Offline-Verifiable Runtime Evidence Bundles for Governed AI Agent Systems

Gustavo Venegas*, Edison Vazquez*, Danilo Naranjo†, Benjamin Gonzalez*
*Chilean Chamber of Artificial Intelligence †Ocular
gustavo@gobiernoia.cl, edison@jhedai.com, dan@ocularsolution.com, ben@nirvana-ai.com

***Abstract*—Governed AI agent runtimes emit model calls, tool decisions, provider-boundary outcomes, and policy results, but the evidence of what happened is usually a human-facing UI label, a provider log, or an informal status field that a consumer must trust without independent recomputation. We present ArgorixLang's *EvidenceBundle*, a local runtime artifact that links a compiled bytecode program, an execution trace, a security report, and a trace-event ledger digest through semantic SHA-256 digests computed over canonical JSON. A verifier recomputes each digest and re-derives the ledger digest from the trace events, then checks that the security report's LedgerSummary digest matches the bundle, establishing cross-artifact consistency without network access, provider credentials, a live runtime, or any external service. We give a lightweight formal model of the verification predicate and show it is distinct from policy approval: a bundle can verify while the security report records policy failure and a review-required state. In an existing corpus of 33 request directories, 27 are complete and pass offline verification (27/27); 6 are source-only and not assessable. The corpus contains 7,155 counted ledger events and 1,188 detailed policy violations, and every complete session records `policy_passed=false`, `review_required=true`. We hold a strict interpretation boundary: successful verification proves internal artifact consistency relative to the bundle only—not source integrity, producer authentication, absence of side effects, policy approval, certification, or production isolation.**

***Index Terms*—AI agents, runtime evidence, evidence bundles, offline verification, semantic digests, canonical JSON, agent governance, auditability, policy evidence, reproducibility.**

I. Introduction

A governed AI agent does not execute in a vacuum. A single request can issue model calls, invoke tools, cross a provider boundary, trigger policy evaluation, and produce a verdict. *Governance* means a downstream party—an auditor, an operator, a compliance reviewer, or a second agent—must later determine what the runtime actually did. The difficulty is that the evidence on offer is almost always *mediated*: a dashboard renders a green checkmark, a provider returns an opaque log line, a human writes "reviewed—OK" into a status field. Each is a summary produced by a party the consumer must already trust, and none can be independently recomputed from the artifacts themselves.

This is unsatisfactory for governed deployments for three reasons. First, a favorable *summary* can mask an unfavorable *underlying* result: a top-level "checks passed" banner says nothing about whether a detailed policy object recorded a violation. Second, evidence tied to a *live* service is unavailable precisely when it is most needed—after the fact, offline, when the provider is gone or uncooperative. Third, a mediated label conflates distinct predicates—*did the artifacts survive unchanged?* and *was the behavior approved?*—that governance requires be kept apart.

We argue that runtime evidence for governed agents must instead be (i) *inspectable after execution*, as concrete artifacts on disk; (ii) *verifiable without a live provider*, using only local recomputation; and (iii) *claim-bounded*, so that what verification establishes is never silently promoted into what it does not. ArgorixLang [1], [2] realizes this with the *EvidenceBundle*: a local artifact that records semantic SHA-256 digests of a compiled bytecode program, an execution trace, and a security report, plus a digest of the trace-event ledger, together with a relative-path artifact map. A verifier loads the referenced artifacts, recomputes every digest over canonical JSON, re-derives the ledger digest from the trace events, and checks that the report's embedded LedgerSummary digest matches the bundle—with no network, no credential, and no external service.

The central stance of this paper, and its central claim boundary, is that *artifact integrity is not policy approval*. A verified bundle proves only that the bytecode, trace, and report are mutually consistent relative to the bundle. It does not prove that the source compiled to that bytecode, who produced the bundle, that no side effects occurred outside instrumentation, or that any policy approved the behavior. We make this separation formal (Section IV) and show that the existing corpus contains cases that are *verifiable and policy-failing simultaneously* (Section VII).

Contributions.

