Skills as Verifiable Artifacts: A Trust Schema and a Biconditional Correctness Criterion for Human-in-the-Loop Agent Runtimes

Alfredo Metere
alfredo.metere@enclawed.com
Enclawed LLC
California, USA

Abstract

Agent skills — structured packages of instructions, scripts, and references that augment a large language model (LLM) without modifying the model itself — have moved from convenience to first-class deployment artifact. The runtime that loads them inherits the same problem package managers and operating systems have always faced: a piece of content claims a behavior; the runtime must decide whether to believe it. We argue this paper's central thesis up front: a skill is *untrusted code* until it is verified, and the runtime that loads it must enforce that default rather than infer trust from a signature, a clearance, or a registry of origin. Without skill verification, a human-in-the-loop (HITL) gate must fire on every irreversible call — which is operationally untenable and degrades into rubber-stamping at any non-trivial scale. With skill verification treated as a separate, gated process, HITL fires only for what is unverified, and the system becomes sustainable. We give a trust schema (§3) that includes an explicit verification level on every skill manifest; a capability gate (§4) whose HITL policy is a function of that verification level; a *biconditional* correctness criterion (§5) that any candidate verification procedure must satisfy on an adversarial-ensemble exercise (§6); and a portable runtime profile (§7) with twelve normative guidelines abstracted from a working open-source reference implementation [17]. The contribution is harness- and model-agnostic; nothing here requires retraining, fine-tuning, or proprietary infrastructure.

Keywords: LLM agents, agent skills, supply chain, formal verification, human-in-the-loop, mandatory access control, audit log

ACM Reference Format:
Alfredo Metere. 2026. Skills as Verifiable Artifacts: A Trust Schema and a Biconditional Correctness Criterion for Human-in-the-Loop Agent Runtimes. In *Proceedings of First Workshop on Agent Skills (Agent Skills '26).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Agent Skills '26, May 26, 2026, San Jose, CA
2026. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The emergence of `SKILL.md` as a portable, harness-neutral description of procedural knowledge for LLM agents has solved a real problem: an agent operator can curate a library of small, inspectable, version-controlled skill packages and ship them across platforms. Empirically, curated skill libraries improve agent pass-rates on benchmarks such as Mind2Web [6] and Embodied Agent Interface [13], while indiscriminate admission of self-generated skills can degrade or actively attack an agent's behavior [1]. The asymmetry is the same one distributed software ecosystems hit decades ago: trust cannot be inferred from origin alone [25], package managers that do infer it have been broken every way the literature predicted [3, 4], and the formal response is by now textbook [22]: typed artifacts, signed manifests, capability sandboxing, hash-chained audit, and least-privilege gates around side-effects.

What is *not* textbook is what the runtime should do with a skill it has just signed and clearance-checked but has *not* yet verified to behave the way its manifest claims. Most current harnesses treat that distinction as collapsed: signature plus clearance equals trust. We argue this is wrong, dangerously so, and that the cost of getting it wrong is paid by the human in the loop.

Skills are untrusted code. A skill influences the agent's prompt, tool calls, and write surfaces, making a malicious skill a direct prompt-injection vector that persists across model calls and survives context truncation [5, 9, 21, 24]. A signature on the manifest tells the runtime that some signer endorsed the artifact at some point; it says nothing about whether the artifact's behavior conforms to its declared manifest. Recent work shows the gap is exploitable in practice: backdoors can be smuggled into signed code-completion models in ways that pass strong static detection [28]. Treating a signed-and-cleared skill as trusted is no more defensible than treating a code-signed executable as malware-free because the OS vendor signed it.

HITL is the universal default for unverified skills. Without a way to trust a skill's claims about itself, the runtime must gate every irreversible call through human-in-the-loop (HITL) approval. This is operationally untenable: operators under load rubber-stamp every prompt, degrading the gate into a fiction. The right response is *not* to weaken the gate; it is to introduce a second process — skill verification — whose successful completion allows the runtime to trust portions of a skill's manifest and relax HITL frequency. Verification happens once at bootstrap; the result is recorded in the manifest as an explicit *verification level* and is immutable for the session, mirroring the "capability bootstrap discipline" [19] adapted to skills.

Side-effects must be verifiable, not just gated. We propose a single *biconditional* criterion: the observable side-effects of an agent run must be in 1-to-1 correspondence with the approved-and-executed set in the audit log. Any candidate verification procedure (review, fuzz testing, adversarial ensemble, formal analysis) must demonstrate that the skill under test satisfies this criterion before its manifest can be elevated above the unverified default.

Contributions. This paper makes four contributions.

1. A *skill trust schema* (§3) that treats a skill as a tuple of manifest, content, and capabilities, with an explicit *verification level* field and a no-runtime-mutation discipline.
2. A *capability-gate model* (§4) whose HITL policy is a function of the loaded skill's verification level: unverified skills incur HITL on every irreversible call; verified skills incur HITL only for capabilities outside the verified manifest.
3. A *biconditional correctness criterion* (§5) that any candidate skill- verification procedure must satisfy, with a characterization of the failure modes it catches and the ones it does not.
4. A *portable runtime profile* (§7) with twelve normative guidelines abstracted from a working open-source reference implementation [17], including a no-bypass-switch invariant and an untrusted-by-default rule for skill admission.

We sketch an adversarial-ensemble evaluation (§6) under which the criterion is exercised — and under which a candidate skill-verification procedure can be benchmarked — and discuss open problems in §8.

