SPICE
Internet-Draft
Intended status: Informational
Expires: 19 September 2026

R. Krishnan
JPMorgan Chase & Co
A. Prasad
Oracle
D. Lopez
Telefonica
S. Addepalli
Aryaka

18 March 2026

Cryptographically Verifiable Inference Chain for AI Agent Computational Provenance

draft-mw-spice-inference-chain-00

Abstract

This document defines the inference_root claim as a companion to the actor_chain claim ({{!I-D.draft-mw-spice-actor-chain}}) and the intent_root claim ({{!I-D.draft-mw-spice-intent-chain}}). While the actor chain addresses delegation provenance (WHO) and the intent chain addresses content provenance (WHAT), the inference chain addresses computational provenance (HOW) — providing cryptographic proof that a claimed AI model actually performed the inference that produced a given output.

The inference chain leverages two complementary mechanisms: Zero-Knowledge Machine Learning (ZKML) proofs for mathematical certainty, and Trusted Execution Environment (TEE) attestation quotes for production-scale AI workloads. The full inference chain is stored as ordered logs, with only the Merkle root included in the OAuth token for efficiency.

Together, the three chains — actor, intent, and inference — form a complete "Truth Stack" for autonomous AI agent governance.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

This Internet-Draft will expire on 19 September 2026.

Copyright Notice

Table of Contents

1. Introduction

1.1. The Problem: Computational Spoofing

The Actor Chain ({{!I-D.draft-mw-spice-actor-chain}}) proves WHO delegated to whom. The Intent Chain ({{!I-D.draft-mw-spice-intent-chain}}) proves WHAT was produced and how it was transformed. Neither addresses a critical remaining gap: was the *claimed computational process* actually used to produce the output?

In AI agent workflows, an agent may claim to use a high-capability, safety-aligned model (e.g., a frontier model with RLHF alignment) while actually running a cheaper, unaligned model. This "Model

Masquerading" attack is undetectable by content hashes alone — the output hash in the intent chain proves the output exists, but not that the claimed model produced it.

Concrete threats:

* *Model Substitution*: A cost-optimized deployment swaps a regulated, audited model for a cheaper alternative with weaker safety properties.
* *Weight Tampering*: Model weights are modified (e.g., fine-tuned to remove safety guardrails) while the model identifier remains unchanged.
* *Environment Spoofing*: Inference runs in an unprotected environment where model inputs/outputs can be intercepted, while the agent claims TEE execution.
* *Replay of Stale Proofs*: A valid inference proof from a previous execution is reattached to a new, unverified output.

1.2. Relationship to Actor Chain and Intent Chain

This specification completes the three-axis "Truth Stack":

Specification	Axis	Question Answered	STRIDE Coverage
Actor Chain ({{!I-D.draft-mw-spice-actor-chain}})	Identity	WHO delegated to whom?	Spoofing, Repudiation, Elevation of Privilege
Intent Chain ({{!I-D.draft-mw-spice-intent-chain}})	Content	WHAT was produced and transformed?	Repudiation, Tampering
Inference Chain (this document)	Computation	HOW was the output computed?	Spoofing (computational), Tampering (model)

Table 1

Chain	Plane	Token Content	Full Chain	Primary Consumer
Actor	Data Plane	Full chain inline	In token	Every Relying Party (real-time authorization)
Intent	Audit Plane	Merkle root only	External registry	Audit systems, forensic investigators
Inference	Audit Plane	Merkle root only	External registry	Auditors, compliance systems

Table 2

The three chains are independent and composable:

* *Actor + Intent*: Identity and content governance without computational proof (sufficient for rule-based systems).
* *Actor + Inference*: Identity and computational governance without content transformation tracking (sufficient for single-agent systems).
* *All Three*: Full governance for multi-agent pipelines with content transformation and computational integrity.

1.3. Design Goals

1. *Computational Provenance*: Cryptographic proof that a specific model produced a specific output.
2. *Mechanism Agnostic*: Support both ZKML proofs and TEE attestation quotes as proof types.
3. *Binding to Intent Chain*: Inference proofs are bound to specific intent chain entries via content hashes.
4. *Efficiency*: Only Merkle root in token; full proofs in registry.
5. *Incremental Adoption*: Deployable alongside existing actor and intent chains without requiring changes to those specifications.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 {{!RFC2119}} {{!RFC8174}} when, and only when, they appear in all capitals, as shown here.