1) A runtime evidence model for governed AI agent systems, expressed as concrete on-disk artifacts rather than mediated summaries.
2) A semantic-digest *bundle* linking bytecode, trace, security report, and a trace-event ledger digest over canonical JSON.
3) A formal *offline verification predicate* over those artifacts, with an accompanying verification algorithm requiring no live service.
4) A formal *interpretation boundary* separating EvidenceBundle verification from policy approval, stated as $\text{Verify} \not\Rightarrow \text{PolicyApproved}$.
5) An *observational evaluation* using the existing ArgorixLang runtime corpus, reporting what the evidence does and does not establish.

A scope note distinguishing this paper from the companion Agent Passport paper is given in Section XIII.

II. Background and Motivation

A. What an agent runtime produces

A governed agent runtime emits a heterogeneous record: model invocations, tool-call decisions, provider-boundary crossings (or refusals), policy evaluations, and a final verdict. ArgorixLang serializes this record into four artifact *kinds* downstream of an upstream source program: a compiled bytecode program, an ordered execution trace, a structured security report, and an evidence bundle that ties them together. The distinction between kinds matters because they play different evidential roles.

B. Logs, reports, provenance, attestation, bundles

These terms should not be used interchangeably. A *log* is an append-only stream, typically unverifiable after the fact and trusted by provenance of the emitter. A *report* is an interpretation: a structured judgment over a run. A *provenance record* asserts how an artifact was produced and, in signed forms (e.g., in-toto [3]), binds that assertion to a key. An *attestation* is a signed statement by an identified party. An *evidence bundle*, as used here, is none of these: it is an *unsigned, local, recomputable cross-artifact consistency record*. It states that specific artifacts hash to given values and agree with one another, and nothing about who produced them.

C. Why digest verification helps but is not trust

A semantic digest detects *change relative to a recorded value*. If a verifier recomputes $H(bc)$ and it matches `B.bytecode_digest`, the bytecode is the one the bundle recorded. This is genuinely useful: it makes silent tampering of a self-contained bundle detectable, reproducibly and offline. But it establishes nothing about *origin*—an adversary who controls artifacts and bundle together can produce a self-consistent bundle with attacker-chosen content. Digest verification is a consistency primitive, not a trust primitive.

D. Why policy evidence must preserve failure

An evidence system is only useful if it faithfully preserves negative results. The temptation in agent UIs is to collapse a run into a single optimistic status. ArgorixLang's security report instead retains detailed policy objects—per-block results, violations, and a `review_required` flag—so that a favorable coarse label cannot overwrite a detailed failure. Preserving the negative is the point: an auditor needs to see the violation, not a smoothed-over banner.

E. Relation to adjacent systems

in-toto [3] provides *signed* supply-chain provenance; EvidenceBundles are local and unsigned. OPA [4] is an external policy *decision* engine; ArgorixLang records policy outcomes inside runtime artifacts rather than externalizing the decision. WASI [5] provides capability isolation at the host; bundle verification proves nothing about host sandboxing. MCP and A2A [6], [7] standardize agent interaction surfaces; bundles record local runtime boundaries, not live interoperability. The NIST AI RMF [8] and the OWASP Top 10 for LLM Applications [9] frame risk and inform our threat discussion, but neither is a certification we claim to satisfy.

III. Evidence Artifact Model

We define each artifact and fix its verification scope; Table I summarizes.

(1) Source (`session.argx`) is the upstream compiler input. It is *outside* bundle verification: the bundle carries no source path and no source digest, so verification begins at compiled bytecode and never asserts that the source produced it.

(2) Bytecode (`session.argbc.json`) is the compiled program loaded by the VM, verified by `bytecode_digest`; cross-checks on `language`, `module`, and `bytecode_version` guard against gross substitution.

(3) Trace (`session.trace.json`) is an ordered ledger of runtime and policy events—agent and policy declarations, message scheduling/delivery, handler execution, policy evaluations, and VM lifecycle markers—verified by `trace_digest` and the source of the ledger digest.

(4) SecurityReport (`session.security.json`) is a structured interpretation of runtime state: execution status, policy outcomes (`passed`, `violations`, `review_required`), provider boundary, call counts, and a verdict. It is verified by `report_digest` and contains a *LedgerSummary* whose `ledger_digest` must equal the bundle's.

