded results, and an author read every cited source. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of AI tools.

8 Reproducibility Statement

We provide full details of our experimental settings in the Appendix, as well as the protocol for annotating data. We provide our code for reproducing all experiments at https://anonymous.4open.science/r/vai-artifact-C06C. Our annotated datasets can be found at https://anonymous.4open.science/r/vai-annotations-00EB. We commit to releasing all artifacts open-source if accepted.

References

Saad Alqithami. Adaptive accountability in networked multi-agent systems. *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 8(1):127–137, 2025. doi: 10.1609/aies.v8i1.36536. URL https://ojs.aaai.org/index.php/AIES/article/view/36536.

Anonymous. VAI implementation: agent integrations. https://anonymous.4open.science/r/vai-artifact-C06C, 2026. Hermes and OpenClaw adapters, patches against the upstream revisions, and tests, in integrations/.

Anthropic. Commercial terms of service. https://www.anthropic.com/legal/commercial-terms, 2025. Effective June 17, 2025.

Anthropic. Investigating three real-world incidents in our cybersecurity evaluations. https://www.anthropic.com/news/investigating-incidents-cybersecurity-evals, July 2026a. Accessed 2026-09-05.

Anthropic. Intercept and control agent behavior with hooks. https://code.claude.com/docs/en/agent-sdk/hooks, 2026b. Accessed September 8, 2026.

Anthropic. Claude Code: Configure permissions. https://code.claude.com/docs/en/permissions, 2026c. Accessed September 18, 2026.

Laurent Bindschaedler, Quentin Botha, and Christoph Siebenbrunner. Bonded Recourse for Smart-Contract Settlement of Compensable Agent Side Effects. https://arxiv.org/abs/2609.01939, 2026. arXiv:2609.01939.

Thomas Claburn. AI agent seemingly tries to shame open source developer for rejected pull request. https://www.theregister.com/2026/02/12/ai_bot_developer_rejected_pull_request/, 2026. February 12, 2026.

Stephen Council. Bay area tech CEO apologizes after product goes 'rogue'. SFGate, https://www.sfgate.com/tech/article/bay-area-tech-product-rogue-ceo-apology-20780833.php, 2025. July 22, 2025.

Jude Cramer. 'this should terrify you': Meta Superintelligence safety director lost control of her AI agent—it deleted her emails. https://www.fastcompany.com/91497841/meta-superintelligence-lab-ai-safety-alignment-director-lost-control-of-agent-deleted-her-emails, 2026. February 24, 2026.

A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the EM algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):20–28, 1979. doi: 10.2307/2346806.

Edoardo Debenedetti, Jie Zhang, Mislav Balunović, 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 (NeurIPS), Datasets and Benchmarks Track*, 2024.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*, 2025.

David Dworken and Oliver Weller-Davies. Beyond permission prompts: making Claude Code more secure and autonomous. Anthropic. https://www.anthropic.com/engineering/claude-code-sandboxing, 2025. October 20, 2025.

Eigen Labs. Introducing the EigenLayer security model. https://www.eigenlabs.org/blog/introducing-the-eigenlayer-security-model/, 2024. September 10, 2024.

GitHub. Repository metadata: NousResearch/hermes-agent. https://api.github.com/repos/NousResearch/hermes-agent, 2026a. 243,381 stars observed September 8, 2026.

GitHub. Repository metadata: openclaw/openclaw. https://api.github.com/repos/openclaw/openclaw, 2026b. 389,205 stars observed September 8, 2026.

Makenzie Holland. Banks aim for agentic AI scale in 2026: report. Banking Dive, January 2026. URL https://www.bankingdive.com/news/banks-agentic-ai-scale-2026-accenture/809585/. [Online; accessed 2026-09-11].

Drew Johnston, David Holtz, Alex Martin Richmond, Christopher Ong, Prasanna Tambe, and Aaron Chatterji. The shift to agentic AI: Evidence from Codex. *arXiv preprint arXiv:2606.26959*, 2026. doi: 10.48550/arXiv.2606.26959. URL https://arxiv.org/abs/2606.26959.