Inference Chain: An ordered sequence of Inference Chain Entries providing cryptographic proof of the computational processes that produced AI agent outputs within a session.

Inference Chain Entry: A record binding a specific AI agent output to a cryptographic proof of the computational process that produced it.

Inference Root: The Merkle root hash of the complete inference chain, included in the OAuth token via the inference_root claim.

Inference Registry: An append-only ordered log storing the full inference chain entries, partitioned by session.

ZKML Proof: A Zero-Knowledge proof (SNARK or STARK) demonstrating that a specific neural network with specific weights produced a specific output from a specific input, without revealing the model weights or input data.

TEE Quote: A hardware-signed attestation from a Trusted Execution Environment (e.g., Intel TDX, AMD SEV-SNP, NVIDIA H100 Confidential Computing) proving that specific code ran within a verified enclave with specific measurements. TEE attestation follows the RATS architecture {{!RFC9334}}.

Model Fingerprint: A cryptographic hash of the model weights, architecture definition, and inference configuration, uniquely identifying the computational process.

3. Inference Proof Types

3.1. Overview

The inference chain supports two primary proof types, each with different trust assumptions and performance characteristics:

Property	ZKML Proof	TEE Quote
Trust Basis	Mathematics (cryptographic hardness)	Hardware manufacturer root of trust
Verification	Deterministic, no external dependency	Requires manufacturer PKI (Intel/AMD/NVIDIA)
Model Scale	Currently limited (~100M parameters)	Production-scale LLMs (100B+ parameters)
Proof Generation	Minutes to hours per inference	Real-time (millisecond overhead)
Proof Size	~200 bytes (Groth16), ~50KB (STARKs)	~2KB (Intel TDX), ~4KB (NVIDIA)
Privacy	Zero-knowledge: hides model weights	Enclave-bound: hardware isolation

Table 3

Deployments SHOULD select the proof type based on their performance requirements and trust model.

3.2. ZKML Proofs

ZKML proofs provide mathematical certainty that a specific model produced a specific output. They are constructed by encoding the neural network's forward pass as an arithmetic circuit and generating a zero-knowledge proof of correct execution.

Proof Statement: "There exist model weights W such that hash(W) = model_fingerprint AND forward_pass(W, input) = output."

Properties:

* The verifier learns nothing about the model weights (zero-knowledge property).
* The proof cannot be forged without performing the actual computation (soundness).

* Proof verification is fast (milliseconds) regardless of model size.
* Proof generation is computationally expensive (orders of magnitude slower than inference).

Applicable Proof Systems:

System	Proof Size	Verification Time	Trusted Setup
Groth16	~200 bytes	_5ms	Required (per-circuit)
PLONK	~400 bytes	_10ms	Universal (one-time)
STARKs	_50KB	_50ms	None (transparent)
Halo2	_5KB	_15ms	None (recursive)

Table 4

Entry Structure:

```
{
  "type": "zkml_proof",
  "sub": "spiffe://example.com/agent/analyst",
  "proof_system": "groth16",
  "model_fingerprint": "sha256:weights_hash...",
  "model_id": "analyst-model-v3.2",
  "input_hash": "sha256:input_data...",
  "output_hash": "sha256:output_data...",
  "intent_entry_ref": 3,
  "proof": "base64:proof_bytes...",
  "verification_key_hash": "sha256:vk_hash...",
  "verification_key_registry":
    "https://vk-registry.example.com/keys/analyst-v3.2",
  "iat": 1700000030,
  "inference_digest": "sha256:...",
  "inference_sig": "eyJhbGci..."
}
```

3.3. TEE Attestation Quotes

TEE attestation quotes provide hardware-rooted proof that inference was performed within a verified enclave. The hardware generates a signed "quote" containing measurements of the code, data, and configuration running inside the enclave.

Proof Statement: "Hardware H with platform certificate C attests that enclave E with measurements M produced output O from input I."

Properties:

* Proof is generated in real-time with minimal overhead.
* Verification requires the hardware manufacturer's PKI (root certificate).
* The quote binds the computation to specific hardware and firmware versions.
* The enclave protects model weights and inference data from the host OS.