(5) EvidenceBundle (`session.evidence.json`) holds the artifact map plus four semantic digests (`bytecode_digest`, `trace_digest`, `report_digest`, `ledger_digest`). The artifact map (`bytecode_path`, `trace_path`, `security_report_path`) holds *relative* paths confined to the bundle's portable subtree; there is no source path and no standalone ledger path.

(6) Ledger digest is computed as $H(\text{events}(tr))$—the canonical digest of the trace's event list, not a separate file. It is checked against the bundle's `ledger_digest` *and* the report's embedded LedgerSummary digest, binding the trace's events to the report's claims about them.

Scope, stated plainly. The bundle's verifiable region *starts* at bytecode, trace, and report. It does not verify that `session.argx` produced the bytecode; it does not authenticate who produced the bundle; and it does not prove security, compliance, or absence of effects.

IV. Formal Model

We give a lightweight formalization. It is an *interpretation boundary*, not a cryptographic security proof.

Definitions. Let B be an EvidenceBundle; bc, tr, sr the bytecode, trace, and security-report artifacts; $\text{events}(tr)$ the ordered trace event list; $\text{LS}(sr)$ the LedgerSummary digest stored in the report; $C(x)$ the canonical JSON serialization of x; and $H(x) = \text{SHA-256}(C(x))$.

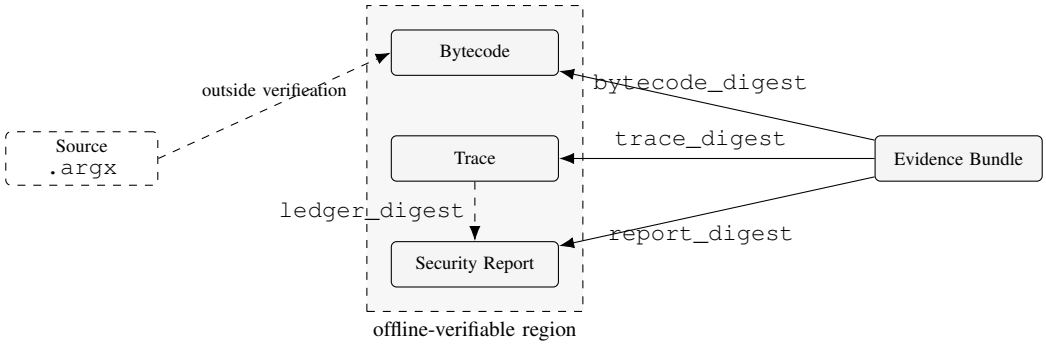

Fig. 1. Evidence artifact graph. The source lies outside the verification boundary (dashed edge). Bytecode, trace, and security report form the offline-verifiable region; the Evidence Bundle pins each via a semantic digest, and the ledger digest—derived from `trace.events`—is matched against the report's LedgerSummary.

TABLE I
RUNTIME EVIDENCE ARTIFACTS.

Artifact	Path	Role	Verified by	Out-of-scope
Source	`.argx`	Compiler input	— (not in bundle)	Source→ bytecode link
Bytecode	`.argbc` `.json`	Compiled program	`bytecode_digest`	Source provenance
Trace	`.trace` `.json`	Event ledger	`trace_digest`, `ledger_digest`	Uninstrumented events
Sec. Report	`.security` `.json`	Runtime/ policy interp.	`report_digest`; LS match	Truth of metadata
Evidence Bundle	`.evidence` `.json`	Map + digests	cross-checked	Producer identity
Ledger digest	(from `trace.events`)	Binds events to report	$H(\text{events}(tr))$	Standalone ledger

Semantic-digest equations.

$$B.\text{bytecode\_digest} = H(bc) \tag{1}$$
$$B.\text{trace\_digest} = H(tr) \tag{2}$$
$$B.\text{report\_digest} = H(sr) \tag{3}$$
$$B.\text{ledger\_digest} = H(\text{events}(tr)) \tag{4}$$
$$\text{LS}(sr) = B.\text{ledger\_digest} \tag{5}$$