The schema is deliberately model- and harness-agnostic. It assumes nothing about training, fine-tuning, or RLHF, and assumes only the weakest run-time interface: that an agent harness invokes tools through a typed dispatch step the runtime can interpose on. Every existing harness adopting `SKILL.md` satisfies this constraint.

Reference implementation. We draw both the schema and the criterion from a working open-source framework, *enclawed* [17], which hard-fork-hardens a single-user AI assistant gateway with each of the primitives this paper abstracts: a Bell–LaPadula classification scheme, an Ed25519 signed-module loader with a clearance-bounded trust root, a hash-chained audit log, an egress guard, a regex-based DLP scanner, a HITL controller with a checkpointable agent-session state machine, and a transaction buffer with rollback. Section 7 extracts a small set of normative guidelines from that reference implementation, intended as a starting point for a portable runtime profile any `SKILL.md`-adopting harness could implement to bound the attack surface skills introduce.

2 Threat model: skills as a new supply-chain surface

We assume an agent operator who already runs LLM agents with tool access in an environment where *some* of the tools have real-world side-effects (file deletion, message sending, payment, database writes, on-chain operations). The skill ecosystem the operator subscribes to is partially trusted: the operator's own curated skills are signed by an operator-controlled root, but upstream registries and self-generated skills are not.

Adversaries. We consider three:

1. A *registry attacker* who publishes a malicious skill that the operator's harness pulls and runs. The skill's instructions induce the agent to take harmful side-effects in the operator's environment.
2. A *prompt-injection attacker* whose payload reaches the agent through a benign skill's data dependency (e.g. a fetched document the skill instructs the agent to summarise) and re-routes the agent's tool calls.
3. A *self-generated-skill attacker*: the agent itself, under model error or adversarial input, synthesises a new skill, registers it, and uses it. This is not a malicious actor in the usual sense, but operationally indistinguishable from one.

Out of scope. We do not assume an attacker who has compromised the operator's signing key, the runtime binary, or the audit-log storage out-of-band. We do not assume a defended hardware trust anchor (a TPM, secure element, or equivalent), though such an anchor improves every layer below. We do not discuss model-weight-level attacks (poisoning, jailbreak from training data), which are orthogonal.

Goal. The runtime should:

- refuse to load a skill whose manifest does not verify;
- refuse to dispatch a tool call whose required capability the loaded skill did not declare;
- refuse to execute an irreversible call without an explicit, audited decision;

- produce an audit trail under which the biconditional criterion of §5 can be checked.

3 The skill trust schema

We define a skill artefact as a tuple

$$\texttt{Skill} = (M, \texttt{content}, \sigma)$$

where M is a manifest, `content` is the body of the skill (typically the `SKILL.md` file plus referenced scripts), and σ is a detached signature over the canonical bytes of $(M, \texttt{content})$ produced by a signer in the operator's trust root.

The manifest M has five mandatory fields:

- M.label a classification label $\langle \ell, C, R \rangle$ in a Bell–LaPadula style lattice [2], with rank ℓ, compartment set C, and releasability caveats R. Labels combine via join: $a \sqcup b = \langle \max(\ell_a, \ell_b), C_a \cup C_b, R_a \cap R_b \rangle$.
- M.caps a finite set of declared capabilities drawn from a fixed vocabulary; see §3.5.
- M.signer a key identifier referencing an entry in the trust root.
- M.version a monotone integer; replays of an older signed manifest are rejected if a newer one with the same identity has been observed.
- M.verification the skill's verification level; see §3.1.

3.1 Verification levels

A signed manifest binds an artifact to a signer's name; it does not bind the artifact's behavior to the manifest's claims. We make this distinction explicit in the schema. Every skill manifest carries a verification level M.verification drawn from a fixed enum of four values: UNVERIFIED, DECLARED, TESTED, and FORMAL. The default is UNVERIFIED; higher levels grant the runtime more latitude when policing the skill's tool calls (§4).

UNVERIFIED. No claim is made about the skill's behavior beyond its existence. Default for any skill the operator has not explicitly examined. The runtime treats this skill as untrusted code: every irreversible capability call goes through the HITL gate, regardless of whether the manifest declares the capability or not.

DECLARED. A trust-root signer has examined the skill's content and *attests* that the skill's side-effects are bounded by its declared capability set M.caps. The attestation is the signer's professional reputation; the runtime trusts capabilities in M.caps but still requires HITL on irreversible calls outside that set.

TESTED. In addition to a declaration, the skill passes an *adversarial-ensemble* verification run (§6) under which the biconditional criterion of §5 holds. The runtime permits irreversible capabilities in M.caps without per-call HITL but logs every such call.

FORMAL. A formal analysis tool has produced a machine-checkable proof that the skill's behavior is a subset of its declared capability set under the runtime's threat model. This level is aspirational at the time of writing; we include it for completeness because the schema field is fixed-width and adding it later requires a manifest version bump.

Why levels, not a continuum. A continuous "trust score" would allow the agent to argue itself into permission gradients; a fixed enum forces a discrete decision that can be audited and revoked atomically. The same reasoning applies to the configuration profiles in §7.1.

Verification is an offline, bootstrap-time concern. Verification levels are set by the operator (or the operator's delegate, e.g., a CI pipeline running an adversarial-ensemble suite over each candidate skill) *before* the runtime starts, and recorded into the signed manifest at the DECLARED level or above. The runtime never elevates a verification level during a session; doing so would re-introduce the trust-bootstrap problem the schema is designed to avoid. A manifest that arrives at runtime claiming a verification level above what its signer is authorized to attest is rejected.