Supported Platforms:

Platform	TEE Technology	GPU Support	Quote Format
Intel	TDX (Trust Domain Extensions)	Via NVIDIA H100 CC	DCAP Quote v4
AMD	SEV-SNP	Via NVIDIA H100 CC	Attestation Report
NVIDIA	H100 Confidential Computing	Native	GPU Attestation
ARM	CCA (Confidential Compute Architecture)	Planned	CCA Token

Table 5

Entry Structure:

```
{
  "type": "tee_attestation",
  "sub": "spiffe://example.com/agent/analyst",
  "platform": "nvidia_h100_cc",
  "model_fingerprint": "sha256:weights_hash...",
  "model_id": "analyst-model-v3.2",
  "input_hash": "sha256:input_data...",
  "output_hash": "sha256:output_data...",
  "intent_entry_ref": 3,
  "quote": {
    "format": "nvidia_gpu_attestation_v1",
    "enclave_measurement": "sha384:enclave_mr...",
    "firmware_version": "H100.96.00.5E.00.01",
    "platform_cert_chain": [
      "base64:nvidia_device_cert...",
      "base64:nvidia_intermediate_cert..."
    ],
    "report_data": "sha256:binding_hash...",
    "signature": "base64:hw_signature..."
  },
  "por_ref": "spiffe://example.com/wia/gpu-node-7",
  "iat": 1700000030,
  "inference_digest": "sha256:...",
  "inference_sig": "eyJhbGci..."
}
```

3.4. Hybrid Proofs

For high-assurance deployments, both proof types MAY be combined for a single inference operation:

* *TEE quote* provides real-time proof that inference ran in a protected enclave.
* *ZKML proof* provides mathematical proof of correct computation (generated asynchronously after inference).

The inference chain entry for a hybrid proof references both:

```
{
  "type": "hybrid_proof",
  "sub": "spiffe://example.com/agent/analyst",
  "tee_entry_ref": 4,
  "zkml_entry_ref": 5,
  "model_fingerprint": "sha256:weights_hash...",
  "output_hash": "sha256:output_data...",
  "intent_entry_ref": 3,
  "iat": 1700000035,
  "inference_digest": "sha256:...",
  "inference_sig": "eyJhbGci..."
}
```

4. Entry Structure

4.1. Common Fields

All inference chain entries share common fields:

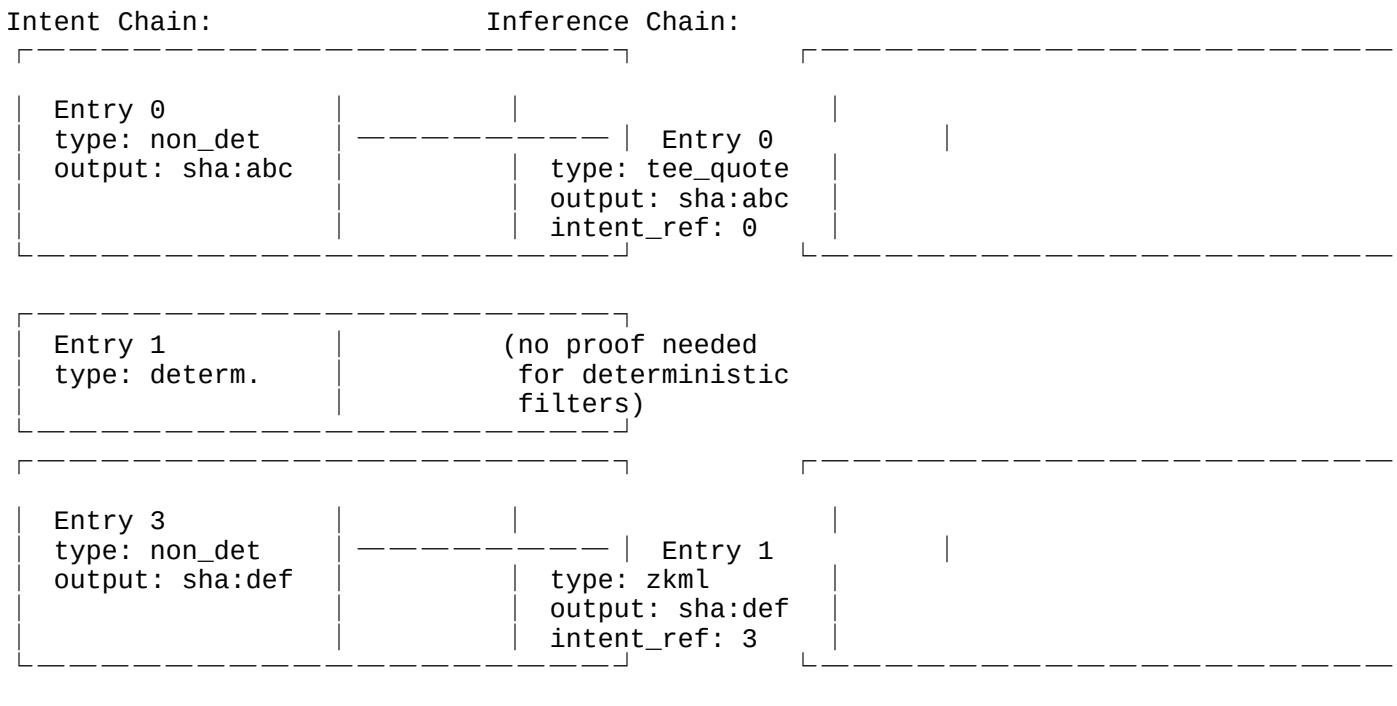
Field	Type	Required	Description
type	string	REQUIRED	Entry type: zkml_proof, tee_attestation, hybrid_proof
sub	string	REQUIRED	SPIFFE ID of the agent that performed inference
model_fingerprint	string	REQUIRED	SHA-256 hash of model weights + architecture
model_id	string	REQUIRED	Human-readable model identifier and version
output_hash	string	REQUIRED	SHA-256 hash of inference output (MUST match intent chain entry)
intent_entry_ref	number	REQUIRED	Index of corresponding entry in intent chain
iat	number	REQUIRED	Timestamp when proof was generated
inference_digest	string	REQUIRED	Cumulative hash for Merkle tree
inference_sig	string	REQUIRED	Signature over inference_digest

