

Merkle Roots for the Validated Skill Graph

*Production Merkle Epoch Roots, Chronicle-Gated Memory,
Selective Disclosure, ENS-Gated Writes, and Root-Verified Capability*

Vincent Boucher

President, QUEBEC.AI & MONTREAL.AI

Research Edition v4.0 / GoalOS V114 | Publication-safe, claim-bounded

Core Thesis

Merkle roots are one of the most useful primitives for the Validated Skill Graph because they let GoalOS commit to large, private, evolving sets of skills, evidence, ProofBundles, policies, attestations, encrypted-capsule references, and graph state with a compact, auditable root. A future mission may trust only what is both Chronicle-admitted and root-verifiable.

AI creates output. GoalOS creates proof. Merkle roots make proof state compact, inspectable, and difficult to silently rewrite.

Contents

1	One-Page Protocol Standard	2
2	Introduction	3
3	Contributions	4
4	System Positioning	4
5	Normative Object Semantics	5
6	Merkle Commitments: Minimal Formalism	6
6.1	Domain-Separated Leaves	6
6.2	Sorted-Pair Merkle Tree	6
6.3	Merkle Forest	7
7	Why Merkle Roots Are Native to the VSG	8
8	Production Object Model	8
8.1	Skill Leaf	9
8.2	Evidence and ProofBundle Leaves	9
9	Algorithms	9
9.1	Canonical Leaf Hashing	9
9.2	Root Construction	10
9.3	Inclusion Verification	10
9.4	Graph-Root Transition	10
10	Chronicle-Gated Root Admission	11
11	Inclusion, Non-Inclusion, and Root Review	11
12	ENS-Gated Direct AGI Agent Authorization	12
13	Public-Private Proof Boundary	12
14	Zero-Knowledge Compatibility	12
15	Registry Surface	13
16	Threat Model	13
17	Evaluation Program	14
18	Production Conformance Levels	15
19	Reference Architecture	15
19.1	Required Artifacts	15
20	Reviewer Workflow	16
21	Implementation Blueprint	16
22	V107 Reference Implementation Surface	17
23	Claim Boundary	18
24	Conclusion	18
	Appendix A: Canonical Merkle Packet Example	18
	Appendix B: Acceptance Tests	19
	Appendix C: Glossary	19
	Appendix D: MVS-001 Acceptance Checklist	20
	Appendix E: Minimal Research Evaluation Table	21

Abstract

The Validated Skill Graph is a proof-carrying memory substrate for autonomous AI work: candidate output becomes reusable capability only after Proof Debt is retired, AGI Jobs return evidence, an Evidence Docket aggregates proof, and Chronicle admits a scoped skill. This paper argues that Merkle roots are the natural commitment primitive for that graph. A Merkle-native Validated Skill Graph represents skills, graph edges, Evidence Dockets, ProofBundles, policies, attestations, revocations, supersessions, encrypted capsule references, and metadata as domain-separated leaves. It derives typed subroots and binds them into a single graph epoch root. The root can be committed on-chain while private intelligence remains encrypted, off-chain, or zero-knowledge-proven. The result is a compact, tamper-evident, inclusion-verifiable substrate for reusable capability. We formalize canonicalization, domain separation, root construction, inclusion proofs, root continuity, Chronicle gating, direct-agent authorization, privacy boundaries, zero-knowledge compatibility, threat models, conformance levels, and evaluation tests. The central claim is architectural and falsifiable: durable capability memory should not be a transcript, prompt library, or mutable database; it should be a proof-gated graph whose admitted state is committed by cryptographic roots and whose private contents remain protected.

Executive Compression

Problem. AI systems produce abundant output, but institutions need evidence-backed memory. **Solution.** Chronicle admits validated skills; Merkle roots commit the admitted state. **Principle.** Store roots, proofs, ciphertext commitments, and controlled pointers - not plaintext private intelligence. **Outcome.** Future missions improve only from skills that are Chronicle-admitted, scope-compatible, and root-verifiable.

1 One-Page Protocol Standard

The Four-Layer Law

State is not authority. A Merkle root proves that an exact graph state existed. **Proof is not privacy leakage.** The public layer may store roots, hashes, attestations, and pointers while private intelligence remains encrypted or off-chain. **Inclusion is not truth.** Chronicle, Evidence Dockets, ProofBundles, validators, proof levels, and rollback conditions decide whether an included object earns reuse authority. **Memory is not a transcript.** Only proof-admitted capability may influence future missions.

Layer	Object	What it proves	What it does not prove
Capability	Validated Skill	A scoped method has passed a policy gate for a defined use envelope.	Universal correctness or unrestricted reuse.
Evidence	Evidence Docket / ProofBundle	The claim has support, replay material, validator route, and boundary notes.	Legal certification or production authorization.
Commitment	Merkle root / graphEpochRoot	The exact canonical graph state and selected leaves are tamper-evidently committed.	Truth, safety, legality, or quality by itself.
Authority	Chronicle Gate / policy	The capability may influence future missions under recorded scope.	Immunity from drift, deployment risk, or liability.

Table 1: The Merkle root is a commitment layer; Chronicle is the authority layer. Keeping the two separate prevents hash inclusion from becoming accidental institutional trust.

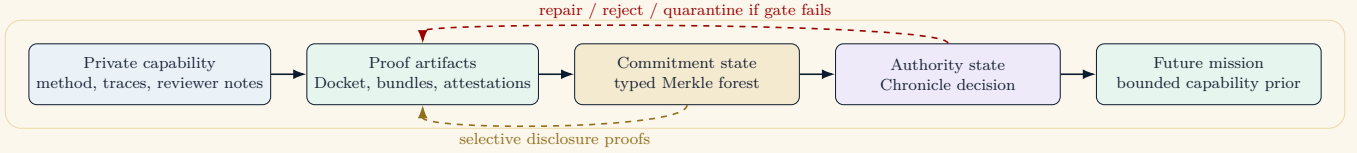


Figure 1: State, proof, commitment, and authority are separated. Merkle roots make the graph auditable; Chronicle decides what earns memory.

2 Introduction

Modern AI systems can generate text, code, plans, traces, claims, and tool actions at extraordinary scale. The bottleneck is no longer generation alone; it is deciding what deserves memory, reuse, settlement, propagation, or future-mission influence. GoalOS addresses that bottleneck with a proof-to-capability loop:

Objective → Proof Mission → AGI Jobs → Evidence → Chronicle
 → Validated Skill → On-chain Graph Root → Future Mission Improvement.