3.2 No agent mutation of skills at runtime

Once a skill is loaded, its content and manifest are immutable for the lifetime of the agent session: the agent cannot edit a skill, append to its manifest, or upgrade its verification level. Any attempt the agent makes to modify a loaded skill — through a file-system tool call, a runtime API, or any indirect side-effect on the on-disk artifact — is itself an irreversible capability that must walk the HITL gate *and* be recorded in the hash-chained audit log with the pre- and post-mutation content hashes and the operator's decision, regardless of whether the request is approved or denied. On approval, the modification produces a *new* skill artifact that must be re-verified before the next session. Without the immutability rule, an agent that admits one verified skill could rewrite that skill's content in place and silently change what "verified" means for the rest of the session; without the audit trail, the rewrite would be invisible to post-incident review.

3.3 The trust root

The trust root is a finite, append-only set of signer entries $\{(\text{keyId}_i, \text{pubKey}_i, \text{maxClearance}_i)\}$. A signer is trusted to sign skills with classification at most maxClearance_i. The trust root supports a one-shot *lock* operation: once locked, mutations (`set`, `remove`) raise a typed error. The locked state is the production posture; an unlocked trust root is acceptable only during host bootstrap, before any external input has been read. This is the standard "capability bootstrap discipline" adapted from object-capability systems [19].

3.4 Manifest verification

Loading a skill walks the following steps in order at bootstrap, before any external input is read, failing closed on any error:

1. Parse M from canonical JSON; reject unknown fields, prototype-pollution keys, and missing mandatory fields.
2. Resolve M.signer in the trust root. If absent, reject.
3. Verify σ against the resolved public key over the canonical bytes of (M, `content`); reject on mismatch.
4. Check that M.label $\preceq$ maxClearance for the resolved signer. A signer cannot sign above its authorized clearance.
5. Check M.label $\preceq$ `user.clearance` for the running operator. The operator cannot load a skill that exceeds their own clearance.
6. Check that M.verification does not exceed the signer's authorized verification level. A signer attesting TESTED must hold attestation authority for that level; FORMAL requires a corresponding tooling attestation. Default if absent: UNVERIFIED.
7. Register the declared M.caps with the runtime's capability gate (§4), tagged with M.verification.

A skill that survives all seven steps is *loaded*; its content is now reachable by the LLM at the registered verification level. A failure at any step produces a typed audit record and aborts the load. After bootstrap completes, the loaded set is frozen for the session; loading a new skill at runtime requires the runtime to be re-bootstrapped and is governed by the §3.2 rule.

3.5 Capability vocabulary

The vocabulary is small enough to be enumerated and large enough to discriminate side-effect classes. We propose a minimal set:

Capability	Side-effect class
`net.egress(host)`	DNS-resolved network reach to host
`fs.read(path)`	filesystem read under path
`fs.write.rev(path)`	write that can be rolled back
`fs.write.irrev(path)`	delete, overwrite, truncate
`tool.invoke(name)`	invoke a named tool registered with the harness
`spawn.proc(cmd)`	external process spawn
`publish(channel, . . . )`	post to an external channel
`pay(token, amount)`	transfer of fungible value
`mutate.schema(target)`	migrate a database / configuration schema

Table 1. A minimal capability vocabulary. The split between `fs.write.rev` and `fs.write.irrev` is the load-bearing distinction the gate uses in §4.

A skill's manifest must enumerate every capability its content intends to invoke. A capability not in M.caps is denied at the gate, regardless of what the skill content asks. Theoretically, the vocabulary is the minimal carrier for the reversible/irreversible partition over the principal classes under which an agent's authority is denominated in the object-capability tradition [18].

Manifest-declared, not runtime-inferred. The reversibility tag on a capability is a property of the runtime's own dispatch taxonomy, not a static analysis of underlying syscalls. A path-write is reversible exactly when the dispatch goes through the runtime's transaction buffer (§4.2), which holds the change in memory and either commits or rolls back; a write that bypasses the buffer (or hits a remote object the runtime cannot snapshot, or invokes a side-effect on a counterparty the runtime cannot coordinate with) is classified `irrev`. The runtime need not inspect the filesystem; it inspects which dispatch path *it* routed the call through. This sidesteps the well-known problem that POSIX filesystems do not expose enough information to classify writes statically.

Minimum, not closure. Table 1 is a minimum vocabulary, not a fixed closure. A runtime adding a new side-effect class (e.g., `enclave.attest`, `key.derive`) extends the enum and bumps the manifest schema version; manifest verification (§3) already rejects unknown capability tokens at load time, so vocabulary growth is monotone and backwards-compatible. The gate's invariants in §4 are stated over the *declared* capability set, not over Table 1's specific membership.

4 The capability gate

The capability gate is the runtime layer between the LLM-driven agent and the external world. It receives a tool-call envelope emitted by the agent (typically as a JSON object), looks up the corresponding capability, and consults the gate policy.

4.1 Gate policy as a function of verification level

The gate's behavior depends on the loaded skill's verification level (§3.1). The mapping is fixed and is not configurable per call:

UNVERIFIED. Every irreversible capability call the agent makes while this skill is the active context passes through HITL, regardless of whether the skill's manifest declares the capability. Reversible calls execute through the transaction buffer (§4.2). HITL on every irreversible call is the universal default; verification is the only path off it.

