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 Intent Chain for AI Agent Content Provenance
draft-mw-spice-intent-chain-00

Abstract

This document defines the intent_chain claim as a companion to the actor_chain claim defined in {{!I-D.draft-mw-spice-actor-chain}}. While the actor chain addresses delegation provenance (WHO delegated to whom), the intent chain addresses content provenance (WHAT was produced and HOW it was transformed).

In AI agent workflows, content flows through multiple processing stages including AI agents and filters. The intent chain provides a cryptographically verifiable, tamper-evident record of this content journey. The full intent chain is stored as ordered logs, with only the Merkle root included in the OAuth token for efficiency.

Together, the actor chain and intent chain provide complete governance for autonomous AI agent systems, addressing Spoofing, Tampering, Repudiation, and Elevation of Privilege threats in the STRIDE threat model.

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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 September 2026.

Table of Contents

1. Introduction

1.1. The Problem: Content Provenance Gap

The Actor Chain extension to {{!RFC8693}} (defined in {{!I-D.draft-mw-spice-actor-chain}}) addresses the Delegation Auditability Gap by providing cryptographic proof of the actual delegation path between AI agents. However, it does not address a complementary gap: *Content Provenance*.

In AI agent workflows, content flows through multiple processing stages:

Agent A -> Filter -> Filter -> Agent B -> Filter -> Agent C -> Tool

Each stage may transform the content. AI agent outputs are inherently non-deterministic and cannot be trusted without validation. Filters (both AI-based and rule-based) transform this content before it reaches the next stage.

For complete governance, systems require proof of:

* *What* each AI agent originally produced (raw output)
* *How* each filter transformed the content
* *Whether* transformations were deterministic (reproducible) or non-deterministic (AI-based)
* *The complete chain* of content transformations within a session

Without this proof, the content transformation history cannot be reconstructed for audit or dispute resolution. Consider the following repudiation scenario:

1. Agent A produces a response containing a harmful instruction (e.g., caused by prompt injection).
2. The response passes through an AI guardrail filter, which fails to catch the harmful content.
3. Agent B acts on the harmful instruction, causing damage.
4. During investigation, Agent A's operator claims "Agent A never produced that output — it must have been injected by a downstream filter."

Without cryptographic proof binding Agent A's identity to its specific output hash, this claim cannot be disproven. The intent chain solves this by requiring every agent to sign input_hash + output_hash via intent_sig, creating non-repudiable evidence of what each agent received and produced.

1.2. Relationship to Actor Chain and Inference Chain

This specification is part of a three-axis "Truth Stack" for AI agent governance:

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 (this document)	Content	WHAT was produced and transformed?	Repudiation, Tampering
Inference Chain ({{!I-D.draft-mw-spice-inference-chain}})	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 Chain Only*: Real-time authorization, access control
* *Intent Chain Only*: Content audit, debugging, filter validation
* *Inference Chain Only*: Computational integrity verification for single-agent systems
* *Actor + Intent*: Full content governance, dispute resolution, regulatory compliance
* *All Three*: Complete "Truth Stack" — identity, content, and computational provenance

1.3. Design Goals

The intent chain is designed with the following goals:

1. *Content Provenance*: Cryptographic proof of what each agent produced
2. *Transformation Tracking*: Record of how filters modified content
3. *Tamper Evidence*: Merkle tree structure prevents undetected modification
4. *Efficiency*: Only Merkle root in token; full chain in logs
5. *Scalability*: Append-only logs scale horizontally
6. *Modularity*: Usable independently or with actor chain
7. *Standards Alignment*: Compatible with OAuth 2.0, JWT, SPIFFE

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.

Intent Chain: An ordered sequence of Intent Chain Entries representing the complete content journey from originating agent through filters to final output within a session.

Intent Chain Entry: A record identifying a single content transformation, including the agent identity, entry type, content hashes, and cryptographic signature.

AI Agent: An autonomous decision-maker that produces content and can delegate authority. AI agents appear in both the actor chain (for delegation) and the intent chain (for output provenance).

Non-Deterministic Filter: A processor (typically AI-based) whose output cannot be reproduced from its input. Examples include AI guardrails, LLM-based content rewriters, and semantic classifiers. Both input and output MUST be signed.

Deterministic Filter: A processor whose output can be reproduced from its input and rules. Examples include schema validators, regex sanitizers, and bounds checkers. Output can be re-derived for verification.

Intent Root: The Merkle root hash of the complete intent chain, included in the OAuth token.
Intent Registry: An append-only ordered log storing the full intent chain entries, partitioned by session.
Actor Chain Registry: The append-only ordered log storing the full per-actor signature evidence for the actor chain, as defined in {{!I-D.draft-mw-spice-actor-chain}}. Referenced by the actor_chain_registry claim in the token.

3. Architecture Overview

3.1. Three-Layer Governance Model

The governance model consists of three layers:

* *Session*: Root of trust and lifecycle management. Initiated by human approval or system authorization. Contains subject, expiry, and approval reference.
* *Actor Chain (WHO)*: Contains AI agents only. Full chain stored in token. Addresses Spoofing, Repudiation, and Elevation of Privilege. Defined in {{!I-D.draft-mw-spice-actor-chain}}.
* *Intent Chain (WHAT)*: Contains AI agents and filters. Full chain stored in ordered logs; Merkle root in token. Addresses Repudiation and Tampering. Defined in this document.
* *Inference Chain (HOW)*: Contains per-inference computational proofs. Full proofs stored in ordered logs; Merkle root in token. Addresses Computational Spoofing and Model Tampering. Defined in {{!I-D.draft-mw-spice-inference-chain}}.