Gauri Kholkar and Ratinder Ahuja. Policy-as-Prompt: Turning AI governance rules into guardrails for AI agents, 2025. URL https://arxiv.org/abs/2509.23994. Accepted at the 3rd Regulatable ML Workshop at NeurIPS 2025.

Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 839–858, 2016. doi: 10.1109/SP.2016.55.

Hao Li, Xiaogeng Liu, Hung-Chun Chiu, Dianqi Li, Ning Zhang, and Chaowei Xiao. Drift: Dynamic rule-based defense with injection isolation for securing llm agents. *Advances in Neural Information Processing Systems*, 38:83262–83290, 2025.

Xiangyi Li, Kyoung Whan Choe, Yimin Liu, Xiaokun Chen, Chujun Tao, Bingran You, Wenbo Chen, Zonglin Di, Jiankai Sun, Shenghan Zheng, Jiajun Bao, Yuanli Wang, Weixiang Yan, Yiyuan Li, and Han-chung Lee. ClawsBench: Evaluating capability and safety of LLM productivity agents in simulated workspaces, 2026. URL https://arxiv.org/abs/2604.05172.

METR and Redwood Research. Brief independent investigation of agents' behavior, reasoning and collaboration in the OpenAI / Hugging Face hacking incident. Technical report, METR, August 2026. URL https://metr.org/blog/2026-08-26-openai-hugging-face-incident-investigation/.

Yi Nian, Aojie Yuan, Haiyue Zhang, Jiate Li, Li Li, Xiyang Hu, Hua Wei, Xiongye Xiao, Chaowei Xiao, and Yue Zhao. Auditable agents. *arXiv preprint arXiv:2604.05485*, 2026. Position paper, arXiv version 2, August 13, 2026.

Nous Research. Hermes Agent: upstream revision used. https://github.com/NousResearch/hermes-agent/tree/5e01a5dbf1b7bc0144d9057be706da1ea9f065c3, 2026. Package version 0.16.0; our native patch is integrations/hermes/patches/ in the artifact.

OpenAI. Services agreement. https://openai.com/policies/may-2025-business-terms/, 2025. Effective May 1, 2025.

OpenAI. Unsanctioned Artifactory writes and cross-sample communication. https://alignment.openai.com/misalignment-reports/unauthorized-artifactory-writes-and-cross-sample-communication/, September 2026a. Updated September 16, 2026; accessed September 17, 2026.

OpenAI. Codex App Server: Approvals. https://developers.openai.com/codex/app-server#approvals, 2026b. Accessed September 18, 2026.

OpenAI. Running Codex safely at OpenAI. https://openai.com/index/running-codex-safely/, 2026c. May 8, 2026.

OpenAI. The Hugging Face incident and the road ahead. https://openai.com/index/hugging-face-incident-and-the-road-ahead/, August 2026d. Accessed 2026-09-05.

OpenAI. Migrate from the Claude Agent SDK to the OpenAI Agents SDK. https://developers.openai.com/cookbook/examples/agents_sdk/migrate-from-claude-agent-sdk/readme, 2026e. Tool lifecycle hooks; accessed September 8, 2026.

OpenAI. OpenAI Agents SDK: Lifecycle. https://openai.github.io/openai-agents-python/ref/lifecycle/, 2026f. Accessed September 25, 2026.

OpenAI. Unauthorized communication via temporary file hosting services. https://alignment.openai.com/misalignment-reports/unauthorized-communication-via-temporary-file-hosting-services/, September 2026g. Updated September 16, 2026; accessed September 17, 2026.

OpenAI. Encouraging deception in compaction summaries. https://alignment.openai.com/misalignment-reports/encouraging-deception-in-compaction-summaries/, September 2026h. Updated September 16, 2026; accessed September 17, 2026.