The loop treats unsupported claims as Proof Debt; Proof Debt becomes AGI Jobs; AGI Jobs return ProofBundles; ProofBundles update an Evidence Docket; Chronicle decides what may become durable memory; admitted results become Validated Skills; and the Validated Skill Graph provides future missions with scoped reusable capability.

This paper focuses on one primitive inside that architecture: the Merkle root. A Merkle root compresses a set of leaves into a single digest while preserving efficient membership proofs and tamper detection. For a Validated Skill Graph, this primitive is unusually valuable because the graph is large, dynamic, partially private, and audit-sensitive. A reviewer should be able to verify that a skill, evidence item, ProofBundle, validator attestation, policy object, revocation, or supersession was included in a committed graph state without requiring the entire private graph to be exposed.

Operating Law

A skill may influence a future mission only if it is admitted by Chronicle and bound to an auditable graph-root state. A root is not the skill. It is the tamper-evident public commitment to the skill graph state.

3 Contributions

This paper makes twelve contributions.

1. **Merkle-native VSG model.** A formal representation of the Validated Skill Graph as a forest of typed Merkle roots.
2. **Domain-separated leaf hashing.** A deterministic scheme for skill, edge, evidence, ProofBundle, policy, attestation, revocation, supersession, metadata, and graph-root leaves.
3. **Graph-epoch composition.** A method for binding multiple typed roots into one graph epoch root with chain, registry, agent, epoch, policy, snapshot, and previous-root context.
4. **Chronicle-root law.** A rule that only Chronicle-admitted skills may be committed as future-mission-influencing capability.
5. **Selective disclosure and review paths.** Reviewer workflows for verifying that a claim, skill, or evidence item is or is not part of a committed state.
6. **Privacy-preserving on-chain boundary.** A root-and-proof pattern that supports encrypted capsules and zero-knowledge receipts while avoiding plaintext private intelligence on-chain.
7. **Direct-agent authorization.** An ENS-style direct `*.agent.agi.eth` authorization model for graph-root writers.
8. **Lifecycle semantics.** Root continuity, supersession, revocation, retirement, and stale-root protections.
9. **Threat and falsification model.** Tests for tampering, equivocation, unauthorized writes, replay, stale roots, memory contamination, and serialization drift.
10. **Production artifact standard.** JSON schemas, algorithms, acceptance tests, conformance levels, and UI requirements for implementing Merkle roots inside GoalOS.
11. **V107-compatible registry surface.** Contract-level methods for `computeLeaf`, `verifyEpochLeaf`, `commitGraphEpoch`, expected-root matching, epoch-number enforcement, and previous-root continuity.
12. **Reviewer-centered verification UX.** A practical review path that shows non-technical operators what a Merkle proof proves, what remains unproven, and when a new proof mission is required.

4 System Positioning

The proof-loop doctrine can be summarized as follows: start with the objective, not the agent; keep the workbench empty until Proof Debt exists; generate only custom AGI Jobs; make Evidence Dockets inspectable; use Chronicle as the memory firewall; and store proof roots rather than plaintext private intelligence. This is the operating rule that turns autonomous work into proof and proof into reusable capability.

A Validated Skill is not a note. It is a governed capability object. It has scope, method, evidence, tests, validators, risk boundaries, policy, version, rollback conditions, and utility. A Validated Skill Graph stores those skills and the policy-scoped relationships between them: reuse, compose, teach, package, audit, retire, supersede, and future-prior influence.

The graph has three distinct layers:

1. **Private capability layer:** methods, reviewer notes, traces, sensitive data, internal tooling details, and private evidence.
2. **Proof artifact layer:** ProofBundles, Evidence Docket entries, replay manifests, validator attestations, and Chronicle decisions.
3. **Commitment layer:** Merkle roots, content hashes, ZK proof hashes, policy hashes, direct-agent identity, and append-only root history.

Merkle roots are the bridge between the proof artifact layer and the commitment layer.

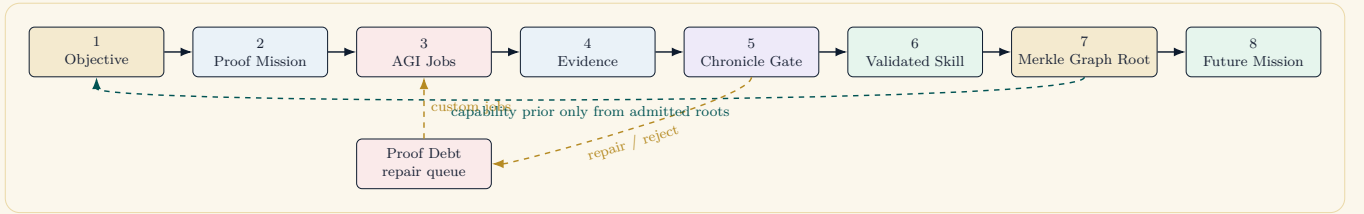


Figure 2: Merkle-native proof loop. A future mission receives capability prior only after evidence passes Chronicle and the admitted skill is bound to a Merkle graph-root state.

5 Normative Object Semantics

A Merkle-native Validated Skill Graph needs a small vocabulary so implementations do not confuse storage with governance.

Term	Definition	Required invariant
Leaf	Domain-separated hash of one canonical object.	The leaf domain and schema hash must be explicit.
Class root	Merkle root over one object class, such as skills, edges, docket, bundles, attestations, policies, revocations, supersessions, or metadata.	Cross-class replay must be impossible under normal verification.
Graph epoch root	Hash that binds all class roots plus chain, registry, agent node, epoch, previous root, policy hash, schema hash, snapshot hash, and proof level.	A single bit change in any bound field must change the epoch root.
Chronicle decision	Policy decision that admits, repairs, rejects, quarantines, supersedes, or retires a skill.	Future influence requires an admit decision and a matching Merkle inclusion proof.
Root packet	Public-safe bundle containing roots, policy identifiers, proof level, snapshot URI hash, and inclusion proof indexes.	It must be reviewable without exposing private capsule contents.

Table 2: Normative semantics for a Merkle-native Validated Skill Graph.

Protocol invariant

A graph epoch is valid only if it is canonicalized, domain-separated, proof-level-bound, policy-bound, previous-root-bound, and authorized by the direct AGI Agent that owns the relevant `label.agent.agi.eth` name.

6 Merkle Commitments: Minimal Formalism

Let H be a collision-resistant hash function, instantiated in implementation as `keccak256` or another protocol-approved hash. Let $\text{canon}(x)$ be deterministic canonicalization over JSON or typed data: stable key ordering, normalized strings, explicit schema version, explicit domain, and no nondeterministic fields unless explicitly included as data.