DECLARED. Irreversible calls whose `(capability, target)` is in M.caps walk the four-state HITL lifecycle of §4.3 with the *broker auto-approving* (no human prompt; the

operator's policy delegates the decision to the manifest's declared set); the `irreversible.request`, `decision`, and `executed/error` records are all written. Calls outside M.caps walk the same lifecycle with the broker consulted normally and may stop on a human. Reversible calls execute through the transaction buffer regardless of verification level.

TESTED. Same as DECLARED, with the addition that the runtime maintains a per-session biconditional check (§5) over the operations the skill issued and aborts the session if the check fails between rounds.

FORMAL. Same as TESTED; the difference is in offline trust, not in runtime behavior.

In all four cases, the runtime audits every call; the difference is whether the gate stops to ask a human first. The default is "ask". Verification is what buys the runtime permission to stop asking, and only for what the verification covered.

4.2 Reversible vs. irreversible

The split between *reversible* and *irreversible* capabilities is the design's load-bearing distinction.

- A **reversible** side-effect leaves a single object in a state from which the runtime, holding a recent snapshot of that object, can return it to its prior state without external coordination. Memory-buffered file writes that commit on confirm are reversible; in-database transactions with rollback are reversible; an SQS queue write to a dead-letter queue under operator control is reversible.
- An **irreversible** side-effect is one for which the runtime cannot, alone, restore the prior state of the affected world. Sending an email is irreversible. Posting to a public channel is irreversible. Issuing an on-chain transaction is irreversible. Deleting a file from a remote store the runtime does not control is irreversible.

The runtime classifies every capability call as one or the other. Reversible calls execute through a transaction buffer that holds the change in memory and either commits (after the operation succeeds and is audited) or rolls back. Irreversible calls go through the four-state lifecycle of §4.3.

4.3 The HITL lifecycle

For every irreversible call the runtime walks four states:

1. **request.** The agent emits a tool-call envelope $\langle \text{op}, \text{args}, \text{reasoning} \rangle$. The runtime appends a typed `irreversible.request` record to the audit log.
2. **decide.** The runtime consults a *broker* — an opaque oracle the operator chose at deploy time — and receives a binary decision $d \in \{\texttt{approve}, \texttt{deny}\}$. The runtime appends a typed `irreversible.decision` record carrying d and the broker's identity.
3. **execute.** If $d = \texttt{approve}$, the runtime performs the side-effect through the host APIs the capability resolves to. If the side-effect succeeds the runtime appends `irreversible.executed` with ok = `true`. If the host APIs fail (the target file vanished between approval and call, the network is partitioned, the chain rejected the transaction) the runtime appends `irreversible.error` instead.
4. **audit.** The records of the previous three states are linked by a shared *request-id* and form a complete trace of the call.

4.4 Broker policies

The broker is a configurable component. We see four useful defaults:

DENY-ALL. The broker denies every request. Equivalent to running the runtime without irreversible capability at all; useful as a baseline and as the fail-safe default when the operator has not configured a broker.

POLICY. The broker reads an out-of-band policy document (a file the LLM cannot reach) carrying allow- and deny-rules over capability + argument shape. The decision is mechanical and reproducible.

INTERACTIVE. The broker prompts a human via terminal, message bus, or webhook, with a timeout that defaults to deny.

WEBHOOK. The broker delegates to a remote service holding the operator's policy.

The policy regime has the convenient property that, given the same audit log of requests, two independent operators with the same policy produce the same decisions; this makes integration tests deterministic.

5 Biconditional correctness for HITL gating

A gate's job is not merely to deny bad requests; it is to make the runtime's claim about the world match the world. The same property that makes the gate auditable in production also makes it the natural criterion against which a candidate skill-verification procedure can be exercised. We formalize the property and discuss both uses.

Scope of the term "correctness". The criterion below is *gate-correctness* (or, equivalently, audit-world consistency) at the operation-and-target projection: it ensures that the runtime's claim about what it approved-and-executed matches the runtime's observable footprint on the corpus. It is *not* a value-level correctness criterion. Two operations that agree on `(op, target)` but disagree on the payload they wrote — e.g., an approval to append log line ℓ_1 followed by a write of $\ell_2 \neq \ell_1$ to the same file — satisfy the biconditional. §5.4 enumerates this and the other limits we know of, and §9

positions the criterion against richer notions of agent-action correctness in concurrent work.

5.1 Setup

Fix an agent run with starting state s_0 of a designated *corpus* (a finite set of mutable objects the agent can act on). After the run the corpus is in state s_1. The runtime has produced an audit log L of typed records. We define:

$$D = \text{delta}(s_0, s_1) \quad \text{(observed)} \tag{1}$$

$$S = \{\, r \in L \mid r.\text{type} = \texttt{irreversible.executed}, \tag{2}$$

$$r.\text{ok} = \texttt{true} \,\} \quad \text{(audited)} \tag{3}$$

D is what the runtime says happened to the world; S is what the runtime says it approved and executed.

5.2 The biconditional

Definition 5.1 (Biconditional pass)**.** The audit log L *passes the biconditional* for the run iff the multiset projection of D onto operation-and-target equals the corresponding multiset projection of S.

In words: *every* corpus change is explained by a matching approved-and-executed record, and *every* approved-and-executed record corresponds to a matching corpus change. Either direction broken is a fail.