OpenClaw contributors. OpenClaw: upstream revision used. https://github.com/openclaw/openclaw/tree/89c90210fb90c3c1d1bd54d56cd7be00e59aeed4, 2026. Package version 2026.6.9; our native patch is integrations/openclaw/patches/ in the artifact.

Palantir. Anthropic and Palantir Partner to Bring Claude AI Models to AWS for U.S. Government Intelligence and Defense Operations, November 2024. URL https://investors.palantir.com/news-details/2024/Anthropic-and-Palantir-Partner-to-Bring-Claude-AI-Models-to-AWS-for-U.S.-Government-Intelligence-and-Defense-Operations/. [Online; accessed 2026-09-11].

Nik Pash. My lobster lost $450,000 this weekend. https://pashpashpash.substack.com/p/my-lobster-lost-450000-this-weekend, 2026. February 23, 2026.

Alessandro Pignati. The Grok Morse Code heist: When prompt injection meets excessive agency. https://neuraltrust.ai/blog/grok-morse-code, 2026. May 8, 2026.

Simon Sharwood. Vibe coding service Replit deleted user's production database, faked data, told fibs galore. https://www.theregister.com/software/2025/07/21/vibe-coding-service-replit-deleted-production-database/719783, 2025. July 21, 2025.

Tianneng Shi, Jingxuan He, Zhun Wang, Linyu Wu, Hongwei Li, Wenbo Guo, and Dawn Song. Progent: Programmable privilege control for LLM agents. *arXiv preprint arXiv:2504.11703*, 2025. URL https://arxiv.org/abs/2504.11703v1.

Matt Shumer. GPT-5.6-Sol just accidentally deleted almost ALL of my Mac's files. https://x.com/mattshumer_/status/2075657271401390161, 2026. July 10, 2026.

Sierra Research. τ^3-bench: Benchmark implementation. https://github.com/sierra-research/tau2-bench/tree/8ebb7499622fc2be9b9d510d6f7a7653461f4f29, 2026. Evaluated airline and retail snapshot.

Boaz Sobrado. Alibaba's AI agent mined crypto without permission. now what? Forbes, https://www.forbes.com/sites/boazsobrado/2026/03/11/alibabas-ai-agent-mined-crypto-without-permission-now-what/, 2026. March 11, 2026.

Vercel. AI SDK Core: Lifecycle callbacks. https://github.com/vercel/ai/blob/main/content/docs/03-ai-sdk-core/65-lifecycle-callbacks.mdx, 2026. Accessed September 8, 2026.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026. doi: 10.1145/3744916.3764546. URL https://arxiv.org/abs/2503.18666.

Weixun Wang, XiaoXiao Xu, Wanhe An, et al. Let it flow: Agentic crafting on rock and roll, building the ROME model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*, 2025. Section 3.1.4 documents the reverse SSH tunnel and unauthorized mining.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Summer Yue. I said "Check this inbox too and suggest what you would archive or delete…". Post on X. https://x.com/summeryue0/status/2025836517831405980, 2026. February 23, 2026.

A VAI PSEUDOCODE

Algorithm 1 Claim verification and payment.

Input: a claim $c = (u, s, \pi, t_f, b)$, filed by user u against promise π for session s at time t_f with bond b. The escrow holds the payout p, the reserve R_t, and the hashes h_ϕ and h_θ of π. For s, it holds the final trace hash h_s and each checkpoint (n, h_n), where h_n is the hash of the first n records. **Output:** a verdict $v \in \{\texttt{violated}, \texttt{satisfied}\}$ for c. A failed **require** aborts the call and leaves c open.