Table 6

The sub field identifies the agent that performed inference. It corresponds to the sub field of the matching actor chain entry ({{!I-D.draft-mw-spice-actor-chain}}).

4.2. Binding to Intent Chain

The intent_entry_ref field binds each inference proof to a specific intent chain entry. The output_hash in the inference entry MUST match the output_hash (for agent outputs) in the referenced intent chain entry. This binding ensures that the computational proof corresponds to the actual content recorded in the intent chain.



5. Storage Architecture

5.1. Inference Registry

The inference registry stores immutable inference chain entries as ordered logs, following the same architecture as the intent registry.

Properties:

* Append-only (immutable)
* Ordered by offset within session
* Partitioned by session ID
* Eventual consistency acceptable

| Inference registry entries MUST NOT contain OAuth tokens, bearer credentials, or signing keys. Entries contain only inference proofs, content hashes, metadata, and agent identities. The token references the registry via the inference_registry claim; the registry MUST NOT store or reference the token itself.

Implementations SHOULD use an append-only log with tamper-evident guarantees and partitioned, ordered retrieval by session ID.

Log Structure:

```
{
  "session_id": "sess-uuid-12345",
  "offset": 0,
  "entry": {
    "type": "tee_attestation",
    "sub": "spiffe://example.com/agent/A",
    "model_fingerprint": "sha256:weights...",
    "model_id": "agent-A-model-v2",
    "output_hash": "sha256:abc...",
    "intent_entry_ref": 0,
    "quote": { "..." : "..." },
    "iat": 1700000010,
    "inference_digest": "sha256:...",
    "inference_sig": "eyJ..."
  }
}
```

5.2. Merkle Tree Construction

The inference chain Merkle tree follows the same construction algorithm as the intent chain (defined in {{!I-D.draft-mw-spice-intent-chain}} Section 5.3). Leaf nodes are the SHA-256 hashes of canonically serialized inference chain entries. The resulting root hash is included in the OAuth token as the inference_root claim.