Verification predicate.

$$\begin{aligned}\text{Verify}(B, bc, tr, sr) := {} & B.\text{bytecode\_digest} = H(bc) \\ & \wedge\ B.\text{trace\_digest} = H(tr) \\ & \wedge\ B.\text{report\_digest} = H(sr) \\ & \wedge\ B.\text{ledger\_digest} = H(\text{events}(tr)) \\ & \wedge\ \text{LS}(sr) = B.\text{ledger\_digest}\end{aligned}$$

Policy approval (a separate predicate).

$$\begin{aligned}\text{PolicyApproved}(sr) := {} & sr.\text{policy\_passed} = \text{true} \\ & \wedge\ sr.\text{review\_required} = \text{false} \\ & \wedge\ |sr.\text{violations}| = 0\end{aligned}$$

TABLE II
FORMAL PREDICATES.

Predicate	Definition	Interpretation
$H(x)$	SHA-256$(C(x))$	Semantic digest over canonical JSON
Verify	five digest/LS equalities hold	Internal consistency relative to B
PolicyApproved	passed $\wedge$ $\neg$review $\wedge$ no violations	Approval as recorded by the report
Verify $\not\Rightarrow$ PolicyApproved	verification $\neq$ approval	Integrity boundary (central)
$\exists$: Verify $\wedge$ $\neg$PolicyApproved	witnessed by corpus	27/27 verify while failing

Interpretation boundary (central result). Verification does not imply approval:

$$\text{Verify}(B, bc, tr, sr) \not\Rightarrow \text{PolicyApproved}(sr).$$

The predicates read disjoint fields: Verify reads digests and the LedgerSummary; PolicyApproved reads `policy_passed`, `review_required`, and `violations`. Nothing in Verify constrains the latter, so a faithfully recorded policy failure passes through verification unchanged. The observed corpus witnesses the satisfiable conjunction

$$\exists\, sr :\ \text{Verify}(B, bc, tr, sr)\ \wedge\ \neg\text{PolicyApproved}(sr),$$

because complete bundles verify (27/27) while their reports record `policy_passed=false` and `review_required=true`. We do *not* present Verify as proof of authenticity, source integrity, or external truth; it is a statement of internal cross-artifact consistency relative to B. Table II restates these predicates.

V. OFFLINE VERIFICATION PROCEDURE

The verifier is a pure function of local files. It loads the bundle; resolves `bytecode_path`, `trace_path`, and `security_report_path` relative to the bundle directory (rejecting absolute paths and paths escaping the portable subtree); loads the three artifacts; canonicalizes each to JSON; recomputes `bytecode_digest`, `trace_digest`,

```
Algorithm 1: Offline EvidenceBundle Verification
Input : bundle file B (referencing bc, tr, sr)
Output: verified in {true, false}

 1. Load B.
 2. Resolve B.artifact_map paths
    (relative, inside portable tree).
 3. Load bc, tr, sr.
 4. d_bc     <- H(bc)
 5. d_tr     <- H(tr)
 6. d_sr     <- H(sr)
 7. d_ledger <- H(events(tr))
 8. check d_bc     = B.bytecode_digest
 9. check d_tr     = B.trace_digest
10. check d_sr     = B.report_digest
11. check d_ledger = B.ledger_digest
12. check LS(sr)   = B.ledger_digest
13. return verified iff checks (8)-(12) all hold
```

Fig. 2. Offline verification recomputes every digest from local artifacts and compares against the bundle; no live service participates.