6.1 Domain-Separated Leaves

Each leaf is domain-separated:

$$\text{Leaf}_D(x) = H(\text{GoalOS} \parallel D \parallel : \parallel \text{schemaHash}(D) \parallel \text{canon}(x)). \quad (1)$$

The domain D prevents an evidence object from being replayed as a skill object or a policy object from being replayed as an attestation.

Domain	Leaf material	Purpose
GoalOSSkillV2	Validated Skill Passport, scope, proof level, version, utility, rollback.	Commit reusable capability nodes.
GoalOSEdgeV2	Source skill, target skill, edge type, edge policy, utility weight.	Commit graph topology and allowed operations.
GoalOSEvidenceV2	Evidence Docket claims, source refs, replay manifests, risk records.	Prove evidence inclusion without exposing all evidence.
GoalOSProofBundleV2	AGI Job outputs, worker signatures, validator verdicts, artifact hashes.	Bind job returns to proof state.
GoalOSPolicyV2	Proof-level policy, Chronicle thresholds, claim boundaries, allowed uses.	Bind admission to a specific governance rule.
GoalOSAttestationV2	Reviewer or AGI Node verdicts, signatures, quorum records.	Bind validation coverage and accountability.
GoalOSRevocationV2	Revocation reason, revoker, timestamp, affected root.	Preserve loss of future influence without deleting history.
GoalOSSupersessionV2	Old skill, new skill, reason, proof link, effective epoch.	Link improved or corrected capability to prior state.
GoalOSMetadataV2	Snapshot URI hash, schema hash, tree policy, implementation version.	Make review and replay unambiguous.

Table 3: Domain separation prevents cross-type replay and makes reviewer interpretation unambiguous.

6.2 Sorted-Pair Merkle Tree

Given a leaf list $L = [\ell_1, \dots, \ell_n]$, sort leaves lexicographically by digest unless a policy requires order-preserving construction. For each parent level:

$$\text{Parent}(a, b) = H(\text{GoalOSMerkleNodeV2} \parallel \min(a, b) \parallel \max(a, b)). \quad (2)$$

If a level has an odd number of nodes, the final node is duplicated or promoted according to the tree policy. The policy must be explicit in `policyRoot`; silent tree-policy changes are not allowed.

Implementation Rule

The tree policy is part of the commitment. A proof generated under sorted-pair duplication is invalid under order-preserving promotion unless the policy explicitly says otherwise. The canonicalization standard, hash suite, empty-root rule, leaf domains, and odd-node rule must be fixed by policy.

6.3 Merkle Forest

The Validated Skill Graph is best represented as a forest of typed roots rather than one undifferentiated tree:

$$\text{skillNodeRoot}_t = \text{Root}(\{\text{Leaf}_{\text{Skill}}(s_i)\}), \quad (3)$$

$$\text{skillEdgeRoot}_t = \text{Root}(\{\text{Leaf}_{\text{Edge}}(e_i)\}), \quad (4)$$

$$\text{evidenceDocketRoot}_t = \text{Root}(\{\text{Leaf}_{\text{Evidence}}(d_i)\}), \quad (5)$$

$$\text{proofBundleRoot}_t = \text{Root}(\{\text{Leaf}_{\text{Proof Bundle}}(p_i)\}), \quad (6)$$

$$\text{policyRoot}_t = \text{Root}(\{\text{Leaf}_{\text{Policy}}(\pi_i)\}), \quad (7)$$

$$\text{attestationRoot}_t = \text{Root}(\{\text{Leaf}_{\text{Attestation}}(a_i)\}). \quad (8)$$

The final graph epoch root binds all typed roots into a single state commitment:

$$\text{graphEpochRoot}_t = H(\text{GoalOSGraphEpochRootV2} \parallel \text{chainId} \parallel \text{registry} \parallel \text{agentNode} \quad (9)$$

$$\parallel \text{epoch}_t \parallel \text{graphEpochRoot}_{t-1} \parallel \text{skillNodeRoot}_t \parallel \text{skillEdgeRoot}_t \quad (10)$$

$$\parallel \text{evidenceDocketRoot}_t \parallel \text{proofBundleRoot}_t \parallel \text{attestationRoot}_t \quad (11)$$

$$\parallel \text{policyRoot}_t \parallel \text{revocationRoot}_t \parallel \text{supersessionRoot}_t \quad (12)$$

$$\parallel \text{metadataRoot}_t \parallel \text{snapshotURIHash}_t \quad (13)$$

$$\parallel \text{schemaHash}_t \parallel \text{policyHash}_t \parallel \text{proofLevel}_t). \quad (14)$$

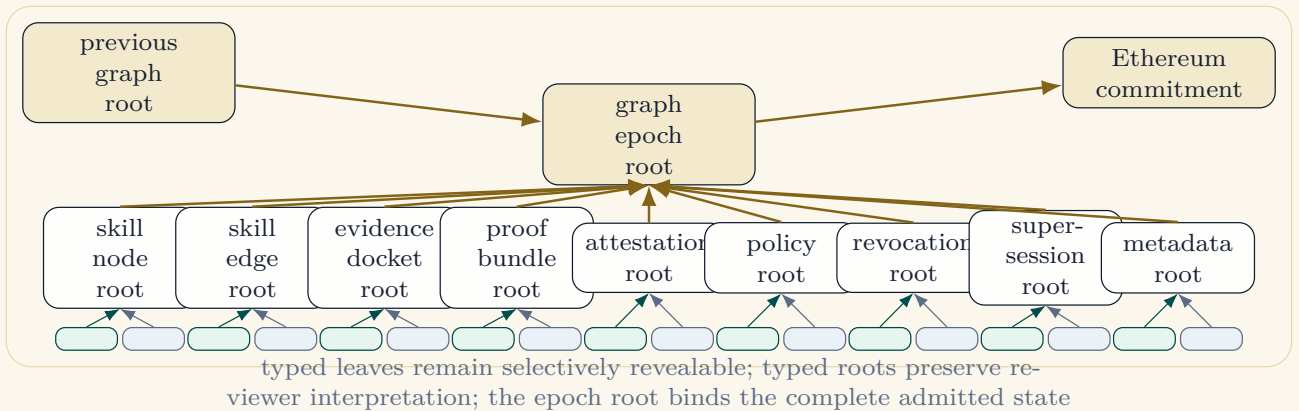


Figure 3: Merkle forest for the Validated Skill Graph. Split labels and typed root lanes keep every class visible while the graph epoch root binds the complete admitted state.