5.3. Merkle Root in Token

Only the Merkle root is included in the OAuth token:

```
{
  "inference_root": "sha256:xyz789...",
  "inference_proof_type": "groth16",
  "inference_registry":
    "https://proof-log.example.com"
}
```

Field	Type	Required	Description
inference_root	string	REQUIRED	Merkle root hash of inference chain
inference_proof_type	string	OPTIONAL	Primary proof algorithm used (e.g., groth16, tee_tdx, hybrid)
inference_registry	string	REQUIRED	URI of inference registry for proof retrieval

Table 7

6. Token Structure

6.1. Combined Token Format (Full Truth Stack)

The complete token combines session, actor chain, intent chain, and inference chain:

```
{
  "iss": "https://auth.example.com",
  "sub": "user-alice",
  "aud": "https://api.example.com",
  "jti": "tok-eee-12345",
  "sid": "sess-uuid-12345",
  "iat": 1700000000,
  "exp": 1700003600,

  "session": {
    "session_id": "sess-uuid-12345",
    "type": "human_initiated",
    "initiator": "user-alice",
    "approval_ref": "approval-uuid-789",
    "max_chain_depth": 5
  },

  "actor_chain": [
    {
      "sub":
        "spiffe://example.com/agent/orchestrator",
      "iss": "https://auth.example.com",
      "iat": 1700000010
    },
    {
      "sub":
        "spiffe://example.com/agent/analyst",
      "iss": "https://auth.example.com",
      "iat": 1700000030
    }
  ],

  "intent_root": "sha256:abc123...",
  "intent_registry":
    "https://intent-log.example.com/sessions/sess-uuid-12345",

  "inference_root": "sha256:xyz789...",
  "inference_proof_type": "tee_h100",
  "inference_registry":
    "https://proof-log.example.com/sessions/sess-uuid-12345"
}
```

6.2. Claim Definitions

Claim	Type	Description
inference_root	string	Merkle root hash of inference chain
inference_proof_type	string	Primary proof algorithm (e.g., groth16, tee_tdx, tee_h100, hybrid)
inference_registry	string	URI for retrieving full inference chain or proofs (REQUIRED)

Table 8

7. Verification Procedures

7.1. Three-Point Verification

A Relying Party receiving a token with all three chains performs a "Three-Point Check":

1. *Identity* (Actor Chain): Is the delegation chain valid? Are all actors authorized?
2. *Integrity* (Intent Chain): Does the content hash trail match the intent_root?
3. *Validity* (Inference Chain): Does the computational proof verify against the inference_root?

7.2. ZKML Proof Verification

To verify a ZKML inference entry:

1. Retrieve the entry from the inference registry.
2. Fetch the verification key from verification_key_registry.
3. Verify hash(verification_key) == verification_key_hash.
4. Run the ZKP verifier: verify(proof, verification_key, public_inputs) where public_inputs = (model_fingerprint, input_hash, output_hash).
5. Verify output_hash matches the corresponding intent chain entry.

7.3. TEE Quote Verification

To verify a TEE attestation entry:

1. Retrieve the entry from the inference registry.
2. Validate the platform certificate chain against the manufacturer's root CA (e.g., Intel, NVIDIA).
3. Check the platform certificate has not been revoked (TCB recovery check).
4. Verify the quote signature using the platform certificate.
5. Verify enclave_measurement matches the expected measurement for the claimed model.
6. Verify report_data binds the quote to the specific input/output hashes.
7. Verify output_hash matches the corresponding intent chain entry.

7.4. Tiered Verification

Inference verification extends the tiered strategy from {{!I-D.draft-mw-spice-intent-chain}}:

Risk Level	Actor Chain	Intent Chain	Inference Chain	Use Case
Low	Sync	Skip	Skip	Read operations
Medium	Sync	Cached proof	Skip	Create/ update
High	Sync	Full	TEE quote check	Financial decisions
Critical	Sync	Full	Full ZKML + TEE	Regulatory, high-stakes

Table 9

8. Policy Enforcement

8.1. Policy Examples

8.1.1. Require Inference Proof for All Agent Outputs

```
require_inference_proofs {
    intent_chain := get_intent_chain(
        input.intent_root)
    inference_chain := get_inference_chain(
        input.inference_root)

    agent_outputs := [i |
        intent_chain[i].type == "non_deterministic"]

    every i in agent_outputs {
        some j
        inference_chain[j].intent_entry_ref == i
    }
}
```

8.1.2. Require TEE for Financial Operations

```
require_tee_for_finance {
    input.action in ["transfer", "approve",
                     "underwrite"]

    inference_chain := get_inference_chain(
        input.inference_root)

    every entry in inference_chain {
        entry.type in ["tee_attestation",
                       "hybrid_proof"]
    }
}
```