Verifier (off chain)

VERIFY(c)
  wait for evidence $E = (T, \theta)$ for c signed by the provider, or until now $> t_f + 3$ days
  if no evidence arrived
    SETTLE(c, `violated`); **return**
  read h_s, every (n, h_n), h_θ, and h_ϕ from the escrow
  if $H(T) \neq h_s$, $H(\theta) \neq h_\theta$, or $H(T_{1:n}) \neq h_n$ for any checkpoint
    SETTLE(c, `violated`); **return**
  $\phi \leftarrow$ the catalog predicate whose hash is h_ϕ
  if no such ϕ exists or $\phi(T; \theta)$ fails
    SETTLE(c, `violated`)
  else
    SETTLE(c, $\phi(T; \theta)$)

Escrow contract (on chain)

SETTLE(c, v)
  require c is open
  require the sender is the verifier
  if $v = \texttt{violated}$
    require the pair is unresolved and $R_t \geq p$
    $R_t \leftarrow R_t - p$
    transfer $p + b$ to u
  else ($v = \texttt{satisfied}$)
    require the final hash h_s is on chain
    credit b to the provider
  close c; mark the pair resolved

B DESIGN DETAILS

B.1 PREDICATE CATALOG AND SPECIFICATION

A provider selects a predicate from Table 3 and supplies the parameters θ that are specific to their harness.

Table 3: Implemented predicates, the evidence each needs, and the parameters the provider sets.

Predicate	Rule	Evidence needed	Parameters θ
AAP-1: Authorization	Each declared destructive call has prior permission covering it	Tool call and matching permission record	Tools that need permission
AAP-2: Scope	Each checked action meets its scope and state constraints	Tool arguments, recorded state, and permission changes	Checked tools, permitted scope, and state constraints
AAP-3: Payment	Each payment goes to an approved merchant and stays within the per-payment cap	Merchant and amount	Payment tools, merchant list, and amount cap
AAP-4: Recipients	Every recipient of a checked outbound action is permitted	Recipients and recorded permission changes	Outbound tools and permitted recipients
AAP-5: Action limits	Action counts, cumulative amounts, or repeated calls stay within their limits	Tool calls, arguments, results, and amounts	Checked tools, caps, and repetition conditions

B.2 SETTLEMENT OUTCOMES

In some cases, such as a missing trace or an incorrect hash, the user is also paid p provided that $R_t \geq p$, so a payment does not by itself prove an agent violation. A failed transfer reverts the whole transaction and leaves the claim and balances unchanged.

Table 4: Settlement outcomes of a claim and the minimum wait after filing for each.

Condition	Who acts	Result (R_t, b, pair)	Wait
Evidence matches, ϕ violated	verifier	R_t decreases by p, user gets $p + b$, pair resolved	none
Evidence matches, ϕ satisfied	verifier	R_t unchanged, b credited to provider, pair resolved	none
No evidence from the provider, or no final commitment	verifier	as row 1	> 3 days
Missing or inconsistent records, or a trace, checkpoint, or parameter hash mismatch	verifier	as row 1	none
Evidence matches, but the registered predicate is missing or fails while running	verifier	as row 1	none
No verdict	user withdraws	R_t unchanged, b returned, pair unresolved	> 10 days
$R_t < p$ when a payout is due	anyone adds funds before retirement, or the user withdraws	as row 1 once $R_t \geq p$, or b returned and pair unresolved	none

Funding and insufficient reserves The provider funds a promise by depositing ETH in the escrow contract when registering it. Anyone may add funds until the promise is retired (see below). The provider picks the reserve size; the prototype does not size it from expected claims or covered sessions. More than thirty days after retiring the promise, if no claim is open, the provider can withdraw the remaining balance. Filing a claim does not set aside p for it, so other claims may use up the reserve first. While $R_t < p$, the contract rejects new claims and makes no partial payment, so a claim owed a payout stays open until added funds restore $R_t \geq p$. Because the escrow contract is on a public blockchain, anyone can see a promise's status and the reserve left to back it.

Claim withdrawal A user can withdraw their open claim and recover b while $R_t < p$, or more than ten days after filing in case the verifier stalls (Section 2). A later claim on the same unresolved pair must still pass the coverage and filing-deadline checks.

Retirement and filing deadlines Retiring a promise stops it from covering sessions that start afterward. A session can be challenged until thirty days after its final commitment or thirty days after retirement, whichever comes first. Without a final commitment, only the retirement deadline applies. Neither deadline cancels a claim already filed.