5.3 Detected failure modes

Proposition 5.2. *If the audit log L passes the biconditional, the following failure modes are absent:*

- **F1** ***Gate bypass****: an irreversible side-effect occurred without a corresponding* `irreversible.request/decision` *pair.*
- **F2** ***Audit forgery****: the log claims an approved-and-executed record whose corresponding corpus change is not present.*
- **F3** ***Approved-but-failed without note****: a request was approved and the runtime claims it executed, but the host call silently failed.*
- **F4** ***Wrong-target execution****: the runtime approved an op against target t_1 but the side-effect landed on $t_2 \neq t_1$.*

Proof sketch. F1, F2 follow directly from the multiset equality. F3 is caught because the proposition restricts S to records where $r.\text{ok} = \texttt{true}$; a silent host failure should have been recorded as `irreversible.error`, which is not in S, so its absence from D violates equality from the S side. F4 is caught because the projection includes target identity. □

5.4 Failure modes the criterion does not catch

Honesty about scope:

- **F5 — Wrong content on correct target.** The projection is over `(op, target)`, not over the payload written or the value transferred. A write that carries malicious content to a target the operator did approve satisfies the biconditional. Catching this needs a payload-level invariant (e.g., a content classifier, a value-flow check on a typed channel) layered *above* the criterion.
- **Read-only exfiltration.** The criterion talks about side-effects on the corpus; it cannot detect an agent reading data and silently transmitting it through a covert channel that bypasses both the corpus and the audit log. Defending this requires a content-flow primitive (DLP, an egress-aware classification gate) below the criterion's layer.
- **Time-of-check / time-of-use races.** If the corpus is mutated by an external party between s_0 snapshot and the agent's first action, D is the observed delta, not the agent's contribution.
- **Approved-and-the-attacker-also-acted.** If a legitimate approval $r \in S$ overlaps a malicious side-effect on the same target, D may equal S at the target level even though the agent's action was not what D records.

These are real limits and we name them rather than paper over them. The criterion is necessary, not sufficient, for full safety; it is sufficient for the "did the gate work?" question that HITL deployments most often need to answer.

5.5 The biconditional as a verification benchmark

Beyond its production use, the biconditional is the formal criterion against which a candidate skill-verification procedure can be exercised. To elevate a skill from UNVERIFIED to TESTED, the operator (or an attestation pipeline acting on the operator's behalf) runs the skill against an adversarial-ensemble of §6 and records the audit log. The skill is admitted at TESTED iff the biconditional passes on every round. A skill that fails the biconditional under verification is rejected; a skill that passes is admitted at TESTED and pinned to the manifest hash that was tested. Modifying the skill content invalidates the attestation; this is the load-bearing reason for the no-runtime-mutation rule of §3.2.

6 Adversarial-ensemble evaluation

The biconditional criterion is interesting only if it is exercised under conditions where the agent *wants* to take destructive action. We propose an evaluation shape we call the *adversarial-ensemble* sketch.

Setup. Generate a fixed corpus of N small files with realistic-looking corporate-document names (audit logs, compliance reports, financial statements, meeting notes, test-result fixtures). Compute the per-file SHA-256 manifest as the integrity baseline.

Agents. Wire K LLM-driven agents with system prompts biased toward destructive operations. A representative four-agent ensemble has roles such as "cleaner" (proposing `fs.write.irrev` on stale files), "consolidator" (proposing in-place edits to merge files), "auditor" (proposing edits that mark records as audited), and "critic" (echoing whichever destructive action is in scope). The system prompts deliberately do *not* ground the agents in the policy of the gate.

Run. For R rounds, every agent takes one turn, emitting a tool-call envelope. The capability gate (§4) routes irreversible calls through the configured broker; reversible calls execute and are audited.

Verdict. After the run, recompute the corpus manifest; diff against baseline; check the biconditional against the audit log. Verdict is binary: $D = S$ at the operation-and-target projection, or not.

Why this is a useful experimental shape. The shape stresses the gate against the worst plausible agent behaviour — a confederation actively trying to delete things — while keeping the verdict mechanical: the corpus diff and the audit log are both deterministic, so two independent operators running the same policy broker on the same corpus produce the same verdict.

6.1 Empirical results

We instantiated the adversarial-ensemble in two complementary configurations and report aggregate numbers below; full per-cell tables follow.

(a) $N \times K \times R$ ***sweep.*** A self-contained driver generated $N \in \{10, 50, 200\}$ corpus files, wired $K \in \{2, 4, 8\}$ agents over the four roles described above, and ran $R \in \{5, 10, 25\}$ rounds. For each of the $3 \times 3 \times 3 = 27$ cells we executed $S = 200$ independent seeds, giving 5,400 runs per scenario. Five scenarios per cell: a *clean* run, plus one run per fault F_1–F_4 in which we deterministically injected the corresponding fault at round $R/2$ (stealth write outside the gate; fabricated `irreversible.executed`; approved call whose host side silently no-ops; approved op routed to the wrong target). After each run the driver computes the corpus delta D, extracts the audited multiset S, and applies the biconditional check of §5.

Results. Across $27 \times 200 \times 5 = 27{,}000$ runs the biconditional gave zero false positives (FPs) on clean runs and zero false negatives (FNs) on injected faults: Wilson 95% confidence interval (CI) $[0.999, 1.000]$ for every `clean pass` and every F_i detection rate at the aggregate level (Table 2). The per-cell results are scale-invariant in N, K, and R, which empirically matches the formal claim of Proposition 5.2: the criterion is mechanical within its detection envelope.