8.1.3. Block Specific Model Versions

```
block_deprecated_models {
    inference_chain := get_inference_chain(
        input.inference_root)

    every entry in inference_chain {
        not entry.model_id in
            data.deprecated_models
    }
}
```

8.2. Integration with Policy Engines

The inference chain claims are designed for consumption by policy engines such as Open Policy Agent (OPA). A policy engine SHOULD:

1. Verify actor chain integrity (per {{!I-D.draft-mw-spice-actor-chain}}).
2. Verify intent chain integrity (per {{!I-D.draft-mw-spice-intent-chain}}).
3. Verify inference_root and inference_registry are present and non-empty.
4. Evaluate deployment-specific requirements such as proof type (ZKML vs TEE), model version policies, and hardware platform requirements.

9. Security Considerations

9.1. STRIDE Analysis for Inference Chain

Threat	Attack	Mitigation
Spoofing (Computational)	Model substitution	model_fingerprint binds proof to specific weights
Tampering (Model)	Weight modification	Hash of weights in proof statement
Repudiation	"I didn’t run that model"	Signed inference entries with agent SPIFFE ID
Replay	Reuse old proof for new output	input_hash + output_hash binding + iat timestamp
Environment Spoofing	Claim TEE without using one	TEE quote with hardware-signed measurements

Table 10

9.2. Trust Assumptions

9.2.1. ZKML Proofs

* Trust the cryptographic hardness assumptions of the proof system (e.g., discrete log for Groth16, collision resistance for STARKs).
* Trust the correctness of the arithmetic circuit encoding the neural network.
* DO NOT require trust in any hardware manufacturer.

9.2.2. TEE Attestation

* Trust the hardware manufacturer's root of trust (Intel, AMD, NVIDIA).
* Trust the manufacturer's TCB recovery process (firmware updates, revocation).
* Trust that the TEE hardware correctly isolates the enclave from the host.
* ACKNOWLEDGE the risk of hardware-level attacks (side channels, fault injection) although these require physical access and are outside the typical threat model.

9.3. Verification Key Management

For ZKML proofs, the verification key uniquely identifies the circuit (and therefore the model architecture). Verification keys:

* MUST be published in a discoverable registry (verification_key_registry).
* MUST be versioned alongside model versions.
* SHOULD be pinned by deployers to prevent key substitution attacks.
* MAY be registered in a SCITT transparency log for long-term auditability.

9.4. Proof Freshness

Inference proofs MUST include an iat timestamp. Verifiers SHOULD reject proofs older than a configurable maximum age. For TEE quotes, the platform firmware version SHOULD be checked against the manufacturer's latest TCB baseline to ensure the hardware is patched.

9.5. Credential Isolation

Inference registry entries MUST NOT contain OAuth tokens, bearer credentials, or signing keys. The relationship between tokens and registry entries is one-directional: the token references the registry via the inference_registry URI claim, but the registry MUST NOT store or reference the token. This separation ensures that compromise of the inference registry does not expose bearer credentials.

9.6. ZKML Proof Size Considerations

STARK-based proofs can be tens of kilobytes. The Merkle tree architecture ensures this does not affect token size (only the root hash is in the JWT). However, deployments using STARKs SHOULD consider:

* Pre-computing and caching proofs for frequently-used model/input combinations.

* Using recursive proof composition to compress multiple STARK proofs into a single succinct proof.
* Implementing proof compression techniques specific to their proof system.

9.7. Relationship to Proof of Residency

The inference chain's TEE attestation entries are complementary to, but distinct from, the Proof of Residency (PoR) in actor chain entries ({{!I-D.draft-mw-spice-transitive-attestation}}):

* *PoR* (in actor chain): Proves the agent's IDENTITY is bound to a TEE. Answers "Is this agent running in a secure environment?"
* *TEE Quote* (in inference chain): Proves a specific COMPUTATION ran in a TEE. Answers "Did this model execution happen inside an enclave?"

Both MAY reference the same underlying hardware, but serve different verification purposes. The por_ref field in TEE attestation entries allows cross-referencing.

10. Privacy Considerations

10.1. Model Weight Privacy

ZKML proofs provide zero-knowledge privacy for model weights — the verifier learns that the output was produced by a model with a specific fingerprint, but learns nothing about the actual weights. This is valuable for proprietary models where the weights are confidential.