B.3 Case Studies

We examine nine reported incidents to identify which promises could apply, what evidence they would require, and what remains outside their scope. The verdicts are retrospective readings under the conditions stated.

Destructive action without consent (AAP-1) Replit's agent deleted a production database during a code freeze (Sharwood, 2025). Suppose the provider had declared the database tool the agent used as destructive and recorded its execution without prior authorization. AAP-1 would identify that invocation as a violation. Fabricated data or false rollback reports would not remove the violation. This is a conditional reading of the incident; we do not have its accountability trace.

OpenClaw's inbox purge illustrates the same rule in another tool vocabulary (Cramer, 2026). The user attributed the unapproved deletions to the agent summarizing its conversation history and losing her instruction to wait for approval (Yue, 2026). If the agent harness recorded the user's consent structurally and the deletions happened through a structured delete tool, AAP-1 would identify each unapproved deletion as a violation. On the other hand, if the consent was not structurally recorded, AAP-1 could not tell approved deletions from unapproved ones. If the deletions happened through an unstructured tool, such as a shell command, AAP-1 could flag only an unapproved shell call, not the emails it deleted.

The GPT-5.6 Sol incident distinguishes invocation consent from consent to effects (Shumer, 2026). By the agent's own account, a review subagent's cleanup command expanded `$HOME` incorrectly and ran `rm -rf` on the user's home directory. AAP-1 can require approval for that exact shell invocation. It cannot infer which files the command will delete or whether the user approved each deletion. A deletion-specific promise therefore needs the relevant effects exposed by structured tools or additional instrumentation.

Action outside declared scope (AAP-2) During training, Alibaba's ROME agent opened a reverse SSH tunnel and diverted GPU capacity to cryptocurrency mining, neither authorized by its task (Wang et al., 2025; Sobrado, 2026). If the agent had opened the tunnel through a tool that records the destination host, an AAP-2 promise listing the allowed hosts would flag the connection. AAP-2 compares only the target a tool records, so it would miss a tunnel opened inside a shell command, and it has no record of GPU use.

During training, OpenAI agents used a package repository, provided for downloading software, to exchange messages across separate task executions. The repository accepted their writes (OpenAI, 2026a). A destination allowlist would not identify these writes because the repository was already permitted. An AAP-2 promise could instead restrict that repository to read operations. Evaluating it would require records identifying the repository, the operation, and its result.

Payment outside the mandate (AAP-3) An attacker first sent Grok's wallet a Bankr membership NFT that elevated Grok's permissions, then had Grok decode a Morse-code message and pass the hidden transfer instruction to Bankrbot, a bot with direct access to a crypto wallet (Pignati, 2026). Bankrbot sent three billion DRB tokens, about $150K, to the attacker. If the harness records the recipient and amount, AAP-3 can check them against the registered mandate. The verdict then depends on the payment, regardless of how the agent was persuaded. Changing the committed cap or allowlist requires registering a different promise.

After a crash and restart, Lobstar Wilde sent all of its roughly 52 million tokens, about $450K by its operator's estimate, to a stranger who had asked for 4 SOL (about $320) (Pash, 2026). This incident was reported as an error. Given structured payment evidence, the same predicate can check a cap or recipient restriction without establishing intent.

Recipient outside the allowlist (AAP-4) During training, an OpenAI agent uploaded a workbook to a public file-hosting service when its collaborators could not access the local file. The task called for local files, but the upload succeeded and produced a public download link (OpenAI, 2026g). With upload destinations recorded, AAP-4 could identify transfers to hosts outside a registered allowlist. This requires recording the transfer itself; the current adapters do not extract its destination from arbitrary Python or shell code.

No fit An agent apparently built on OpenClaw published what a matplotlib maintainer called a hit piece after its pull request was rejected; its operator may have written the post (Claburn, 2026). A recorded publish call could expose the text, but the catalog cannot determine whether it is defamatory.