7 Why Merkle Roots Are Native to the VSG

Merkle roots solve six concrete problems.

1. **Compact public state.** The graph may contain thousands of skills and evidence artifacts, but the on-chain record can store a small root bundle.
2. **Membership proofs.** A reviewer can verify that a skill or evidence item belongs to a committed state without downloading the entire graph.
3. **Tamper evidence.** Changing a single leaf changes the root.
4. **Privacy boundary.** Private payloads remain encrypted or off-chain while their commitments remain verifiable.
5. **Version continuity.** Each root can point to the previous root, yielding append-only lineage.
6. **Selective disclosure.** A claimant can reveal one skill passport, its proof path, and its root commitment rather than revealing all other skills.

Core Insight

The Merkle root is not a storage trick. It is the cryptographic boundary between private capability and public authority.

8 Production Object Model

A Merkle-native VSG implementation should emit a `graph-root-packet.json`. A minimal packet is:

Listing 1: *Graph-root packet skeleton.*

```
{
  "schema": "goalos.graphRootPacket.v2",
  "agent": "john.agent.agi.eth",
  "agent_labelhash": "0x...",
  "agent_node": "0x...",
  "epoch": 7,
  "proof_level": "L3_EXTERNAL_REVIEW",
  "previous_graph_root": "0x...",
  "roots": {
    "skillNodeRoot": "0x...",
    "skillEdgeRoot": "0x...",
    "evidenceDocketRoot": "0x...",
    "proofBundleRoot": "0x...",
    "attestationRoot": "0x...",
    "policyRoot": "0x...",
    "revocationRoot": "0x...",
    "supersessionRoot": "0x...",
    "metadataRoot": "0x..."
  },
  "snapshotURI": "ipfs://...",
  "snapshotURIHash": "0x...",
  "schemaHash": "0x...",
  "policyHash": "0x...",
  "graphEpochRoot": "0x...",
  "tree_policy": "SORTED_PAIR_DUPLICATE_LAST_V2",
  "canonicalization": "JCS_KECCAK256_V2",
  "chronicle_decision": "ADMIT_WITH_SCOPE",
  "live_transaction": false,
  "boundary": {
    "plaintext_private_intelligence_onchain": false,
    "wallet_connected_in_local_mode": false,
```

```

    "settlement_in_local_mode": false
  }
}

```

8.1 Skill Leaf

A skill leaf commits to a skill passport:

Listing 2: *Validated skill leaf material.*

```

{
  "schema": "goalos.validatedSkill.v2",
  "skill_id": "skill-ai-claim-to-proof-room-001",
  "name": "Convert AI vendor claims into a buyer-ready proof room",
  "scope": {
    "domain": "AI vendor trust / procurement",
    "allowed_use": ["claim matrix", "evidence docket", "reviewer room"],
    "excluded_use": ["legal certification", "security certification", "investment advice"]
  },
  "proof_level": "L3_EXTERNAL_REVIEW",
  "evidence_docket_hash": "0x...",
  "proofbundle_refs": ["proofbundle-001", "proofbundle-002"],
  "validator_route": ["human-reviewer", "agi-node-validator"],
  "replay_manifest_hash": "0x...",
  "risk_ledger_hash": "0x...",
  "rollback_condition": "retire if reviewer verdict is invalidated",
  "worth_keeping_score": 0.86,
  "chronicle_admitted": true
}

```

8.2 Evidence and ProofBundle Leaves

Evidence leaves bind claim support and risk; ProofBundle leaves bind job outputs.

Listing 3: *Evidence and ProofBundle leaves.*

```

{
  "evidence_leaf": {
    "schema": "goalos.evidence.v2",
    "claim_id": "claim-001",
    "support_status": "supported_with_limits",
    "source_provenance_hash": "0x...",
    "replay_path_hash": "0x...",
    "blocked_claims_hash": "0x...",
    "reviewer_verdict_hash": "0x..."
  },
  "proofbundle_leaf": {
    "schema": "goalos.proofBundle.v2",
    "job_id": "agi-job-external-review-001",
    "worker_artifact_hash": "0x...",
    "validator_verdict_hash": "0x...",
    "acceptance_test_result_hash": "0x...",
    "rollback_hash": "0x..."
  }
}

```

9 Algorithms

9.1 Canonical Leaf Hashing

Listing 4: *Reference leaf-hashing pseudocode.*

```
function leafHash(domain, schemaHash, object):
  bytes = canonicalize(object)    // deterministic JSON / typed-data canonicalization
  return keccak256("GoalOS:" || domain || ":" || schemaHash || bytes)
```

9.2 Root Construction

Listing 5: *Sorted-pair Merkle root.*

```
function merkleRoot(leaves):
  if leaves.length == 0:
    return keccak256("GoalOS:EmptyRootV2")
  level = sort(leaves)
  while level.length > 1:
    next = []
    for i in range(0, level.length, 2):
      a = level[i]
      b = level[i+1] if i+1 < level.length else level[i]
      lo, hi = sortPair(a, b)
      next.push(keccak256("GoalOSMerkleNodeV2" || lo || hi))
    level = next
  return level[0]
```

9.3 Inclusion Verification

Listing 6: *Merkle membership proof verification.*

```
function verifyProof(leaf, proof, root):
  computed = leaf
  for sibling in proof:
    lo, hi = sortPair(computed, sibling)
    computed = keccak256("GoalOSMerkleNodeV2" || lo || hi)
  return computed == root
```

9.4 Graph-Root Transition

$$\text{ValidTransition}(R_{t-1}, R_t) = \text{AuthorizedAgent} \wedge \text{ChroniclePass} \wedge \text{RootContinuity} \wedge \text{PolicyBound}. \quad (15)$$

Listing 7: *Graph-root update rule.*

```
function commitGraphRoot(packet):
  require(authorizeDirectAgent(packet.agent_labelhash, msg.sender))
  require(packet.epoch == latestEpoch[packet.agent_labelhash] + 1)
  require(packet.previous_graph_root == latestRoot[packet.agent_labelhash])
  require(packet.chronicle_decision in ["ADMIT", "ADMIT_WITH_SCOPE"])
  require(packet.policy_root != ZERO)
  require(packet.evidence_root != ZERO)
  require(packet.proofbundle_root != ZERO)
  require(computeGraphRoot(packet) == packet.expected_graph_root)
  latestRoot[packet.agent_labelhash] = packet.graph_root
  emit GraphRootCommitted(packet.agent_labelhash, packet.graph_root, packet.previous_graph_root)
```

10 Chronicle-Gated Root Admission

A Merkle root is only as meaningful as its admission policy. GoalOS must distinguish between roots over candidate data and roots over Chronicle-admitted skills. Candidate roots may be useful for debugging, but they must not confer future-mission influence.