3.2. Session as Root of Trust

Every governance chain traces back to a session. The session provides:

* *Origin*: Who or what initiated the workflow
* *Lifecycle*: When the session expires
* *Revocation Point*: Revoking session invalidates all activity
* *Audit Boundary*: Session defines the unit of audit

Session information is captured in standard OAuth token claims:

Claim	Purpose
sub	Session subject (human or system initiator)
sid	Session identifier — stable across token exchanges within a session. Equals session.session_id. Defined as a top-level claim in {{!I-D.draft-mw-spice-actor-chain}}
jti	Token identifier (unique per token exchange)
iat	Session start time
exp	Session expiry time

Table 3

3.3. Actor Chain (WHO) - Reference

The actor chain is defined in {{!I-D.draft-mw-spice-actor-chain}}. Key properties:

* Contains AI agents only (autonomous decision-makers)
* Tracks delegation of authority
* Full chain included in token
* Addresses Spoofing, Repudiation, Elevation of Privilege

This document assumes familiarity with the actor chain specification.

3.4. Intent Chain (WHAT) - This Document

The intent chain is defined in this document. Key properties:

* Contains AI agents AND filters
* Tracks content production and transformation
* Merkle root in token; full chain in ordered logs
* Addresses Repudiation and Tampering

3.5. STRIDE Threat Model Coverage

Threat	Mitigation	Component
S - Spoofing	Cryptographic identity, signed chain entries	Actor Chain

T - Tampering	Merkle tree integrity, append-only logs	Intent Chain
R - Repudiation	Signed delegation (Actor) + Signed outputs (Intent)	Both
I - Information Disclosure	Selective disclosure (SD-JWT)	Both (optional)
D - Denial of Service	Session expiry, rate limits	Session + Infrastructure
E - Elevation of Privilege	Scope attenuation, policy enforcement	Actor Chain

Table 4

4. Intent Chain Definition

4.1. Entry Types

The intent chain contains two types of entries:

Entry Type	Determinism	Signed Fields	Type-Specific Fields
Non-Deterministic (AI agent output, AI-based filter)	Non-deterministic	input_hash + output_hash	model_info (optional)
Deterministic (rule-based filter)	Deterministic	input_hash + output_hash	rule_id, rule_hash

Table 5

All entry types REQUIRE both input_hash and output_hash. This uniform structure ensures that every consecutive pair satisfies entry[i].output_hash == entry[i+1].input_hash, creating a complete content provenance chain. The cost is approximately 40 bytes per entry in the ordered logs — not in the token itself, which carries only the Merkle root regardless of entry count.

4.2. Non-Deterministic Entries

Non-deterministic entries record outputs from AI agents or AI-based filters whose output cannot be reproduced from the input alone.

Examples:

* AI agent outputs (orchestrator, planner, tool agent)
* AI guardrails (Llama Guard, NeMo Guardrails)
* LLM-based content rewriters
* Semantic classifiers

Properties:

* Sub is an AI agent or AI-based filter (agents also appear in actor chain)
* Output is non-deterministic (cannot be reproduced)
* input_hash and output_hash MUST be recorded and signed

Agent Output Example:

```
{
  "type": "non_deterministic",
  "sub": "spiffe://example.com/agent/orchestrator",
  "input_hash": "sha256:fff000...",
  "output_hash": "sha256:abc123...",
  "iat": 1700000010,
  "intent_digest": "sha256:...",
  "intent_sig": "eyJhbGci..."
}
```

AI Filter Example:

```
{
  "type": "non_deterministic",
  "sub": "spiffe://example.com/filter/ai-guardrail",
  "filter_version": "v2.1",
  "input_hash": "sha256:abc123...",
  "output_hash": "sha256:def456...",
  "model_info": {
    "model": "llama-guard-3",
    "categories": ["violence", "pii", "prompt_injection"]
  },
  "iat": 1700000015,
  "intent_digest": "sha256:...",
  "intent_sig": "eyJhbGci..."
}
```

4.3. Deterministic Entries

Deterministic filter entries record transformations by rule-based filters whose output can be reproduced from the input and rules.

Examples:

* Schema validators (JSON Schema)
* Regex sanitizers (XSS removal)
* Bounds checkers (amount limits)
* PII redactors (pattern-based)

Properties:

* Sub is a rule-based filter
* Output CAN be reproduced from input + rules
* input_hash and output_hash MUST be recorded and signed
* rule_id and rule_hash are type-specific signed fields in the log entry, enabling independent re-verification
* Output can be re-derived by re-applying the rule to the input

Structure:

```
{
  "type": "deterministic",
  "sub": "spiffe://example.com/filter/schema-validator",
  "filter_version": "v1.0",
  "input_hash": "sha256:def456...",
  "output_hash": "sha256:ghi789...",
  "rule_id": "ticket-schema-v2",
  "rule_hash": "sha256:rrr...",
  "transform_applied": {
    "fields_validated": ["title", "priority", "amount"],
    "fields_modified": ["priority"],
    "modification": {
      "priority": {
        "from": "critical",
        "to": "medium",
        "reason": "bounds_exceeded"
      }
    }
  },
  "reproducible": true,
  "iat": 1700000016,
  "intent_digest": "sha256:...",
  "intent_sig": "eyJhbGci..."
}
```

4.4. Entry Structure

All intent chain entries share common fields:

Field	Type	Required	Description
type	string	REQUIRED	Entry type: non_deterministic, deterministic
sub	string	REQUIRED	SPIFFE ID of the agent or filter
input_hash	string	REQUIRED	SHA-256 hash of the input content
output_hash	string	REQUIRED	SHA-256 hash of the output content
iat	number	REQUIRED	Timestamp when entry was created
intent_digest	string	REQUIRED	Hash of the canonically serialized entry for Merkle leaf computation
intent_sig	string	REQUIRED	Signature over intent_digest using the agent’s or filter’s private key

Table 6

4.4.1. intent_digest Computation

The intent_digest field is computed as the SHA-256 hash of the canonically serialized entry, excluding the intent_digest and intent_sig fields themselves. This hash serves as the leaf node in the Merkle tree.