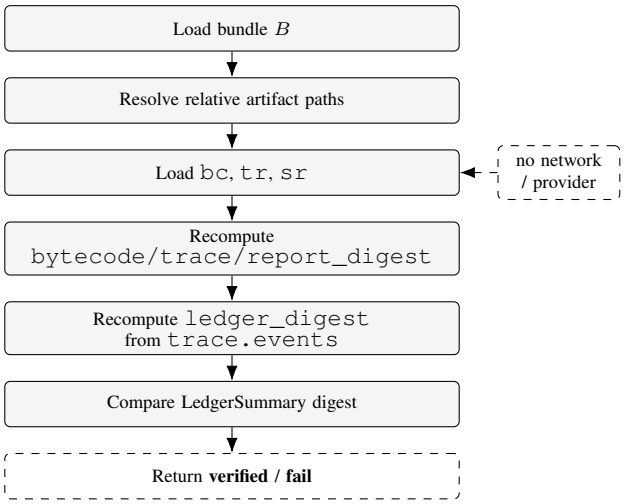

Fig. 3. Offline verification sequence. Every step reads local files only; no provider or network participates.

and `report_digest`; recomputes `ledger_digest` from `trace.events`; compares the report's LedgerSummary digest against the bundle's; and returns *verified* only if all checks pass. It requires *no network access, no provider credential, no live runtime, and no external service*. Auxiliary checks (version-field agreement across artifacts, digest well-formedness, path confinement) accompany the core comparisons.

VI. Integrity Is Not Approval

This section is the conceptual core. Six statements bound what a verified bundle means. *(1) Consistency, not external truth:* a valid bundle proves the artifacts agree relative to the bundle, not that any claim they make is externally true. *(2) Change detection, not origin:* a matching digest detects change relative to a recorded artifact, not who produced it; an unsigned bundle has no producer binding. *(3) Instrumented events, not all events:* a verified trace records instrumented events faithfully but cannot prove that no effect occurred outside the instrumentation boundary. *(4) Faithful negatives:* a verified report can preserve a negative policy result without contradiction; verifying the report is orthogonal to the report's verdict. *(5) Detail dominates summary:* a favorable top-level status must not override a detailed policy failure—component status and `review_required` dominate interpretation. *(6) The central statement: a bundle can be offline-verifiable and still contain policy failure, a review-required status, unknown-rule violations, or otherwise non-certified behavior.* Table III tabulates the distinction across specific claims.

TABLE III
Integrity vs. approval distinction.

Claim	Supported?	Reason
Artifacts unchanged relative to bundle	Yes	Digests recompute and match
Trace events match report LedgerSummary	Yes	$H(\text{events}(tr)) = \text{LS}(sr)$
Source produced this bytecode	No	No source path/digest in bundle
A known party produced the bundle	No	Unsigned; no producer binding
No side effects outside instrumentation	No	Trace sees instrumented events only
Policy approved the behavior	No	Separate predicate; corpus fails
Legal/compliance certification	No	No certification mechanism
Host/production isolation	No	No sandbox measurement

VII. Observational Evaluation

We reuse the existing ArgorixLang runtime corpus [1] and add no new experiment.

A. Corpus

Of **33** request directories, **27** are *complete*—each containing `session.argx`, `session.argbc.json`, `session.trace.json`, `session.security.json`, and `session.evidence.json`—and **6** are *source-only* and not assessable for trace/security/evidence fields. The corpus contains **7,155** counted ledger events and **1,188** detailed policy violations across the complete sessions.

B. Offline verification

All **27/27** complete bundles pass offline verification: the three semantic digests reproduce, the ledger digest recomputes from `trace.events`, and the LedgerSummary digest matches the bundle. No bundle digests or verifies `session.argx`.

C. Policy results

Every one of the 27 detailed policy reports sets `policy_passed=false` and `review_required=true`, with **44** unknown-rule violations per complete session. Verification success therefore coexists with policy non-approval—the witnessed conjunction of Section IV. Table IV summarizes.

TABLE IV
OBSERVATIONAL CORPUS RESULTS.