Let c be a candidate skill, ℓ a proof level, $E(c)$ evidence strength, $R(c)$ risk posture, $V(c)$ validation coverage, $P(c)$ replay readiness, $B(c)$ boundary compliance, and $W(c)$ worth-keeping utility. Define:

$$\text{ChroniclePass}_\ell(c) = \mathbf{1}\{E(c) \geq \theta_\ell^E \wedge R(c) \leq \theta_\ell^R \wedge V(c) \geq \theta_\ell^V \wedge P(c) = 1 \wedge B(c) = 1 \wedge W(c) \geq \theta_\ell^W\}. \quad (16)$$

Then:

$$\text{FutureInfluence}(c, t) = 1 \Rightarrow \text{ChroniclePass}_\ell(c) = 1 \wedge \text{Verify}(\text{Leaf}_{\text{Skill}}(c), \pi_c, \text{skillNodeRoot}_t) = 1. \quad (17)$$

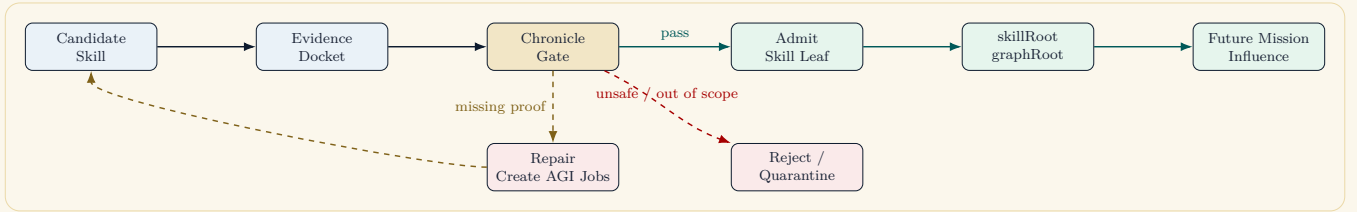


Figure 4: Chronicle-gated Merkle admission. The root is trusted only when it commits Chronicle-admitted skill leaves, not candidate output.

11 Inclusion, Non-Inclusion, and Root Review

A Merkle proof is a membership proof. It says that a leaf belongs to a committed root. It does not say that the leaf is true, safe, legal, or reusable. Reviewer UX must therefore make four operations obvious:

1. **Build roots.** Show leaves, typed roots, and final graph root.
2. **Verify inclusion.** Select a skill or evidence leaf and show its Merkle path.
3. **Run tamper test.** Modify a leaf and show the root mismatch.
4. **Compare roots.** Show old root, new root, diff summary, admitted skills, retired skills, and repair jobs.

User action	What the user sees	Concept taught
Build Merkle Forest	Leaves become roots; roots become graphRoot.	Large proof state can be compactly committed.
Verify Proof	One leaf follows a path to the root.	Membership can be checked without full graph disclosure.
Tamper Test	One character change breaks the proof.	Integrity is structural, not visual.
Commit Packet	Root bundle becomes a reviewable packet with liveTransaction=false.	Local proof preparation is separate from live blockchain action.
Future Mission	Mission lift appears only after Chronicle and root pass.	Rooted memory governs future influence.

Table 4: User-facing Merkle interactions. The interface should teach by changing state, not by showing static claims.

12 ENS-Gated Direct AGI Agent Authorization

Graph-root writing is authority-bearing. The registry should not accept arbitrary callers or arbitrary names. For a label λ such as `john`, define:

$$labelhash(\lambda) = H(\lambda), \quad (18)$$

$$agentNode(\lambda) = namehash(\lambda.agent.agi.eth). \quad (19)$$

Equivalently, if $P = \text{namehash}(\text{agent.agi.eth})$, then:

$$N_\lambda = \mathsf{H}(P \parallel \textit{labelhash}(\lambda)). \quad (20)$$

The contract should check current ownership or approved delegation for N_λ ; callers should not submit arbitrary ENS nodes as proof of authority.

Authorization Law

A graph-root writer must be the current controller, wrapped-name owner, approved operator, or registry-approved delegate for the direct `λ.agent.agi.eth` node. Deeper names may exist, but they do not grant direct graph-write authority.

13 Public-Private Proof Boundary

The on-chain graph root is a public commitment, not the skill itself. The full skill lives in the encrypted capsule, Evidence Docket, ProofBundles, validator attestations, replay manifests, risk ledgers, Chronicle decisions, and policy records. The correct boundary is:

Private intelligence: off-chain or encrypted

Public coordination: proof commitments, roots, attestations, and receipts.

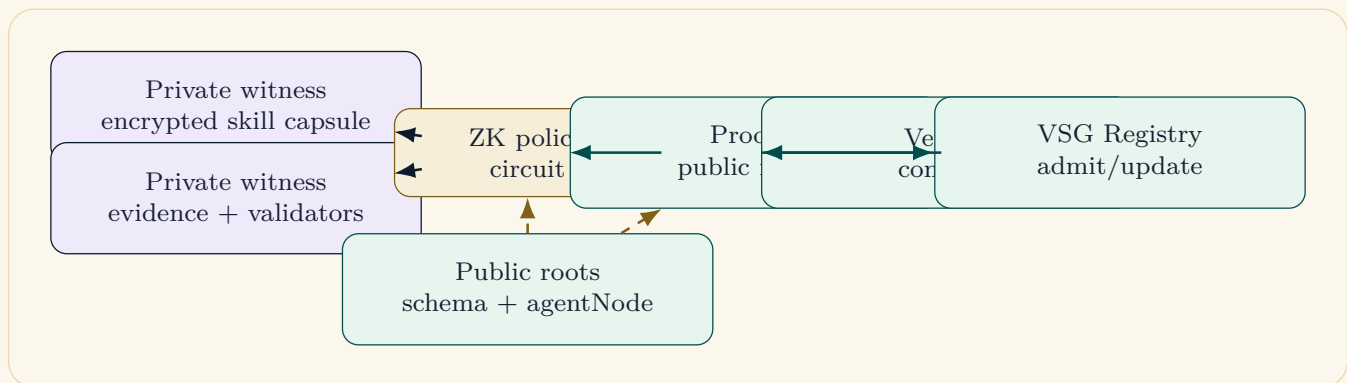


Figure 5: *Public-private proof boundary. The registry learns that policy predicates are satisfied, not the hidden method, trace, reviewer notes, or sensitive skill contents.*