For an entry E with fields {type, sub, input_hash, output_hash, iat, ...}:

```
intent_digest = SHA-256(canonical_json(E \ {intent_digest, intent_sig}))
```

Where canonical_json follows JSON Canonicalization Scheme {{JCS}} (RFC 8785) to ensure deterministic serialization.

The intent_sig (when REQUIRED) is computed over the intent_digest value using the agent's private key:

intent_sig = Sign(agent_key, intent_digest)

This two-step process ensures that: (a) the digest is stable and independent of signature ordering, and (b) the signature covers all content-relevant fields of the entry.

Additional fields by entry type:

Field	Entry Types	Description
filter_version	Filters	Version of filter
rule_id	Deterministic	Identifier of rule applied
rule_hash	Deterministic	Hash of rule definition
model_info	Non-deterministic	AI model information
transform_applied	Filters	Details of transformation
reproducible	Deterministic	Boolean indicating reproducibility

Table 7

5. Storage Architecture

5.1. Intent Registry (Ordered Logs)

The intent registry stores immutable intent chain entries as ordered logs.

Contents:

* Non-deterministic entries (AI agent outputs, AI-based filters)
* Deterministic entries (rule-based filters)

> | Intent registry entries MUST NOT contain OAuth tokens, bearer
> | credentials, or signing keys. Entries contain only content
> | hashes, metadata, agent identities, and entry-level signatures.
> | The token references the registry via the intent_registry claim;
> | the registry MUST NOT store or reference the token itself.

Properties:

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

Implementations SHOULD use an append-only log that supports partitioned, ordered retrieval by the token's session.session_id claim and provides tamper-evident guarantees (e.g., via hash chaining or inclusion proofs).

Log Structure:

```
{
  "session_id": "sess-uuid-12345",
  "offset": 0,
  "entry": {
    "type": "non_deterministic",
    "sub": "spiffe://example.com/agent/A",
    "input_hash": "sha256:prompt...",
    "output_hash": "sha256:abc...",
    "iat": 1700000010,
    "intent_digest": "sha256:...",
    "intent_sig": "eyJ..."
  }
}
```

5.1.1. Relationship Between session_id and jti

The session_id is a stable identifier for the end-user interaction. It remains constant as the delegation chain grows through multiple token exchanges, each of which produces a new token with a distinct jti:

```
User session: sess-uuid-12345

  Token Exchange 1 (jti: "tok-aaa")
    User → Agent A
    Intent entries: offset 0 (Agent A output)

  Token Exchange 2 (jti: "tok-bbb")
    Agent A → Agent B
    Intent entries: offset 1 (filter), offset 2 (Agent B output)

  Token Exchange 3 (jti: "tok-ccc")
    Agent B → Agent C
    Intent entries: offset 3 (filter), offset 4 (Agent C output)
```

All intent chain entries share session_id: "sess-uuid-12345" regardless of which token exchange produced them. The session_id is carried forward during each token exchange as part of the session claim. During forensic verification, the investigator retrieves all entries for a session_id to reconstruct the complete content journey.

5.2. Merkle Tree Construction

The Merkle tree is constructed from ordered log entries. Leaf nodes are the SHA-256 hashes of canonically serialized intent chain entries. Internal nodes are the SHA-256 hash of the concatenation of their two child hashes. When a level has an odd number of nodes, the last node is promoted to the next level.

See Appendix A for a visual depiction and reference construction algorithm.

5.3. Merkle Root in Token

Only the Merkle root is included in the OAuth token:

```
{
  "intent_root": "sha256:abc123def456...",
  "intent_alg": "sha256",
  "intent_registry": "https://intent-log.example.com"
}
```

Field	Type	Required	Description
intent_root	string	REQUIRED	Merkle root hash of intent chain
intent_alg	string	OPTIONAL	Hash algorithm (default: sha256)
intent_registry	string	REQUIRED	URI of intent registry for proof retrieval

Table 8

6. Token Structure

6.1. Combined Token Format

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

```
{
  "iss": "https://auth.example.com",
  "sub": "user-alice",
  "aud": "https://api.example.com",
  "jti": "tok-aaa-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/support",
      "iss": "https://auth.example.com",
      "iat": 1700000030
    }
  ],

  "intent_root": "sha256:abc123def456789...",
  "intent_registry":
    "https://intent-log.example.com/sessions/sess-uuid-12345"
}
```

6.2. Claim Definitions

6.2.1. Session Claims

Claim	Type	Description
sid	string	Session identifier — stable across token exchanges. Equals session.session_id. Defined as a top-level claim in {{!I-D.draft-mw-spice-actor-chain}}

session.session_id	string	Stable identifier for the end-user interaction, assigned by the AS on first token issuance and carried forward during subsequent token exchanges. MUST equal the top-level sid claim
session.type	string	Session type: human_initiated, system_initiated, scheduled
session.initiator	string	Identity of session initiator
session.approval_ref	string	Reference to approval record
session.max_chain_depth	number	Maximum allowed delegation depth

Table 9

6.2.2. Actor Chain Claims

Defined in {{!I-D.draft-mw-spice-actor-chain}}.

6.2.3. Intent Chain Claims

Claim	Type	Description
intent_root	string	Merkle root hash of intent chain
intent_alg	string	Hash algorithm used (default: sha256)
intent_registry	string	URI for retrieving full chain or proofs (REQUIRED)

Table 10

6.3. Examples

6.3.1. Minimal Token (Actor Chain Only)

```
{
  "iss": "https://auth.example.com",
  "sub": "user-alice",
  "jti": "tok-bbb-12345",
  "iat": 1700000000,
  "exp": 1700003600,

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

6.3.2. Minimal Token (Intent Chain Only)

```
{
  "iss": "https://auth.example.com",
  "sub": "user-alice",
  "jti": "tok-ccc-12345",
  "iat": 1700000000,
  "exp": 1700003600,

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

6.3.3. Full Token (Both Chains)

See Section 6.1.