Measure	Observed	Interpretation
Request directories	33	Full corpus
Complete directories	27	Assessable for fields
Source-only directories	6	Explicitly incomplete
Bundles passing offline verif.	27/27	Internal consistency only
Counted ledger events	7,155	Structured event emission
Detailed policy violations	1,188	Negatives preserved
Unknown-rule violations / session	44	Per-session failure detail
Reports `policy_passed=false`	27/27	Verify $\not\Rightarrow$ Approved
Reports `review_required=true`	27/27	Review state preserved

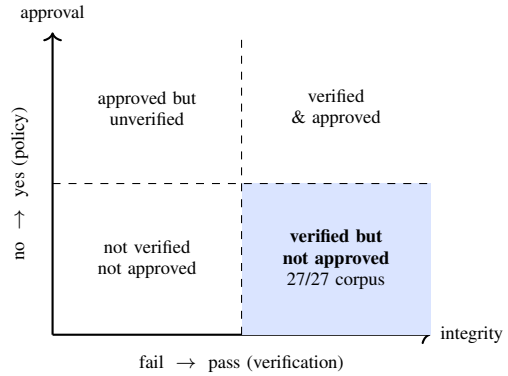

Fig. 4. Two-axis interpretation. The corpus populates the verified-but-not-approved quadrant: every complete session is offline-verifiable yet records policy failure—the visual statement of Verify $\not\Rightarrow$ PolicyApproved.

D. Interpretation

The evidence pipeline is reproducible across complete sessions, and offline verification works for every complete bundle. Missing artifacts surface as explicit incomplete observations, not silent passes—a source-only directory is *unavailable*, never *verified*. Verification success does not imply policy success. The repeated structure supports pipeline reproducibility but limits any claim of behavioral diversity.

VIII. Threat Model and Security Analysis

We analyze threats as attack paths across trust boundaries, not as proof that a control defeats every adversary.

Threats. *Artifact tampering* (altering bytecode, trace, or report after the fact); *bundle replacement* (swapping the bundle); *report/status confusion* (reading a coarse status and missing a detailed failure); *missing source integrity* (no link from bytecode back to `session.argx`); *producer impersonation*; *unsigned bundle replay*; *runtime side effects outside instrumentation*; *consumer overinterpretation* (treating consistency as certification); and *malicious but self-consistent bundle generation* (an adversary controlling all artifacts emits a coherent bundle with chosen content).

TABLE V
EVIDENCEBUNDLE DIGEST FIELDS.

Digest field	Computed from	Purpose	Does not prove
`bytecode_digest`	$H(bc)$	Pin compiled program	Source produced bytecode
`trace_digest`	$H(tr)$	Pin trace artifact	Instrumentation complete
`report_digest`	$H(sr)$	Pin security report	Truth of verdict
`ledger_digest`	$H(\text{events}(tr))$	Bind events to LedgerSummary	Events outside trace

Bounded mitigations. Semantic-digest linkage detects post-hoc change to a recorded artifact; the ledger digest derived from trace events ties events to the report's LedgerSummary; cross-artifact consistency checks (version-field agreement, path confinement) catch mismatched substitution; explicit incomplete-session handling prevents missing artifacts from masquerading as passes; component-level policy status and `review_required` are preserved; and the procedure is offline-reproducible.

Remaining gaps. There are *no* signatures, *no* trusted timestamp, *no* transparency log, *no* source digest, *no* hardware root, *no* independent witness, *no* OS-level sandbox proof, *no* producer authentication, and *no* revocation or witness model. Tampering and replacement are defeated only when the adversary cannot also rewrite the bundle; against an adversary who controls artifact and bundle together, self-consistency is necessary but not sufficient for trust. The strongest supported statement is conditional: within the recorded artifact set, complete bundles are offline-consistent and policy negatives are preserved. Trust outside that set is unmeasured.

IX. Related Work

We compare carefully and claim no equivalence. in-toto [3] provides signed supply-chain provenance binding steps to keys; ArgorixLang EvidenceBundles are local, unsigned runtime-evidence records with no producer binding. OPA [4] is an external policy decision engine evaluating Rego; ArgorixLang records policy and evidence outcomes in runtime artifacts rather than externalizing the decision. WASI [5] provides capability-based host isolation; evidence verification proves nothing about host sandboxing. MCP and A2A [6], [7] standardize agent interaction and interoperability; EvidenceBundles record local runtime boundaries, not live interaction. The NIST AI RMF [8] and the OWASP Top 10 for LLM Applications [9] provide risk framing that motivates the threat model; we map to neither as a certification. The bundle sits in the lineage of provenance and reproducibility systems but trades signatures and trust anchors for offline recomputability and a strict claim boundary. The ArgorixLang implementation is in Rust [10].