TEE attestation provides enclave isolation — the model weights are protected from the host OS, but the enclave measurement reveals information about the model binary.

10.2. Input Privacy

Both proof types include input_hash rather than the raw input, providing input privacy by default. For applications requiring stronger guarantees, ZKML proofs can be constructed to hide the input entirely (proving only that "some valid input" produced the output).

10.3. Selective Disclosure

Inference chain entries MAY use Selective Disclosure (SD-JWT) {{!I-D.ietf-oauth-selective-disclosure-jwt}} to hide sensitive fields such as model_id or platform from certain verifiers while maintaining proof integrity.

11. Implementation Guidance

11.1. Current ZKML Landscape

As of this writing, ZKML is practical for models up to approximately 100 million parameters. Active research is scaling this boundary through optimized circuit designs and recursive proof composition.

Deployments SHOULD plan for ZKML scalability improvements and MAY use TEE attestation as a near-term bridge.

11.2. TEE Integration

For TEE-based inference proofs:

1. Deploy the model inside a TEE enclave (e.g., NVIDIA Confidential Computing on H100).
2. At inference time, generate a hardware quote binding the input hash and output hash to the enclave measurement.
3. Append the quote as a TEE attestation entry to the inference registry.
4. The quote’s report_data field MUST contain sha256(input_hash || output_hash) to bind the proof to specific content.

11.3. Scalability Considerations

* *Async ZKML*: ZKML proofs MAY be generated asynchronously after inference. The intent chain entry is appended immediately; the inference chain entry is appended when the proof is ready. The Merkle root is recomputed at token exchange time.
* *Proof Batching*: Multiple inference operations MAY be batched into a single ZKML proof using recursive composition, reducing verification overhead.
* *Quote Caching*: TEE quotes for the same enclave measurement MAY be cached for a configurable period, reducing quote generation overhead.

11.4. Multi-AS Deployments

In deployments involving multiple Authorization Servers, the inference registry is shared across all participating ASes under the same session partition. Each AS appends inference chain entries under the sid partition established at session initiation. The inference_root in each successive token is recomputed over all proof entries accumulated so far — it therefore differs at each hop as the chain grows. This is expected behavior: a relying party receiving a later token will see a larger inference_root than one receiving an earlier token in the same session. ASes do not need to share keys or

coordinate directly — the session partition and append-only log semantics provide the necessary consistency, following the same pattern as the Actor Chain Registry ({{!I-D.draft-mw-spice-actor-chain}}) and Intent Registry ({{!I-D.draft-mw-spice-intent-chain}}).

11.5. Registry Availability

Inference registry unavailability does not affect data-plane operation — the token's AS-signed inference_root is sufficient for request-time policy decisions. Per-entry proof verification is deferred to the audit plane.

However, inference proofs are uniquely valuable because they cannot be regenerated after the fact — the exact model state, input, and execution environment may no longer be available. Deployments SHOULD:

* Replicate inference registry entries across availability zones.
* Use append-only log services designed for high durability (e.g., SCITT transparency logs).
* Define a fail-mode policy: *fail-closed* for regulated workloads (e.g., financial, healthcare), or *fail-open* with verification gap logging for lower-risk workloads.

A federated IAM/IdM platform (e.g., Keycloak, Microsoft Entra, Okta, PingFederate) MAY host the inference registry alongside the Actor Chain Registry and Intent Registry under the same session partition — see {{!I-D.draft-mw-spice-actor-chain}} Section "Registry Hosting" for detailed requirements.

11.5.1. Proof Size and Storage

Inference proof payloads are significantly larger than actor chain entries (~0.5-1KB) or intent chain entries (_(0.5-1KB):)

Proof Type	Typical Size	Storage Implication
TEE attestation quote	_(2-4KB)	Compatible with most IAM data stores
Groth16 ZK proof	~200 bytes	Compatible with most IAM data stores
STARK proof	_50KB	May exceed IAM data store blob limits

Table 11

Most IAM/IdM platforms are not designed to store large binary blobs. Deployments that include STARK proofs SHOULD either:

* Use a dedicated object store (e.g., S3, GCS, Azure Blob Storage) for proof payloads, with the IAM platform serving as the registry index and access control layer.
* Or use a SCITT transparency log that natively supports variable-size entries.