7. Verification Procedures

7.1. Request-Time Policy Checks

At request time, the Relying Party performs lightweight checks on the intent chain metadata in the token. Full chain verification is unnecessary on the hot path because:

* The content has already been produced; verifying signatures cannot undo it.
* The Relying Party has the token, not the raw content, so it cannot cross-check content hashes.
* O(n) signature verification per request adds latency without improving authorization decisions.

The Relying Party SHOULD:

1. Verify the JWT outer signature (covers intent_root as a signed claim).
2. Check that intent_root and intent_registry are present (policy: "intent chain coverage required").
3. Apply policy rules against intent chain entry types fetched from the registry (e.g., "must include at least one deterministic entry").

The tiered verification table reflects the appropriate level of intent chain checking based on risk:

Risk Level	Actor Chain	Intent Chain	Use Case
Low	Verify JWT signature	Check intent_root present	Read operations
Medium	Verify JWT signature	Async policy check on entry types	Create/ update
High	Verify JWT signature + actor sigs	Full forensic verification	Delete, transfer, admin

Table 11

7.2. Forensic Verification

The primary value of the intent chain is post-hoc troubleshooting and dispute resolution. When an incident occurs, an auditor or investigator performs full chain verification to determine which agent caused the problem.

7.2.1. Token Archival

Forensic verification requires two inputs: the original JWT (containing intent_root) and the intent chain entries from the registry. The intent registry stores chain entries but does not store the token itself.

To enable forensic analysis, tokens SHOULD be archived by one or more of the following:

* *Relying Parties*: Archive tokens at the point of presentation (most common).
* *Audit services*: A dedicated audit service receives a copy of each token for compliance purposes.
* *Authorization Servers*: The AS MAY maintain a log of issued tokens indexed by jti.

Archived tokens MUST be stored securely and access-controlled, as they contain identity and delegation information.

7.2.2. Full Chain Verification

To perform a complete forensic investigation:

1. *Retrieve inputs*: Extract intent_root and intent_registry from the archived token. Fetch the full intent chain from the registry using the session_id.
2. *Verify Merkle integrity*: Rebuild the Merkle tree from all entries. Compare the computed root with intent_root from the token. If they do not match, the chain has been tampered with.
3. *Verify signatures*: For each entry, verify intent_sig over intent_digest using the sub's public key (discoverable via SPIFFE trust bundle or JWKS). A failed signature indicates a forged entry.
4. *Verify chain linkage*: For each consecutive pair, verify entry[i].output_hash == entry[i+1].input_hash. A broken link indicates content was modified between steps without being recorded.
5. *Re-derive deterministic outputs*: For deterministic entries, retrieve the rule definition matching rule_hash, re-apply it to the content matching input_hash, and verify the output matches output_hash. A mismatch indicates the filter did not behave as recorded.
6. *Identify the fault point*: If step 4 reveals a broken link at position k, the content was corrupted between entry[k] and entry[k+1]. If step 3 reveals a failed signature at position k, entry[k] was forged. The sub field of the faulty entry identifies the responsible agent.

7.2.3. Cross-Chain Correlation

When used together with the actor chain, forensic verification can answer both *WHO* and *WHAT*:

1. For each intent chain entry of type non_deterministic: verify that entry.sub appears in the actor_chain of the same token.

2. Verify that entry.iat falls within the actor's active window (between the actor's own iat and the next actor's iat in the chain).
3. A mismatch indicates an unregistered agent produced content — the intent chain records output from an identity not present in the delegation chain.

7.2.4. Dispute Resolution Workflow

When a dispute arises (e.g., "Agent A did not produce that harmful output"):

1. Retrieve the archived token and full intent chain.
2. Locate entries where sub matches Agent A's SPIFFE ID.
3. Verify Agent A's intent_sig on each of those entries. A valid signature proves Agent A attested to producing that specific output_hash.
4. Check input_hash of Agent A's entry to verify what Agent A received as input.
5. If Agent A's output was modified by a downstream filter, the filter's entry shows input_hash matching Agent A's output_hash and a different output_hash — proving the filter made the change, not Agent A.

7.2.5. Single Entry Verification (Merkle Proof)

To verify a single entry without fetching the full chain:

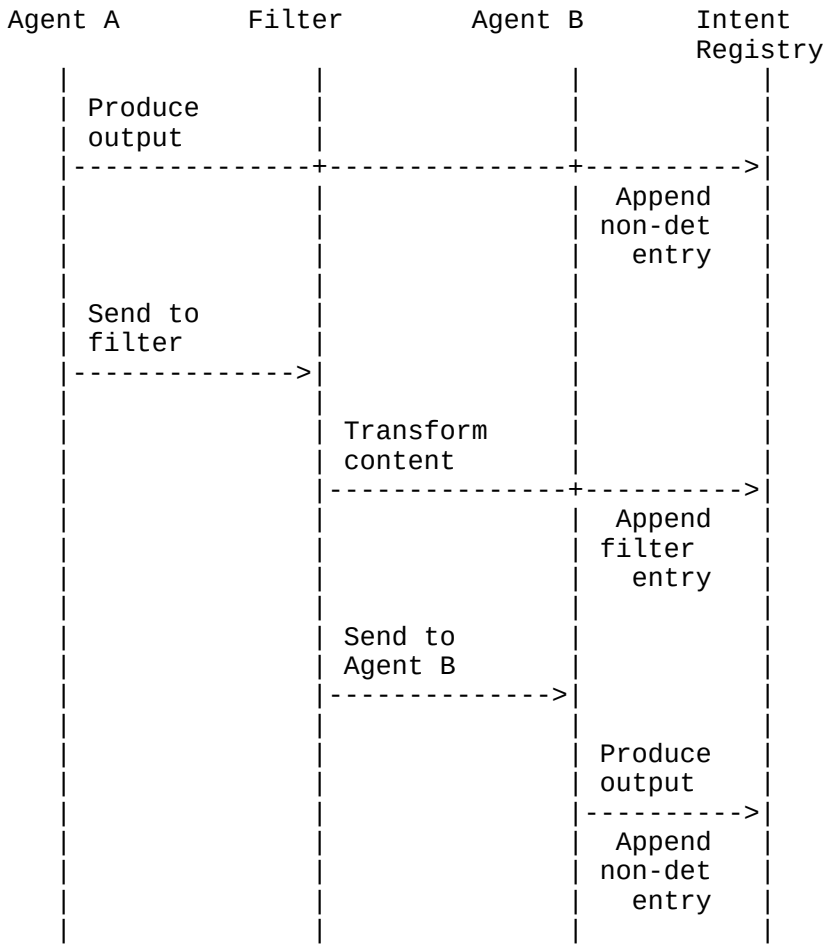
1. Extract intent_root from the token.
2. Request a Merkle proof for entry[i] from the registry.
3. The registry returns the entry and sibling hashes on the path to the root.
4. Compute the hash of the entry and the path to the root using sibling hashes.
5. Compare the computed root with intent_root.

Proof Size: O(log n) where n is the number of entries.

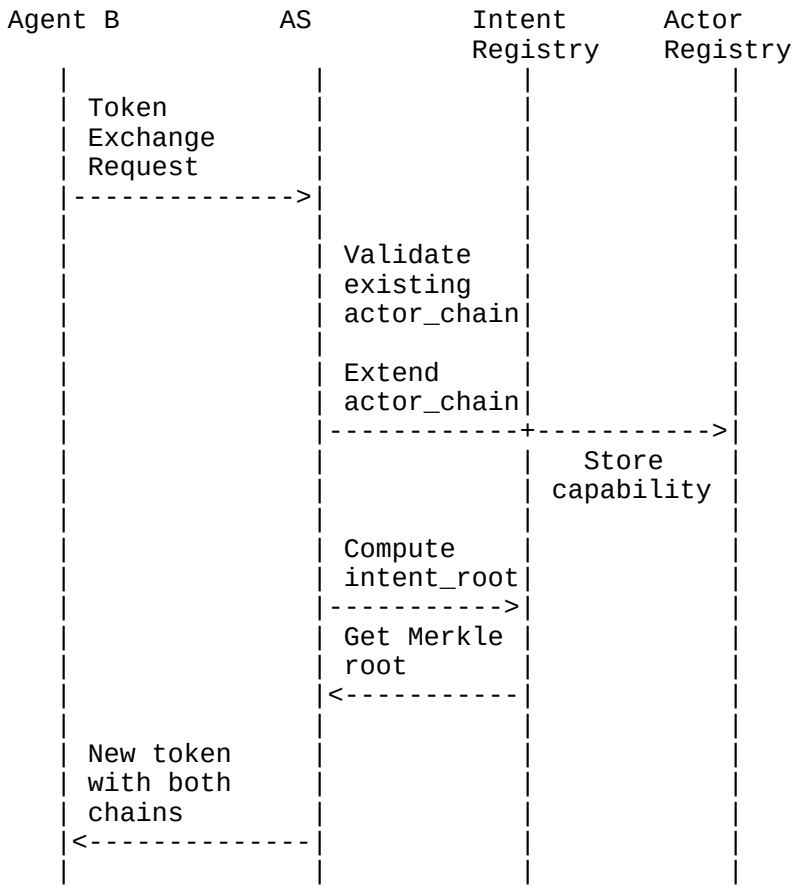
```
{
  "entry": {
    "type": "non_deterministic",
    "sub": "spiffe://example.com/agent/A",
    "input_hash": "sha256:prompt...",
    "output_hash": "sha256:abc...",
    "iat": 1700000010
  },
  "proof": {
    "index": 0,
    "siblings": [
      {"position": "right", "hash": "sha256:111..."},
      {"position": "right", "hash": "sha256:222..."},
      {"position": "left", "hash": "sha256:333..."}
    ]
  }
}
```