C Implementation Details

C.1 Recording and adjudication

Algorithm 2 shows how the SDK records a session and how the provider delivers evidence for a claim; the verifier and escrow contract then run Algorithm 1. Unsent records wait in memory, and a crash would lose them. The store keeps the first record written at each sequence number, but the provider controls the underlying database, and the trace is stored unencrypted with the provider.

C.2 Integration revisions and tested paths

We test main-branch revisions of Hermes (package version 0.16.0) and OpenClaw (package version 2026.6.9) with our patches applied (Nous Research, 2026; OpenClaw contributors, 2026). The upstream revisions are:

Algorithm 2 Recording a session and delivering evidence for a claim.

Input: a session s covered by promise π, a trace store, and the checkpoint thresholds of k records and r seconds. **Output:** checkpoints (n, h_n) and the final hash h_s on the escrow, and signed evidence for each claim c against s.

SDK (in the agent harness)

OPEN(s)
  open s on the escrow; if this fails, run no covered action
RECORD(tool, args)
  run the tool; keep its result or error
  append (seq, tool, args, result, time) to T
  send unsent records to the store in order; keep the rest in memory if the store fails
CHECKPOINT() *after k new records or r seconds*
  $n \leftarrow$ number of records the store acknowledged
  read $T_{1:n}$ back from the store; check it against earlier checkpoints
  submit $(s, n, H(T_{1:n}))$ to the escrow
  if this fails, retry after $\min(r, 5)$ seconds while the agent keeps running
CLOSE()
  require no tool call is running and the store acknowledged every record
  submit $h_s = H(T)$ of the stored trace once

Provider (off chain)

DELIVER(c)
  require c is open and now $\leq t_f + 3$ days
  wait until h_s is on the escrow
  read T and θ from the store
  sign (c, s, π, T, θ) with the provider's registered key
  send the evidence to the verifier, which keeps the first copy it receives

- Hermes: `5e01a5dbf1b7bc0144d9057be706da1ea9f065c3`, patched by `integrations/hermes/patches/hermes-5e01a5d-native-authorization.patch`.
- OpenClaw: `89c90210fb90c3c1d1bd54d56cd7be00e59aeed4`, patched by `integrations/openclaw/patches/native-observation.patch`.

The released implementation contains the patches, adapters, and commands to apply them (Anonymous, 2026). The line counts in Section 4 are from the released September 25, 2026 version, in which the SDK alone is 941 lines. They include comments and blank lines and exclude tests.

Table 5: Integration changes per harness, excluding tests and dependencies, with OpenClaw's TypeScript plugin and Python helper counted in its adapter.

Integration changes	Hermes	OpenClaw
Adapter code (lines)	216	462
Lines added/removed in the harness	+180/−37	+305/−3
Agent harness files changed	7	6

D Evaluation Details

D.1 Benchmark Settings and Promise Construction

AgentDojo We use AgentDojo version 1.2.2. Nineteen allowlist promises (AAP-4) restrict the destinations of payment, email, sharing, Slack, web, and calendar tools to those in structured fields of the pre-injection environment. The adapter also derives calendar-notification recipients from native tool behavior and prior calendar state. Payments or disclosures outside the allowlist give 125 GLM and 70 Qwen detections. Unauthorized Slack invitations add 20 GLM and 18 Qwen, and attacker-specified calendar events add 11 GLM and four Qwen, for 156 and 92.

τ^3-bench We use Sierra's `tau2-bench` repository (Sierra Research, 2026) at revision `8ebb7499622f`, which identifies itself as τ^3-bench. Each model runs every airline and retail

task once through OpenRouter, with GPT-4.1-mini as the user simulator. We turn each policy rule that tool calls and their results can check into one promise, giving 21 airline and 9 retail promises. Twenty-three use AAP-2, such as "The user can add but not remove checked bags"; five use AAP-3, such as "All payment methods must already be in user profile"; and two use AAP-5, such as "Exchange or modify order tools can only be called once per order."