14 Zero-Knowledge Compatibility

Zero-knowledge is useful when the system needs to prove that a hidden skill capsule satisfies a policy without revealing the underlying intelligence. Define a private witness:

$$W = (C, E, P, V, K), \quad (21)$$

where C is the skill capsule, E the Evidence Docket, P the ProofBundle set, V the validator attestations, and K the encryption/key policy. Public inputs include the roots and policy identifier:

$$X = (\text{graphRoot}, \text{evidenceRoot}, \text{proofBundleRoot}, \text{schemaHash}, \text{proofLevel}, \text{agentNode}, \text{policyHash}). \quad (22)$$

The proof statement is:

$$\text{Verify}_{zk}(X, \pi) = 1 \Rightarrow \begin{cases} C \text{ commits to } \text{graphRoot}, \\ E \text{ commits to } \text{evidenceRoot}, \\ P \text{ commits to } \text{proofBundleRoot}, \\ \text{validatorquorumissatisfied}, \\ \text{blockedclaimsareexcluded}, \\ \text{prooflevelthresholdismet}, \\ \text{rollbackconditionexists}. \end{cases} \quad (23)$$

15 Registry Surface

A minimal production registry stores proof commitments, not private skill payloads. The authority-changing operation can be expressed as:

Listing 8: *CommitGraphEpochParams* surface.

```
commitGraphEpoch({
  labelhash,
  epoch,
  previousEpochRoot,
  roots: {
    skillNodeRoot,
    skillEdgeRoot,
    evidenceDocketRoot,
    proofBundleRoot,
    attestationRoot,
    policyRoot,
    revocationRoot,
    supersessionRoot,
    metadataRoot
  },
  snapshotURI,
  snapshotURIHash,
  schemaHash,
  policyHash,
  proofLevel,
  expectedGraphEpochRoot
})
```

The contract should verify direct-agent ownership, epoch number, previous-root continuity, expected-root match, nonzero roots, proof-level range, and append-auditable event emission. Supersession and revocation preserve history while removing or replacing future influence.

16 Threat Model

Threat	Failure mode	Mitigation
Leaf tampering	Evidence, skill, or ProofBundle content is changed after review.	Canonical leaf hash changes; inclusion proof fails; root mismatch visible.
Root equivocation	Operator presents different roots to different reviewers.	Append-only root history, previous-root binding, signed graph-root packets.
Cross-domain replay	Evidence leaf is reused as a skill leaf or policy leaf.	Domain-separated leaf hashing and schema hashes.
Unauthorized write	Non-agent or nested-name caller commits graph root.	Direct *.agent.agi.eth derivation and owner/delegate checks.
Stale root influence	Future mission uses old or superseded root.	Root versioning, retirement events, freshness checks, latest-root query.
Memory contamination	Unsupported claim enters future mission context.	Chronicle Gate before skill leaf admission; future-prior only from admitted roots.
Privacy leakage	Sensitive method or data is published as plaintext.	Encrypt capsules; publish roots, hashes, commitments, ZK proofs, controlled pointers only.
Hash migration risk	Hash algorithm weakens or changes.	Policy-root hash suite, migration certificates, dual-root transition windows.
Non-deterministic serialization	Same object hashes differently across clients.	Canonicalization standard, conformance vectors, golden test fixtures.
Validator capture	Validators collude to admit a weak skill.	Higher proof levels, multi-validator quorum, challenge windows, replay, slashing, reviewer escalation.

Table 5: *Threat model for Merkle-native Validated Skill Graphs.*

17 Evaluation Program

A Merkle-native VSG implementation should be evaluated against eight test families.

1. **Determinism tests.** Same object, same canonicalization, same hash across languages and platforms.
2. **Tamper tests.** One changed leaf, signature, proof path, or policy field invalidates the expected root.
3. **Inclusion tests.** Randomly sampled skills and evidence items verify against committed roots.
4. **Non-contamination tests.** Candidate and rejected skills cannot appear in future mission prompts or skill roots.
5. **Privacy tests.** Public packets reveal only roots, hashes, ciphertext pointers, agent identifiers, and policy-level metadata.
6. **Authorization tests.** Direct-agent writes pass; nested-name and unrelated-name writes fail.
7. **Root-continuity tests.** Graph updates bind to the previous root; history cannot be silently rewritten.

8. **Reviewer usability tests.** A non-technical reviewer can build roots, verify inclusion, run a tamper test, compare roots, and understand what remains unproven.

Falsification Standard

The Merkle-native VSG claim fails if a future mission can be influenced by a skill that is not Chronicle-admitted and inclusion-verifiable under the latest authorized graph root.

18 Production Conformance Levels

Level	Name	Requirement
M0	Local Hashing	Deterministic leaf hashes for skills, evidence, ProofBundles, policies, and attestations.
M1	Merkle Forest	Typed roots and final graphRoot produced from canonical leaves.
M2	Proof UX	Inclusion proof, tamper test, and root comparison visible in the interface.
M3	Chronicle Binding	Only Chronicle-admitted skill leaves enter future-influence roots.
M4	Agent Authorization	Direct *.agent.agi.eth write gate and delegate policy implemented.
M5	Privacy Boundary	Encrypted capsule commitments and/or ZK proof hashes included; plaintext private intelligence excluded.
M6	Mainnet Ready	Audited contract, testnet deployment, event indexer, replayable root packets, and rollback procedure.
M7	Independent Review	External replay, independent validator report, root-continuity audit, and challenge-window process.
M8	Production Governance	Root migration, incident procedure, operator review, and legal/security boundary review.
M9	Cross-Institutional Reuse	Selective disclosure and verification can be performed by an external institution without exposing private capsules.

Table 6: Conformance levels for Merkle-native Validated Skill Graphs.