8. Operational Flows

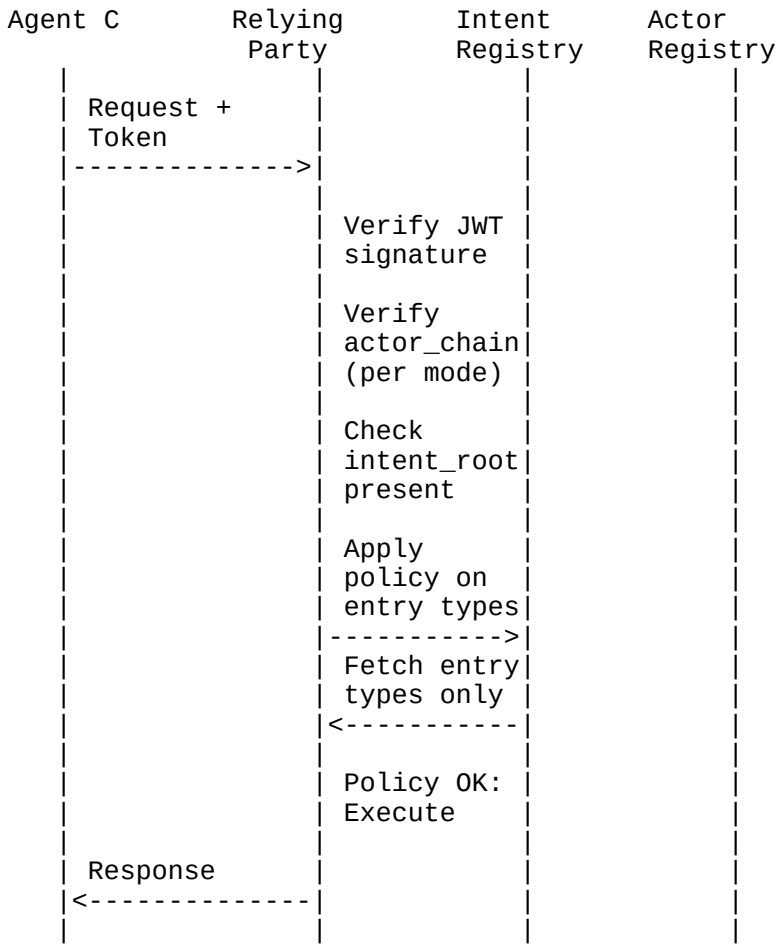
8.1. Intent Chain Construction



8.2. Token Exchange Integration



8.3. Request-Time Check at Relying Party



9. Policy Enforcement

9.1. Policy Examples

9.1.1. Require All Outputs Filtered

Every agent output must be followed by at least one filter:

```
require_filtered_outputs {
    intent_chain := get_intent_chain(input.intent_root)

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

    every i in agent_outputs {
        # Next entry must be a filter (if not last)
        i < count(intent_chain) - 1
        intent_chain[i + 1].type != "non_deterministic"
    }
}
```

9.1.2. Require Non-Deterministic Filter for AI Outputs

AI agent outputs must pass through an AI guardrail:

```
require_ai_guardrail {
    intent_chain := get_intent_chain(input.intent_root)

    every i, entry in intent_chain {
        entry.type == "non_deterministic" implies {
            # Must be followed by an entry with AI guardrail model
            some j
            j > i
            intent_chain[j].model_info.model ==
                "llama-guard-3"
        }
    }
}
```

9.1.3. Verify Specific Transformation Applied

Sensitive fields must be sanitized:

```
require_pii_redaction {
    intent_chain := get_intent_chain(input.intent_root)

    some i
    intent_chain[i].type == "deterministic"
    intent_chain[i].rule_id == "pii-redaction-v1"
}
```

9.2. Integration with Policy Engines

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

1. Validate session expiry and revocation status.
2. Verify actor chain integrity (per {{!I-D.draft-mw-spice-actor-chain}}).
3. Verify intent_root and intent_registry are present and non-empty.
4. Evaluate deployment-specific requirements against the intent chain entries (e.g., requiring filtered outputs, specific guardrail models, or PII redaction).

10. Security Considerations

10.1. STRIDE Analysis

Threat	Mitigation	Mechanism
Spoofing	Cryptographic identity	Actor chain, SPIFFE IDs
Tampering	Merkle tree integrity	Append-only logs, Merkle root in JWT
Repudiation	Signed outputs	intent_sig on agent and filter entries
Information Disclosure	Selective disclosure	SD-JWT, content hashes not content
Denial of Service	Session lifecycle	Expiry, rate limits
Elevation of Privilege	Scope attenuation	Actor chain, policy enforcement

Table 12

10.2. Signing Requirements by Entry Type

Entry Type	Required Signatures	Rationale
non_deterministic	intent_sig over intent_digest (REQUIRED)	Non-reproducible; signs input_hash + output_hash to prove what was received and produced
deterministic	intent_sig over	Reproducible; signs input_hash + output_hash.

		intent_digest (REQUIRED)	Type-specific fields (rule_id, rule_hash) enable independent re-verification

Table 13

10.3. Replay Protection

Each intent chain entry includes an iat (issued-at) timestamp. The session-scoped partitioning of the intent registry prevents cross-session replay. Additionally, the Merkle root is bound to the JWT via the intent_root claim, and the JWT itself carries exp and jti claims, providing token-level replay protection.

10.4. Chain Integrity

The Merkle tree structure provides tamper evidence for the intent chain. Any modification to an entry changes its leaf hash, which propagates up the tree, changing the Merkle root. Since the Merkle root is included in the signed JWT, any tampering with the intent chain entries is detectable.

The append-only property of the intent registry provides additional protection: entries cannot be deleted or modified after creation.

10.5. Credential Isolation

Intent 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 intent_registry URI claim, but the registry MUST NOT store or reference the token. This separation ensures that compromise of the intent registry does not expose bearer credentials that could be used for unauthorized access.

11. Privacy Considerations

11.1. Selective Disclosure

The intent chain contains content hashes rather than actual content. This provides provenance without exposing the content itself. When combined with SD-JWT {{!I-D.ietf-oauth-selective-disclosure-jwt}}, individual intent chain entries can be selectively disclosed to different verifiers.

11.2. Content Hash vs Content Storage

The intent chain stores SHA-256 hashes of content, not the content itself. This design choice provides:

* *Provenance without exposure*: Verifiers can confirm that specific content was produced without seeing the content.
* *Reduced log size*: Hashes are fixed-size regardless of content size.
* *Privacy by default*: Content is not exposed through the intent chain; access to the original content requires separate authorization.

| *Pre-image resistance caveat*: SHA-256 is pre-image resistant for arbitrary-length inputs, but short, low-entropy content (e.g., a 16-digit account number, a boolean flag, or a short enumerated value) may be vulnerable to brute-force guessing. An attacker who knows the hash and the input domain can enumerate all possible inputs and find the one that matches. Deployments handling short, structured content SHOULD salt content before hashing or use SD-JWT to selectively redact content hashes from specific verifiers.