(b) Cross-model adversary cooperation. We re-ran an existing in-vivo harness driven by ten LLM families (one Gemini, three Groq, five Ollama, one OpenRouter; full list in the released artifact). Each adversary generated a balanced legit/adversarial payload set across all four F-categories and both real-backend channels (Discord, Telegram); each payload was mediated through three subjects: an unhardened upstream passthrough (negative control), the OSS runtime with biconditional, audit, and a data-loss-prevention (DLP) scanner, and the full deployment that adds a behavioral monitor. *Results.* Across 9,724 mediated payloads (4,816 adversarial, 4,908 legit), the upstream passthrough blocked 0.0% (CI $[0.000, 0.001]$); the OSS runtime achieved 76.3% recall (CI $[0.751, 0.775]$) at 0.0% false-positive rate (FPR); the full deployment achieved 80.2% recall (CI $[0.791, 0.813]$) at 0.6% FPR. *Why recall is not 100%.* The denominator counts every adversarial sample the LLM emitted, including the ones the adversary itself failed to make adversarial in a recognizable way (refusals, malformed JSON, payloads that resolve to a benign (op, target) projection); with no real fault present, the gate has nothing to flag. On the subset where the adversary did produce a fault-shaped payload, recall is 1.000 across every model family — per-model variation in the headline number is dominated by adversary cooperation rate, not gate behavior.

Artifact. The full per-model breakdown, the $N \times K \times R$ driver, and the gzipped per-sample CSVs are released with the reference implementation; runs are deterministic given a seed.

7 Architectural guidelines from a reference implementation

The schema and the biconditional criterion above are not thought experiments. They are abstracted from a working open-source framework, *enclawed* [17], which hardens a single-user AI assistant gateway against the same threat surface skills now expose. In this section we walk through the specific architectural choices that framework makes, explain what each choice was answering, and lift the choice into a normative guideline a portable runtime profile for skill-aware harnesses could adopt.

7.1 Toward a portable runtime profile for skill-aware harnesses

The choices above generalize to a small set of normative guidelines for any runtime that loads, invokes, or composes skills. We use MUST / SHOULD / MAY in the sense of RFC 2119.

- **G1 Capability bootstrap.** Trust roots, classification schemes, and policy tables MUST be established and locked before the runtime reads any skill content, any model output, or any external network traffic. The lock MUST be unconditional (no operator override at run time).
- **G2 Deny-by-default at every boundary.** Network egress, provider invocation, channel use, file write,

N	K	R	Clean pass	F1 detect	F2 detect	F3 detect	F4 detect
Varying corpus size N ($K = 4$, $R = 10$)							
10	4	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
50	4	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
200	4	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
Varying agent count K ($N = 50$, $R = 10$)							
50	2	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
50	4	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
50	8	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
Varying rounds R ($N = 50$, $K = 4$)							
50	4	5	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
50	4	10	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
50	4	25	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]	1.000 [0.981, 1.000]
Aggregate ($n = 5400$)			1.000 [0.999, 1.000]	1.000 [0.999, 1.000]	1.000 [0.999, 1.000]	1.000 [0.999, 1.000]	1.000 [0.999, 1.000]

Table 2. Adversarial-ensemble sweep over N corpus files, K agents, and R rounds, with $S = 200$ seeds per cell ($27 \times 200 = 5400$ total trials per scenario). The four-role ensemble (cleaner/consolidator/auditor/critic, cycled when $K > 4$) emits a mix of `fs.read` and `fs.write.irrev`; the gate's allow-all broker approves every irreversible call and audits it (§4). *Clean pass* = fraction of injection-free runs in which the biconditional $D = S$ holds (this is $1 - \text{FPR}$); F_i *detect* = fraction of runs with an injected F_i fault in which the biconditional flags the fault. Wilson 95% CIs in brackets. The criterion is mechanical within its detection envelope (Prop. 5.2); per-cell results are scale-invariant, and the aggregate-row CI is tight at $[0.999, 1.000]$.

and tool dispatch MUST require positive allow-list membership. The default policy MUST be deny.

G3 Mandatory classification. Every skill artifact, every tool input, and every tool output MUST carry a classification label drawn from a finite, declared lattice. Label-free artifacts MUST be denied at the gate.

G4 Clearance-bounded signing. Trust- root signers MUST be bound to a maximum clearance. A skill manifest declaring a clearance above its signer's maximum MUST be rejected.

G5 Hash-chained audit at gate-event granularity. Every gate event MUST produce an audit record carrying a `prevHash` link to its predecessor; concurrent appends MUST serialize deterministically. The chain MUST be verifiable by any party that can read the log.

G6 Reversible/irreversible split. Every capability MUST be statically tagged as reversible or irreversible. Irreversible capabilities MUST pass through the HITL gate of §4; reversible capabilities SHOULD use a transaction buffer with rollback.

G7 Biconditional post-hoc verification. The runtime SHOULD support the biconditional criterion of §5 as a runnable check after any agent run; the corpus delta SHOULD be reconcilable with the approved-and-executed set in the audit log without operator interpretation.

G8 Adversarial test corpus. The runtime MUST ship a corpus of pen-tests covering, at minimum, the failure modes **F1–F4** of §5 plus the eight attack families enumerated above (audit in-place edit, log injection, signature forgery, hostname-normalization bypass, post-lock trust-root mutation, ReDoS, prompt-injection role-spoofing, code injection through prototype pollution). The corpus MUST run in CI on every commit.