19 Reference Architecture

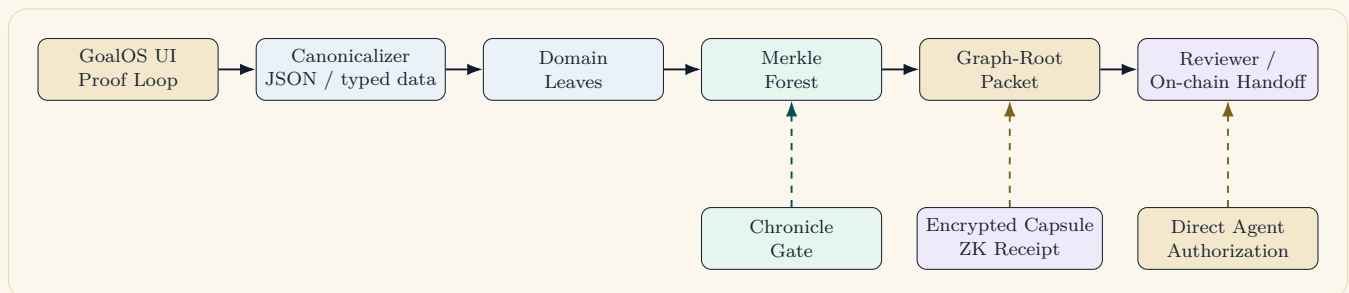


Figure 6: Reference implementation flow. Merkle roots sit between the local proof loop and the reviewer or on-chain commitment boundary.

19.1 Required Artifacts

A production Merkle-root run should export:

Artifact	Purpose
leaf-set.json	Canonical leaves grouped by domain.
merkle-forest.json	Typed roots, tree policy, leaf count, and proof vectors.
graph-root-packet.json	Final graph root, previous root, agent, mission, policy, and live-mode flags.
membership-proofs/ tamper-report.json	Inclusion proofs for selected skills, evidence, bundles, and attestations. Demonstrates root mismatch when a sampled leaf changes.
chronicle-decision.json	Shows why a skill was admitted, repaired, rejected, quarantined, or retired.
reviewer-room.html	Human-readable proof room for inspection.
mainnet-handoff.json	Wallet-free packet for review before any live transaction.

Table 7: *Minimum artifact standard for a Merkle-native VSG run.*

20 Reviewer Workflow

A Merkle-native VSG is valuable only if reviewers can inspect it without becoming cryptographers. The reviewer path should be:

1. Select the claim or skill whose future influence is being requested.
2. Open the Evidence Docket and identify the associated skill leaf, evidence leaf, ProofBundle leaf, attestation leaf, and policy leaf.
3. Recompute the canonical object hash and domain-separated leaf hash.
4. Verify the Merkle proof against the relevant class root.
5. Verify that the class root is bound into the graph epoch root.
6. Verify the graph epoch root is the latest authorized root for the direct AGI Agent, or a permitted historical root under the task policy.
7. Read the Chronicle decision to determine whether inclusion has authority.
8. Check scope, freshness, revocation, supersession, and rollback conditions.
9. If context diverges, create a new Proof Mission rather than reusing the old skill.

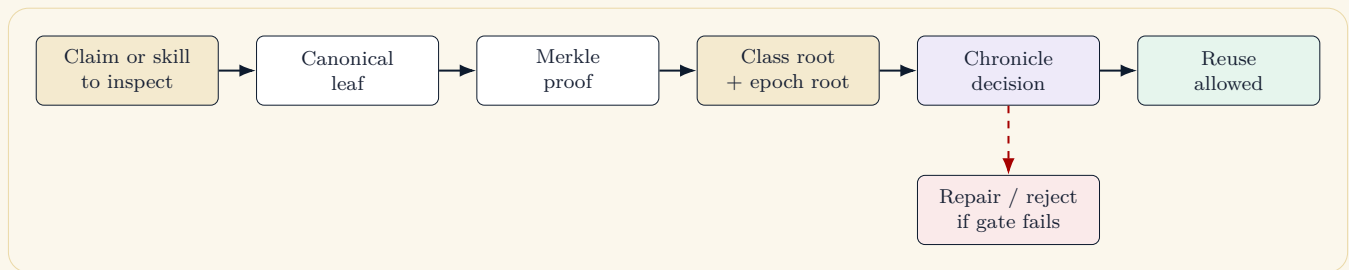


Figure 7: *Reviewer path. A verifier does not trust the root alone: the root supports inclusion, while Chronicle supplies the reuse decision.*

21 Implementation Blueprint

A minimal implementation should contain:

1. a canonicalizer with cross-language test vectors;

2. a Merkle forest builder with typed domains;
3. a proof verifier and tamper-test module;
4. a graph-root packet generator;
5. a Chronicle decision exporter;
6. a reviewer room that explains inclusion versus authority;
7. an ENS-gated registry contract for graph epoch commits;
8. an indexer for root history and latest-root queries;
9. a migration pathway for hash-suite upgrades;
10. a ZK policy proof adapter for hidden skill-capsule predicates.

The reference contract should expose:

Listing 9: *Minimal registry methods.*

```
computeLeaf(leafDomain, objectHash)
computeNode(a, b)
verifyMerkleProof(root, leaf, proof)
verifyEpochLeaf(graphEpochRoot, leafClass, objectHash, proof)
computeGraphEpochRoot(agentNode, epoch, previousRoot, roots, snapshotURIHash,
                        schemaHash, policyHash, proofLevel)
commitGraphEpoch(params)
revokeGraphEpoch(graphEpochRoot, reasonURI)
epochsByAgentNode(agentNode)
latestEpochRootByAgentNode(agentNode)
```

22 V107 Reference Implementation Surface

The implementation surface that corresponds to this paper is a production-style Merkle forest registry and builder. The contract side should expose the following primitives, and the off-chain builder should produce exactly the packet those primitives expect.

Primitive	Purpose
<code>computeLeaf(leafDomain, objectHash)</code>	Domain-separated object commitment.
<code>computeNode(a, b)</code>	Sorted-pair Merkle parent using a node-domain separator.
<code>verifyMerkleProof(root, leaf, proof)</code>	Pure inclusion verification.
<code>verifyEpochLeaf(graphEpochRoot, leafClass, objectHash, proof)</code>	Registry-bound inclusion check for a committed epoch.
<code>computeGraphEpochRoot(...)</code>	Binds chain, registry, agent node, epoch, previous root, class roots, snapshot hash, schema hash, policy hash, and proof level.
<code>commitGraphEpoch(params)</code>	Writes a graph epoch after direct-agent authorization, expected-root match, epoch-number check, and previous-root continuity.
<code>revokeGraphEpoch(root, reasonURI)</code>	Removes future influence without erasing history.

Table 8: *Reference implementation surface for production Merkle VSG commits.*

V107 compatibility requirement

A production packet is not complete unless an independent builder can recompute the `expectedGraphEpochRoot`, verify sample inclusion proofs, and confirm that the on-chain commit either matches the expected root or fails closed.