12. Implementation Guidance

12.1. Intent Registry Implementation

The intent registry stores immutable intent chain entries. Recommended properties:

* Append-only log structure
* Partitioned by session.session_id for isolation
* Configurable retention period
* Merkle root computation triggered on append or at token exchange time

A federated IAM/IdM platform (e.g., Keycloak, Microsoft Entra, Okta, PingFederate) MAY host the intent registry alongside the Actor Chain Registry ({{!I-D.draft-mw-spice-actor-chain}}), since the Authorization Server already mediates token exchanges and can append intent chain entries as a side-effect. Most enterprise IAM/IdM platforms support configurable data stores that can be configured for append-only semantics — see {{!I-D.draft-mw-spice-actor-chain}} Section "Registry Hosting" for detailed requirements.

12.2. Multi-AS Deployments

In deployments involving multiple Authorization Servers (e.g., federated enterprise environments where different ASes serve different organizational domains), the intent registry is shared across all participating ASes. Each AS appends intent chain entries to the same session-partitioned registry, identified by the sid claim carried in the token. This works without coordination between ASes because:

* The sid value is established at session initiation and carried forward unchanged through all token exchanges.
* Each AS appends entries atomically under the session's sid partition.
* The Merkle root is recomputed at each token exchange time over all entries accumulated so far (by any AS).
* The resulting intent_root in the token therefore differs at each hop — each successive AS produces a larger Merkle root reflecting the growing chain. This is expected behavior: a growing root is the normal consequence of an append-only chain and indicates that additional intent entries have been recorded.

This enables cross-domain content provenance tracking without requiring ASes to share keys or coordinate directly — the session partition and append-only log semantics provide the necessary consistency.

12.3. Scalability Considerations

* *Log Partitioning*: Session-based partitioning ensures that intent chains for different sessions are isolated and can be processed in parallel.
* *Merkle Root Caching*: Computed Merkle roots SHOULD be cached to avoid recomputation on every token exchange.
* *Proof Materialization*: Merkle proofs for recent entries SHOULD be pre-computed and cached for O(1) retrieval.

12.4. Operational Recommendations

* *Retention Policy*: Intent chain logs SHOULD be retained for the maximum audit window required by the deployment's regulatory environment.
* *Monitoring*: Operators SHOULD monitor intent chain append latency and Merkle root computation time.
* *Backup*: Intent chain logs SHOULD be replicated across availability zones for durability.

12.5. Registry Availability

Intent registry unavailability does not affect data-plane operation — the token's AS-signed intent_root is sufficient for request-time policy decisions (e.g., "intent chain coverage required"). Per-entry forensic verification is deferred to the audit plane and is not required on the hot path.

However, if the registry is permanently lost, forensic verification becomes impossible. Deployments SHOULD:

* Replicate intent registry entries across availability zones.
* Use append-only log services designed for high durability (e.g., SCITT transparency logs).
* Retain archived tokens (containing intent_root) separately from intent chain entries, so that Merkle root commitments survive independently of the registry.
* Define a fail-mode policy: *fail-closed* (reject tokens whose intent chains cannot be verified) for high-risk operations, or *fail-open* (accept the AS-signed token and log the verification gap) for low-risk operations.

13. Design Rationale: Merkle Root in Token

The intent chain uses a Merkle root in the token rather than embedding the full chain inline. The following table summarizes the trade-offs:

Approach	Token Size	Verification	Privacy	Selective Verify
A. Full chain in token	O(n) — grows per entry	Inline, zero latency	Poor — all entries exposed	All-or-nothing
B. Merkle root in token	O(1) — ~64 bytes	O(log n) per entry	Good — selective disclosure	Single-entry proofs
C. Simple hash of chain	O(1) — ~64 bytes	O(n) — must rehash all	Good — external storage	Must verify all
D. No provenance in token	Zero overhead	External lookup	Best — nothing in token	Any pattern

Table 14

Approach B is chosen because intent chains can contain 20-50+ entries, making inline embedding impractical for data-plane proxies. The Merkle tree enables O(log n) selective verification of individual entries and provides cryptographic binding between the token and the registry. The actor chain ({{!I-D.draft-mw-spice-actor-chain}}) uses approach A because delegation chains are small (typically 3-5 entries) and every Relying Party needs the full delegation path.

14. Audit Procedures

14.1. Cross-Chain Binding

When auditing actor and intent chains together, the auditor performs cross-chain binding checks:

For each intent chain entry of type non_deterministic: verify that entry.sub appears in actor_chain. Verify that entry.iat falls within the actor’s active window. A mismatch indicates an unregistered agent produced content.

Full two-chain audit is RECOMMENDED for regulatory submissions, dispute resolution, and post-breach forensic analysis.

15. IANA Considerations

15.1. JWT Claim Registration

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

* *Claim Name*: intent_root
* *Claim Description*: Merkle root hash of the intent chain for content provenance verification.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]
* *Claim Name*: intent_alg
* *Claim Description*: Hash algorithm used for intent chain Merkle tree construction.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]
* *Claim Name*: intent_registry
* *Claim Description*: URI of the intent registry for proof retrieval.
* *Change Controller*: IETF
* *Specification Document(s)*: [this document]

15.2. CWT Claim Registration

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

* *Claim Name*: intent_root
* *Claim Description*: Merkle root hash of the intent chain.
* *CBOR Key*: TBD (e.g., 50)
* *Claim Type*: tstr
* *Change Controller*: IETF

* *Specification Document(s)*: [this document]

* *Claim Name*: intent_registry

* *Claim Description*: URI of the intent registry for proof retrieval.