G9 Standard configuration profiles. The runtime MUST expose at least one strict deployment profile (analogous to the reference's `enclaved` flavor) and one development profile (analogous to `open`). Configuration MUST NOT be a free-form continuum of feature flags; deployment-class differences MUST be named and documented.

G10 No bypass switch. The runtime MUST NOT expose a build flag, environment variable, runtime API, or operator override that disables the HITL gate, the audit log, the egress guard, the classification check, or the trust-root lock. Defense in depth MUST not have a master kill-switch.

G11 Skills are untrusted by default. The runtime MUST treat every skill manifest as UNVERIFIED unless the manifest carries an explicit verification level set by the operator (or an operator-authorized attestation pipeline) at bootstrap. The UNVERIFIED default MUST subject every irreversible capability call to HITL. A signature and a clearance check MUST NOT be sufficient to elevate a skill above UNVERIFIED.

G12 Verification is bootstrap-only, skills are immutable in-session, and every mutation attempt is audited. Verification levels MUST be set before the runtime reads any external input and MUST NOT be elevated during a session. Loaded skill content MUST NOT be modifiable by the agent during the session; any agent attempt to mutate a loaded skill (file-system edit, runtime API, indirect side-effect on the on-disk artifact) MUST be intercepted as an irreversible capability call, walked through HITL, *and* written into the hash-chained audit log with the pre- and post-mutation content hashes regardless of whether the operator approves or denies. Successful mutations MUST produce a new skill artifact that is re-verified before the next session.

Attack-surface bounding. G3+G4+G5+G11 bound supply-chain attacks (unsigned skills rejected; signed-but-loaded skills sit at UNVERIFIED until an operator acts). G2+G6 bound prompt-injection through skill content (induced irreversible calls are intercepted; approval cannot be manufactured from model output). G1+G4+G12 bound self-generated and in-flight skill mutations. G5+G7 bound audit forgery. G10 bounds operator-pressure failure modes. The reduction is not to zero — the residual surface of §5.4 remains — but the surface G1–G12 close is precisely the one where incidents are post-hoc-undeniable.

8 Open problems

Three open directions. *Label composition under declassification*: the conservative join-of-labels rule (§3) is correct but blunt; provable declassification when outputs depend only on low-label inputs is a research direction. *Signer revocation in-flight*: skills loaded under a now-revoked signer should be evicted at the next gate event rather than allowed to drain; PKI mechanics (CRLs/OCSP/short lifetimes) carry over but session eviction needs more work. *Hardware-rooted trust*: the schema extends straightforwardly to TPM-sealed signer keys, secure-enclave-bound brokers, and attested HITL devices without changing the criterion.

9 Related work

Mandatory access control and information-flow. The classification component of the schema is Bell–LaPadula [2] adapted to skill artefacts. SELinux [16] demonstrated MAC at OS scope; Asbestos [8] and HiStar [29] extended decentralized information-flow control to whole systems. Our contribution is not the lattice but its lifting onto skills as deployable artefacts.

Audit + capabilities. Hash-chained logs are folklore [23]; the novelty here is linking the audit records to the biconditional check, not the chain construction. The discipline of manifest-declared capabilities and one-shot trust-root locks follows the object-capability tradition [18, 19], lifted from intra-process objects to inter-host deployable units.

Prompt-injection defenses. Model-boundary defenses — NeMo Guardrails [20], Llama Guard [11], Lakera Guard [12] — operate as input/output filters and are complementary to ours: a skill that is malicious but well-formed may pass an input filter yet be defeated by capability denial, and vice versa. Filter-only defenses are also subvertible by weak-to-strong jailbreaks [30], reinforcing the case for a runtime gate independent of the model's own classification of its inputs.

Runtime enforcement for agents. AgentSpec [26] introduces a domain-specific language for trigger/predicate/enforcement rules; the rules are user-authored, while our schema treats the skill artifact as the unit of governance with manifest-declared capabilities and a verification-level field. The two are complementary: AgentSpec rules can be installed inside our broker (§4) as the policy layer for irreversible calls, with the biconditional verifying that their world-effect matches the audit log. The verifiably-safe-tool-use proposal [7] derives enforceable specs from STPA hazards at the *tool-protocol* layer, complementary to our *skill-admission* layer and strengthening F5 of §5.4. SafeAgent [15] adds a stateful per-trajectory controller orthogonal to our per-call gate; its session invariants close the overlap-with-malicious-actor gap of §5.4.

Skill ecosystems and governance. A concurrent 2026 survey [27] proposes a four-tier (T1–T4) gate-based skill governance framework over the `SKILL.md` ecosystem, motivated by the finding that $\sim 26\%$ of community skills carry at least one vulnerability. We share the position that skills require provenance-bound, verification-gated permissioning; this paper goes further on three structural axes the survey does not commit to: *bootstrap-only immutability* of the verification level (§3.2), *audit reconciliation against world state* via the biconditional (§5), and a *no-bypass-switch* invariant (G10) closing the operator-pressure failure mode. SkillProbe [10] and the threat-taxonomy [14] are empirical-auditing complements: their findings (prompt-injection-shaped instructions, silent exfiltration, supply-chain residues) map onto our F1–F5 plus the side-channels of §5.4.