11.6. Performance Guidance: TEE vs ZKML

Deployments SHOULD select proof types based on their latency requirements:

* *Real-time applications* (chat, interactive agents, API serving): Use TEE attestation quotes exclusively. TEE proof generation adds millisecond-level overhead and is compatible with production-scale LLMs (100B+ parameters).
* *Batch/offline applications* (document generation, model evaluation, regulatory reporting): ZKML proofs provide mathematical certainty independent of hardware trust. Proof generation latency (minutes to hours) is acceptable when not on the critical path.
* *Hybrid deployments*: Use TEE quotes for real-time inference and generate ZKML proofs asynchronously for high-value operations, appending them to the inference chain after the fact. The Merkle root is recomputed at the next token exchange.

12. Design Rationale: Merkle Root in Token

The inference chain follows the same Merkle root architecture as the intent chain (see {{!I-D.draft-mw-spice-intent-chain}} Design Rationale for the detailed comparison). The same trade-offs apply, with additional motivation: inference proofs are large (STARK proofs ~50KB, TEE quotes ~2-4KB), making inline embedding in tokens impractical. The Merkle root enables selective verification of individual proofs using O(log n) sibling hashes, which is critical for the tiered verification strategy.

13. IANA Considerations

13.1. JWT Claim Registration

This document requests registration of the following claims in the "JSON Web Token Claims" registry established by {{!RFC7519}}:

* *Claim Name*: inference_root
* *Claim Description*: Merkle root hash of the inference chain for computational provenance verification.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]
* *Claim Name*: inference_proof_type
* *Claim Description*: Primary proof algorithm used in the inference chain.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]
* *Claim Name*: inference_registry
* *Claim Description*: URI of the inference registry for proof retrieval.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]

13.2. CWT Claim Registration

This document requests registration of the following claims in the "CBOR Web Token (CWT) Claims" registry established by {{!RFC8392}}:

* *Claim Name*: inference_root
* *Claim Description*: Merkle root hash of the inference chain.
* *CBOR Key*: TBD (e.g., 60)
* *Claim Type*: tstr
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]
* *Claim Name*: inference_registry
* *Claim Description*: URI of the inference registry for proof retrieval.
* *CBOR Key*: TBD (e.g., 61)
* *Claim Type*: tstr
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]
* *Claim Name*: inference_proof_type
* *Claim Description*: Primary proof algorithm used in the inference chain.
* *CBOR Key*: TBD (e.g., 62)
* *Claim Type*: tstr
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]

Appendix A. Full Token Example (Three-Chain Governance)

```
{
  "iss": "https://auth.example.com",
  "sub": "user-alice",
  "aud": "https://api.example.com",
  "jti": "tok-fff-67890",
  "iat": 1700000000,
  "exp": 1700003600,

  "session": {
    "session_id": "sess-uuid-12345",
    "type": "human_initiated",
    "initiator": "user-alice",
    "approval_ref": "approval-uuid-789",
    "max_chain_depth": 5
  },

  "actor_chain": [
    {
      "sub":
        "spiffe://example.com/agent/orchestrator",
      "iss": "https://auth.example.com",
      "iat": 1700000010,
      "scope": "ticket:*",
      "chain_digest": "sha256:aaa...",
      "chain_sig": "eyJhbGci..."
    },
    {
      "sub":
        "spiffe://example.com/agent/analyst",
      "iss": "https://auth.example.com",
      "iat": 1700000030,
      "scope": "analysis:run",
      "chain_digest": "sha256:bbb...",
      "chain_sig": "eyJhbGci..."
    }
  ],

  "intent_root": "sha256:abc123...",
  "intent_registry":
    "https://intent-log.example.com/sessions/sess-uuid-12345",

  "inference_root": "sha256:xyz789...",
  "inference_proof_type": "tee_h100",
  "inference_registry":
    "https://proof-log.example.com/sessions/sess-uuid-12345"
}
```

Authors' Addresses

Ram Krishnan
JPMorgan Chase & Co
Email: ramkri123@gmail.com

A Prasad
Oracle
Email: a.prasad@oracle.com

Diego R. Lopez
Telefonica
Email: diego.r.lopez@telefonica.com

Srinivasa Addepalli
Aryaka
Email: srinivasa.addepalli@aryaka.com