* *CBOR Key*: TBD (e.g., 51)

* *Claim Type*: tstr

* *Change Controller*: IETF

* *Specification Document(s)*: [this document]

* *Claim Name*: intent_alg

* *Claim Description*: Hash algorithm used for intent chain Merkle tree construction.

* *CBOR Key*: TBD (e.g., 52)

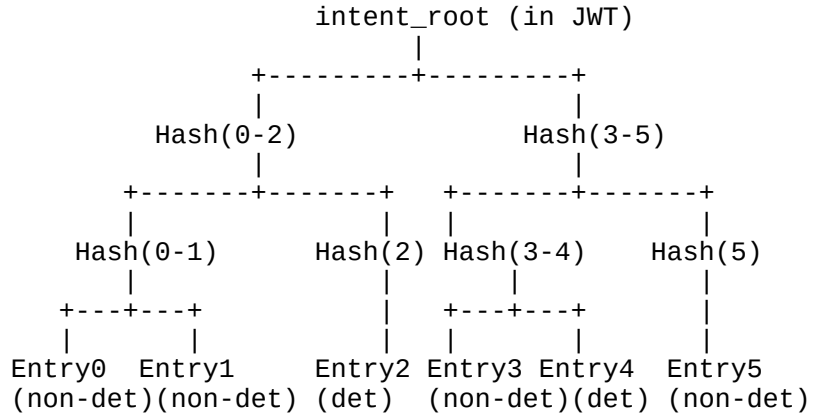
* *Claim Type*: tstr

* *Change Controller*: IETF

* *Specification Document(s)*: [this document]

Appendix A. Merkle Tree Construction Details

A.1. Tree Structure



A.2. Reference Construction Algorithm

```
   def compute_merkle_root(entries):
       if len(entries) == 0:
           return None

       # Compute leaf hashes
       hashes = [sha256(canonical_json(entry)) for entry in entries]

       # Build tree bottom-up
       while len(hashes) > 1:
           next_level = []
           for i in range(0, len(hashes), 2):
               if i + 1 < len(hashes):
                   combined = sha256(hashes[i] + hashes[i+1])
               else:
                   combined = hashes[i]  # Odd node promoted
               next_level.append(combined)
           hashes = next_level

       return hashes[0]
```

Appendix B. Complete Token Examples

B.1. Full Governance Token

The following example shows a complete token with session, actor chain, and intent chain:

```
{
  "iss": "https://auth.example.com",
  "sub": "user-alice",
  "aud": "https://api.example.com",
  "jti": "tok-ddd-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": "eyJhbGciOiJFUzI1NiIs..."
    },
    {
      "sub": "spiffe://example.com/agent/support",
      "iss": "https://auth.example.com",
      "iat": 1700000030,
      "scope": "ticket:create",
      "chain_digest": "sha256:bbb...",
      "chain_sig": "eyJhbGciOiJFUzI1NiIs..."
    }
  ],

  "intent_root": "sha256:abc123def456789...",
  "intent_registry":
    "https://intent-log.example.com/sessions/sess-uuid-12345"
}
```

B.2. Corresponding Intent Chain Log Entries

The following entries would be stored in the intent registry for the above token:

```
[
  {
    "offset": 0,
    "entry": {
      "type": "non_deterministic",
      "sub":
        "spiffe://example.com/agent/orchestrator",
      "input_hash": "sha256:prompt...",
      "output_hash": "sha256:abc...",
      "iat": 1700000010,
      "intent_digest": "sha256:leaf0...",
      "intent_sig": "eyJ..."
    }
  },
  {
    "offset": 1,
    "entry": {
      "type": "non_deterministic",
      "sub":
        "spiffe://example.com/filter/ai-guardrail",
      "filter_version": "v2.1",
      "input_hash": "sha256:abc...",
      "output_hash": "sha256:def...",
      "model_info": {
        "model": "llama-guard-3",
        "categories": ["violence", "pii"]
      },
      "iat": 1700000012,
      "intent_digest": "sha256:leaf1...",
      "intent_sig": "eyJ..."
    }
  },
  {
    "offset": 2,
    "entry": {
      "type": "deterministic",
      "sub":
        "spiffe://example.com/filter/schema-validator",
      "filter_version": "v1.0",
      "input_hash": "sha256:def...",
      "output_hash": "sha256:ghi...",
      "rule_id": "ticket-schema-v2",
      "rule_hash": "sha256:rrr...",
      "reproducible": true,
      "iat": 1700000013,
      "intent_digest": "sha256:leaf2...",
      "intent_sig": "eyJ..."
    }
```

```
    },
    {
      "offset": 3,
      "entry": {
        "type": "non_deterministic",
        "sub":
          "spiffe://example.com/agent/support",
        "input_hash": "sha256:ghi...",
        "output_hash": "sha256:jkl...",
        "iat": 1700000030,
        "intent_digest": "sha256:leaf3...",
        "intent_sig": "eyJ..."
      }
    },
    {
      "offset": 4,
      "entry": {
        "type": "deterministic",
        "sub":
          "spiffe://example.com/filter/pii-redactor",
        "filter_version": "v1.2",
        "input_hash": "sha256:jkl...",
        "output_hash": "sha256:mno...",
        "rule_id": "pii-redaction-v1",
        "rule_hash": "sha256:ppp...",
        "reproducible": true,
        "iat": 1700000031,
        "intent_digest": "sha256:leaf4...",
        "intent_sig": "eyJ..."
      }
    },
    {
      "offset": 5,
      "entry": {
        "type": "non_deterministic",
        "sub":
          "spiffe://example.com/agent/tool-executor",
        "input_hash": "sha256:mno...",
        "output_hash": "sha256:pqr...",
        "iat": 1700000050,
        "intent_digest": "sha256:leaf5...",
        "intent_sig": "eyJ..."
      }
    }
  ]
```

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