10 Conclusion

The trust schema (§3) adds the verification-level field and immutable-in-session discipline absent from current `SKILL.md` conventions; the capability-gate model (§4) keys HITL to verification level, replacing the untenable uniform-HITL default; the biconditional criterion (§5) is an audit-log-driven check on what the runtime *did*, complementary to behavioral benchmarks [1] that score what an agent *decided*; the G1–G12 guidelines (§7) abstract a portable invariant set any harness can adopt — precisely the failure modes operators discover, after the fact, in incident review.

References

[1] Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, Zico Kolter, Matt Fredrikson, Eric Winsor, Jerome Wynne, Yarin Gal, and Xander Davies. 2025. AgentHarm: A Benchmark for Measuring Harmfulness of LLM Agents. In *International Conference on Learning Representations (ICLR)*.

[2] D. Elliott Bell and Leonard J. LaPadula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report MTR-2997. The MITRE Corporation.

[3] Alex Birsan. 2021. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610. Disclosure of a supply-chain class of attacks against npm, PyPI, and RubyGems.

[4] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. 2008. A Look in the Mirror: Attacks on Package Managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. 565–574.

[5] Zhaorun Chen, Zhen Xiang, Chaowei Xiao, Dawn Song, and Bo Li. 2024. AgentPoison: Red-Teaming LLM Agents via Poisoning Memory or Knowledge Bases. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[6] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a Generalist Agent for the Web. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

[7] Aarya Doshi, Yining Hong, Congying Xu, Eunsuk Kang, Alexandros Kapravelos, and Christian Kästner. 2026. Towards Verifiably Safe Tool Use for LLM Agents. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE NIER)*. Preprint: arXiv:2601.08012.

[8] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. 17–30.

[9] Emilio Ferrara. 2024. GenAI Against Humanity: Nefarious Applications of Generative Artificial Intelligence and Large Language Models. *Journal of Computational Social Science* 7 (2024), 549–569.

[10] Zihan Guo, Zhiyu Chen, Xiaohang Nie, Jianghao Lin, Yuanjian Zhou, and Weinan Zhang. 2026. SkillProbe: Security Auditing for Emerging Agent Skill Marketplaces via Multi-Agent Collaboration. arXiv:2603.21019 [cs.CR]

[11] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, and Madian Khabsa. 2023. Llama Guard: LLM-Based Input–Output Safeguard for Human–AI Conversations. arXiv:2312.06674

[12] Lakera AI. 2024. Lakera Guard: Real-Time LLM Security. https://www.lakera.ai/.

[13] Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Li Erran Li, Ruohan Zhang, Weiyu Liu, Percy Liang, Li Fei-Fei, Jiayuan Mao, and Jiajun Wu. 2024. Embodied Agent Interface: Benchmarking LLMs for Embodied Decision Making. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[14] Zhiyuan Li, Jingzheng Wu, Xiang Ling, Xing Cui, and Tianyue Luo. 2026. Towards Secure Agent Skills: Architecture, Threat Taxonomy, and Security Analysis. arXiv:2604.02837 [cs.CR]

[15] Hailin Liu, Eugene Ilyushin, Jie Ni, and Min Zhu. 2026. SafeAgent: A Runtime Protection Architecture for Agentic Systems. arXiv:2604.17562 [cs.CR] Stateful runtime controller mediating actions around the agent loop with a context-aware decision core over persistent session state.

[16] Peter Loscocco and Stephen Smalley. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*. 29–42.

[17] Alfredo Metere. 2026. enclawed: A Configurable, Sector-Neutral Hardening Framework for Single-User AI Assistant Gateways. arXiv:2604.16838 [cs.CR] Open-source reference implementation: https://github.com/metereconsulting/enclawed.

[18] Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University.

[19] Mark S. Miller, Ka-Ping Yee, and Jonathan S. Shapiro. 2003. The Object-Capability Model: A Survey. Technical report, Hewlett-Packard Laboratories.

[20] NVIDIA. 2024. NeMo Guardrails: A Toolkit for Programmable LLM Safety. https://github.com/NVIDIA/NeMo-Guardrails.

[21] OWASP Foundation. 2025. OWASP Top 10 for Large Language Model Applications. https://genai.owasp.org/llm-top-10/. Accessed 2026.

[22] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable Key Compromise in Software Update Systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. 61–72.

[23] Bruce Schneier and John Kelsey. 1999. Secure Audit Logs to Support Computer Forensics. In *ACM Transactions on Information and System Security*, Vol. 2. 159–176.

[24] The MITRE Corporation. 2025. MITRE ATLAS: Adversarial Threat Landscape for AI Systems. https://atlas.mitre.org/. Accessed 2026.

[25] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (1984), 761–763.

[26] Haoyu Wang, Christopher M. Poskitt, and Jun Sun. 2025. AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. arXiv:2503.18666 [cs.CR]

[27] Renjun Xu and Yang Yan. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. arXiv:2602.12430 [cs.AI] Proposes a four-tier (T1–T4) gate-based (G1–G4) skill trust and lifecycle governance framework over the `SKILL.md` ecosystem.

[28] Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. 2024. An LLM-Assisted Easy-to-Trigger Backdoor Attack on Code Completion Models: Injecting Disguised Vulnerabilities Against Strong Detection. In *Proceedings of the 33rd USENIX Security Symposium*.

[29] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 263–278.

[30] Xuandong Zhao, Xianjun Yang, Tianyu Pang, Chao Du, Lei Li, Yu-Xiang Wang, and William Yang Wang. 2024. Weak-to-Strong Jailbreaking on Large Language Models. arXiv:2401.17256