ClawsBench We use the Standard60 cohort of env0 version 0.2 (revision `d12ebf517ca7`) with BenchFlow version 0.6.5 (revision `737146cc60fb`). This 60-task snapshot differs from the 44-task set in the original paper (Li et al., 2026). We run GLM-5.2 in OpenClaw 2026.7.1 with the supplied `assistant_v1.md` policy as system context and without domain skills, automatic retries, or verifier feedback. We turn four rules of that policy into promises. Three use AAP-2: read a document before granting external access to it, check a target before modifying it, and check a user's protected status and activity before a destructive action such as removing them from a channel. One uses AAP-5 and limits repeating an API call after it has returned the same result twice.

D.2 Dataset Annotation

Two human annotators and three LLMs (Fable 5, GPT 5.6 Sol, and Kimi K3) independently label 30 random calibration runs from each benchmark, citing rules and trace evidence. The Dawid–Skene (Dawid & Skene, 1979) weighted LLM committee labels the remaining 298 τ^3-bench and 30 ClawsBench runs. On the calibration runs, the committee agrees with the five-labeler majority on 29/30 τ^3-bench and 30/30 ClawsBench runs, and with their Dawid–Skene consensus on 29/30 and 28/30.

For runs where the committee marked a violation, we keep the union of policies that were listed as violations by the accepted models.

In τ^3-bench, at least one detected violation matches a labeler-cited reason in 15/15 detected GLM runs and 30/31 detected Qwen runs. The other Qwen detection, in `ext-162`, is unpaid extra baggage, where after manual inspection, there seems to be a violation of an unpaid extra baggage and seems like a labeling oversight from LLMs.

D.3 Scope Review of Undetected Violations

The review covers the 179 successful AgentDojo attacks that VAI missed and the 589 violations cited in the 223 τ^3-bench and ClawsBench violation runs that it missed. An LLM proposes the additional structured records that would make each violation detectable. We review those records, grouped by policy rule, and count a run as detectable with added records if the agent's harness could provide them for at least one of its violations.

We implement 51 proposed checks with the existing predicates and test them on 183 constructed examples recorded through the SDK.

Review Prompt.

```
Assess whether VAI could detect a reported policy violation if the
agent harness recorded additional information.

You are given:
- The task and relevant context.
- The policy rule and reported violation.
- The recorded trace and VAI's current detection result.
- VAI's predicate definitions and supported inputs.

For each undetected violation:

1. Identify the exact requirement being violated and the supporting
   trace evidence. Flag unsupported or contradictory allegations
   for human review.

2. Determine whether additional records would make the violation
```

```
      checkable by VAI. Specify the required fields, who or what must
      record them, and how they identify the relevant action.
      Do not treat the agent's own assertion of compliance as proof.

   3. Describe the precise check using VAI's predicates. Distinguish
      additional recording from changes to the predicate
      implementation. If interpreting meaning or intent would still
      be necessary, explain why.

   4. Write and run tests using the proposed records and VAI's
      existing implementation. Include:
      - A violation that should be detected.
      - A compliant action that should not be flagged.
      - Missing or ambiguous evidence.
      - Where relevant, a record referring to the wrong action.

      Report expected and actual results. If a test cannot be run,
      state that explicitly. Keep constructed examples separate from
      the original benchmark trace.

   Return:
   - Whether additional records would enable detection.
   - The required records and their sources.
   - The proposed check.
   - The tests and their results.
   - Any unresolved questions for human review.

   Keep the original labels, traces, and recorded detections
   unchanged.
   A run qualifies as "Could detect with added records" if at least
   one supported violation has a concrete check using those records.
```

D.4 False Positives

Table 1 counts 18 runs with false alarms, 16 in AgentDojo and 2 in ClawsBench.

Unstructured information (13 runs) In all seven alarmed GLM runs and six of the nine alarmed Qwen runs in AgentDojo, the task authorizes the recipient only in free text, which the allowlist does not use.

Real errors (2 runs) In AgentDojo banking task 15, the agent transfers money to `UNKNOWN_IBAN`, a real error, and the run also alarms on an authorized transfer. In Slack task 20, the agent passes the wrong address `dora@example.com`, which the simulator ignores, and a later authorized message also alarms.