23 Claim Boundary

This paper does not claim achieved AGI, achieved ASI, superintelligence, production authorization, legal certification, financial advice, completed security audit, deployed mainnet registry, or live settlement. It defines a research and implementation architecture for using Merkle roots inside the Validated Skill Graph. Empirical claims require real missions, Evidence Dockets, ProofBundles, reviewer verdicts, reproducible traces, testnet or mainnet transactions where applicable, security review, and independent validation.

24 Conclusion

The Validated Skill Graph is the compounding memory layer of GoalOS. Merkle roots make that layer cryptographically legible. They let the system commit to proof-carrying skill state without exposing everything inside the graph. They let reviewers verify inclusion. They make tampering visible. They connect Chronicle-admitted capability to on-chain graph-root commitments. They let future missions learn from memory that is not merely persuasive, but proven, scoped, versioned, and root-verifiable.

Final Operating Law

Proof work creates evidence. Evidence creates Chronicle candidates. Chronicle admits validated skills. Validated skills create Merkle graph roots. Merkle graph roots create trusted memory. Trusted memory improves the next mission.

Appendix A: Canonical Merkle Packet Example

```
{
  "schema": "goalos.merkleForest.v2",
  "tree_policy": "SORTED_PAIR_DUPLICATE_LAST_V2",
  "hash": "keccak256",
  "domains": {
    "skill": "GoalOSSkillV2",
    "edge": "GoalOSEdgeV2",
    "evidence": "GoalOSEvidenceV2",
    "proofbundle": "GoalOSProofBundleV2",
    "policy": "GoalOSPolicyV2",
    "attestation": "GoalOSAttestationV2",
    "revocation": "GoalOSRevocationV2",
    "supersession": "GoalOSSupersessionV2",
    "metadata": "GoalOSMetadataV2"
  },
  "roots": {
    "skillNodeRoot": "0x...",
    "skillEdgeRoot": "0x...",
    "evidenceDocketRoot": "0x...",
    "proofBundleRoot": "0x...",
    "policyRoot": "0x...",
    "attestationRoot": "0x..."
  }
}
```

```

    "revocationRoot": "0x...",
    "supersessionRoot": "0x...",
    "metadataRoot": "0x...",
    "graphEpochRoot": "0x..."
  },
  "proofs": {
    "skill-ai-claim-to-proof-room-001": {
      "leaf": "0x...",
      "path": ["0x...", "0x...", "0x..."]
    }
  }
}

```

Appendix B: Acceptance Tests

1. The same canonical leaf hashes identically across two independent clients.
2. A modified leaf fails inclusion verification.
3. A modified proof path fails inclusion verification.
4. A candidate skill cannot appear in `skillNodeRoot` unless `chronicle_admitted=true`.
5. A rejected or quarantined skill has a preserved reason hash and no future influence.
6. A new graph root binds the previous graph root.
7. The graph-root packet contains `live_transaction=false` in local mode.
8. The packet contains no plaintext private skill content.
9. Direct `john.agent.agi.eth` authorization can pass; nested `x.john.agent.agi.eth` cannot grant direct graph-write authority.
10. A reviewer can verify an inclusion proof without seeing the full private graph.

Appendix C: Glossary

AGI Job	An executable proof object created from Proof Debt.
Chronicle	The GoalOS memory gate that admits, repairs, rejects, quarantines, retires, or supersedes candidate capability.
Evidence Docket	An inspectable proof room tying claims to evidence, risks, replay, validators, and decision state.
Graph Root	A typed commitment to the Validated Skill Graph state.
Merkle Proof	A path proving that a leaf belongs to a committed root.
ProofBundle	Returned evidence from a completed AGI Job.
Validated Skill	A scoped, evidence-backed, replayable, versioned, rollbackable reusable capability object.

References

- [1] Vincent Boucher. *PROOF_LOOPS.md: Field Notes on Machines That Turn Work Into Capability*. GoalOS Protocol Note Series, 2026.
- [2] Vincent Boucher. *GoalOS Mission OS: The Proof OS for Autonomous AI Work*. Montreal.AI Research, 2026.
- [3] Vincent Boucher. *GoalOS: The Validated Skill Graph*. Montreal.AI Research, 2026.
- [4] Vincent Boucher. *GoalOS V97.4: Private On-Chain Validated Skill Graph*. Montreal.AI Research, 2026.
- [5] Vincent Boucher. *GoalOS: The Proof-of-Evolution Constitution, AEP-001*. Montreal.AI Research, 2026.
- [6] Vincent Boucher. *AGI ALPHA: A Scalable Substrate for Intelligence Organizations*. Montreal.AI Research, 2026.
- [7] Ralph C. Merkle. *A Digital Signature Based on a Conventional Encryption Function*. CRYPTO 1987, Lecture Notes in Computer Science 293, 1988.
- [8] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Ethereum Yellow Paper.
- [9] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin*. IEEE Symposium on Security and Privacy, 2014.

Appendix D: MVS-001 Acceptance Checklist

A Merkle VSG implementation passes the MVS-001 checklist only if:

1. all object classes have explicit domains and schema hashes;
2. canonicalization test vectors pass across at least two implementations;
3. sample inclusion proofs verify for every class root;
4. a tamper test changes the relevant class root and graph epoch root;
5. rejected or quarantined skills cannot appear in future-influence roots;
6. direct AGI Agent authorization rejects nested-name inheritance;
7. snapshot URI, policy hash, schema hash, proof level, and previous root are bound into the graph epoch root;
8. revocations and supersessions preserve history while blocking stale future influence;
9. public root packets do not reveal plaintext private intelligence;
10. reviewer documentation states that inclusion is not authority.

Appendix E: Minimal Research Evaluation Table

Evaluation	Passing evidence	Falsifying evidence
Root determinism	Recomputable graph epoch root from public packet and private or redacted leaf set.	Same packet yields different roots across compliant implementations.
Selective disclosure	Reviewer verifies one skill leaf without seeing unrelated private skills.	Proof requires disclosing the entire graph.
Chronicle binding	Future missions use only admitted, in-scope, root-verifiable skills.	Candidate or rejected leaves influence future missions.
Privacy boundary	On-chain/public packet contains roots, hashes, ciphertext pointers, and attestations only.	Plaintext private method, reviewer notes, or sensitive data appear in public proof.
Governance recovery	Revocation or supersession prevents stale root from future influence.	A revoked skill remains active without policy override.

Table 9: *Minimal research evaluation table for Merkle roots in the Validated Skill Graph.*