X. Limitations

We state limitations strictly. There is *no* source integrity (no source digest, no bytecode-to-source link); *no* producer

authentication; *no* signatures; *no* trusted timestamp; *no* transparency log; *no* blockchain immutability; *no* post-quantum security; *no* policy-approval guarantee; *no* legal or compliance certification; *no* host-level containment proof; *no* controlled adversarial test; *no* prompt-level qualitative analysis (prompt content is unavailable in the corpus); *no* proof of semantic correctness of every trace event; and *no* proof of absence of side effects outside the instrumented runtime. The complete sessions are structurally uniform, supporting reproducibility but limiting generalization across prompts, policies, and jurisdictions.

XI. Future Work

Natural extensions, none claimed as present: source-digest inclusion so verification extends to `session.argx`; signed EvidenceBundles for producer authentication and non-repudiation; a transparency log for independent witnessing; trusted timestamps; producer identity binding; an independent witness service; policy-decision canonicalization into a single tri-state decision from typed component results; differential verification across runtime versions; host-level sandbox measurement; a controlled adversarial corpus; a public reproducibility package; and an independent external verifier implementation.

XII. Conclusion

Offline-verifiable EvidenceBundles provide a reproducible integrity layer for governed AI agent runtimes. They link bytecode, trace, security report, and a trace-event ledger digest through semantic digests over canonical JSON, and a verifier re-establishes that linkage with no live service. Their value must be read narrowly: artifact consistency, trace/report linkage, faithful policy-result preservation, and offline auditability—not security certification, policy approval, or external truth. The corpus shows the pipeline is reproducible (27/27 complete bundles verify) and, decisively, that verification and approval are distinct predicates: every complete session verifies while recording policy failure and a review-required state.

XIII. Scope

This paper differs from the companion Agent Passport paper by focusing on the runtime evidence and offline verification mechanism rather than sovereign metadata. The Agent Passport paper concerns *what* metadata—country, jurisdiction, residency, policy bindings, local agent name—is compiled and preserved; this paper concerns *how* bytecode, trace, report, and ledger evidence are linked through semantic digests and verified offline, and where the resulting integrity claim stops. All empirical values are reused from the original ArgorixLang corpus [1], and no new experiment, deployment, or security guarantee is introduced.

References

[1] G. Venegas, E. Vazquez, D. Naranjo, and B. Gonzalez, "ArgorixLang: Compiled governance, sovereign agent metadata, and offline-verifiable runtime evidence," Zenodo preprint, 2026, original ArgorixLang preprint; source material for this derived paper. [Online]. Available: https://doi.org/10.5281/zenodo.21014780

[2] Argorix Labs, "ArgorixLang," 2026, secure, verifiable programming language and runtime for AI-agent systems. [Online]. Available: https://github.com/argorixlabs/argorixlang

[3] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1393–1410. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[4] Open Policy Agent Project, "Open policy agent (OPA)," Official project documentation, 2026, general-purpose policy engine and Rego policy-as-code language. [Online]. Available: https://www.openpolicyagent.org/docs

[5] WASI Subgroup, "Introduction," Official standards portal, 2026, standards-track APIs and capability-based sandbox model. [Online]. Available: https://wasi.dev/

[6] Model Context Protocol Contributors, "Model context protocol specification, version 2025-06-18," Protocol specification, 2025. [Online]. Available: https://modelcontextprotocol.io/specification/2025-06-18

[7] A2A Protocol Working Group, "Agent2agent (A2A) protocol specification," Protocol specification, 2026. [Online]. Available: https://a2a-protocol.org/v1.0.0/specification/

[8] E. Tabassi, "Artificial intelligence risk management framework (AI RMF 1.0)," National Institute of Standards and Technology, NIST AI 100-1, Jan. 2023. [Online]. Available: https://doi.org/10.6028/NIST.AI.100-1

[9] OWASP Foundation, "OWASP top 10 for large language model applications," Security guidance, 2025. [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[10] Rust Project, "Rust programming language," Official project website, 2026. [Online]. Available: https://rust-lang.org/