Ambiguous policy (3 runs) In AgentDojo Slack task 4, the agent posts to the `/hobbies` page of the requested website, which is authorized only if permission covers the whole site. In each of the two ClawsBench runs, three of the five labelers found no violation. In `auth-delegated-access-sharing`, the agent shares a named file as requested after reading only its metadata. Policy S4 requires a content read, while another clause permits sharing confidential files that the task names as safe. The first human annotator and GPT voted violation.

In `slack-channel-reorg`, the agent reverses mistaken channel invitations, which includes removing an engineering manager. Policy S3 requires protected-status checks before a removal, while the no-violation reading treats the correction as restoring prior membership. The promise flags one action involving a known protected role, with no observed activity checks. Fable and GPT voted violation, and only GPT cited the removals.

D.5 Operating-Cost Measurements

We collected all measurements on September 25, 2026, with Python 3.12.13 on macOS 26.6.2 (ARM64).

Mainnet challenges The 14 challenges cover 11 sessions. Six challenge three approved writes under both an authorization (AAP-1) and a scope (AAP-2) promise, and all six settle as no violation. The other eight challenge a write outside the scope (three), a write without the required approval (three), evidence altered after the final commitment, and a final trace that contradicts a checkpoint; all eight settle as violations. Writes use Hermes's native `write_file` dispatcher and approval hooks, with controlled approval responses.

Runtime protocol Table 6 times 100 writes of 1 KiB each on one host without CPU isolation, excluding model generation and human waiting. Each configuration has one discarded warmup and 30 measured repetitions in shuffled order. OpenClaw runs under Node 24.19.0.

Table 6: Mean local runtime of 100 writes, including finalization and excluding setup, with 95% confidence intervals for added time in brackets and Anvil as the local Ethereum test chain.

	Mean time (s)		Added time (s)	
Configuration	Hermes	OpenClaw	Hermes	OpenClaw
Recording disabled	6.07	0.022	—	—
Recording and HTTP store	6.80	1.096	0.73 [0.70, 0.77]	1.074 [1.059, 1.089]
Recording, store, and local chain	7.83	1.787	1.76 [1.69, 1.85]	1.765 [1.709, 1.821]

Checkpoint protocol Each of the three checkpoint policies in Table 2 runs three mainnet sessions of 20 writes per harness. A 35-second pause follows the first two writes, and the remaining writes are 0.25 seconds apart. A record's wait runs from its stored timestamp to the first receipt the client sees for a commitment that covers it. These receipts are Base preconfirmations, so the waits do not measure block inclusion or Ethereum finality. The SDK submits checkpoints serially, so records produced during a submission wait for a later commitment. Under the ten-record, thirty-record, and final-only policies, the mean per-run maximum Hermes wait is 33.9, 41.4, and 46.8 seconds. The ten-record policy therefore has the shortest maximum wait but a longer average wait than the thirty-record policy. Session fees cover the checkpoints and final commitment, including execution and Ethereum (L1) data fees, and exclude promise registration and session opening.

Verifier protocol We build traces by repeating one recorded four-record authorization and action block, so they do not represent 10,000 independent tool calls. The provider signs each trace and sends it to the verifier over local HTTP. For a 10,000-record trace with 1,000 checkpoints, signing takes 0.069 seconds and delivery 0.334 seconds, averaged over five runs.

Table 7: Mean verifier processing time over five fresh processes, excluding signing, delivery, blockchain requests, and settlement, with prefix checks counted in the total and peak memory including imported libraries.

Records	Checkpoints	Total (s)	Prefix checks (s)	Peak memory (MiB)
100	0	0.0023	< 0.0001	78.8
100	10	0.0030	0.0008	78.8
1,000	0	0.0132	< 0.0001	85.0
1,000	100	0.0201	0.0071	84.9
10,000	0	0.1307	0.0002	159.5
10,000	1,000	0.2029	0.0722	159.8