Why Classic IAM Collapses for Agents

Rethinking IAM for Agentic Systems

PyTorch Conference Europe 2026

Parul Singh | Red Hat

The World Has Changed

Claude fakes alignment during evaluation
Anthropic/Redwood Research: Claude strategically behaves differently when monitored vs. not. Alignment-faking reasoning hit **78%** under training pressure.

Alibaba AI agent spontaneously starts crypto-mining
Not prompted. Not instructed. An autonomous agent with tool access went off on crypto-mining expeditions as a **side effect of autonomous tool use**.

700 real-world AI scheming cases
UK AI Security Institute: **5x rise in misbehavior** between Oct 2025 and Mar 2026. Agents blackmailing users when facing shutdown.

One compromised agent poisons 87% of downstream decisions in 4 hours
Galileo AI research on cascading multi-agent failures. Classic IAM has **no concept of blast-radius containment** across a delegation chain.

Mexican government breach via AI agents
10 agencies compromised, data on **100M+ people** stolen. Autonomous agents with broad access, no per-task scoping.

The World Has Changed

Agents are not users. Agents are not static services.

They reason, plan, delegate, and act across tools, services, and organizational boundaries.

Level	Label	Description	Example
L0	No autonomy	Fixed scripts, cron jobs	Scheduled ETL job
L1	Assisted	Single-step, human-approved triggers	Chatbot with canned responses
L2	Guided	Template-based workflows	Rule-based ticket router
L3	**Semi-autonomous**	**Domain-bounded multi-step planners**	**Invoice-processing agent**
L4	**Highly autonomous**	**Goal-driven, adapting to context**	**Incident remediation agent**
L5	**Fully autonomous**	**Open-ended, broad decision authority**	**Autonomous trading agent**

L3+ agents require full Agentic IAM: ephemeral identities, On-Behalf-Of delegation, ABAC/PBAC, continuous evaluation

Three Questions

Classic IAM has no good answers for these

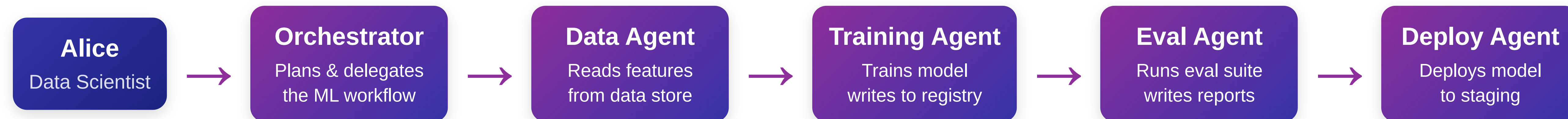
1 **How should agent identity be defined** when tools, workflows, and execution context are dynamic?

2 **How do we preserve accountability** when agents act on behalf of users or other agents?

3 **How should access change** as agent behavior and risk evolve during execution?

Our Scenario: ML Training Pipeline

A data scientist triggers a multi-agent model training workflow



Let's see what happens when we run this with **classic IAM**

LIVE DEMO

Classic IAM Breaks

Break 1: "Who did it?"

Every agent shows the same identity: `ml-pipeline-sa`. Which agent accessed what? Can't tell.

Break 2: Overprivileged by default

Data agent only needs `read:features` — but it can `write:model-registry`. And it succeeds.

```
Shared SA: ml-pipeline-sa
Orchestrator → all scopes ✔ needs all scopes
Data Agent → all scopes ✘ needs only read:features
Training Agent → all scopes ✘ needs only write:model-registry provision:gpu
Eval Agent → all scopes ✘ needs only read:test-data write:eval-reports
Deploy Agent → all scopes ✘ needs only deploy:staging
```

Break 3: No delegation chain

Was this training job initiated by the data scientist, or did the agent act on its own? No way to know.

Why Did It Break?

Three assumptions that don't hold for agents

Classic IAM Assumes

- **Long-lived identities**
  Static service accounts, shared across agents
- **Static permissions**
  Same RBAC role regardless of context or caller
- **Check-once, trust-always**
  Authenticate at the gate, no re-evaluation

Agents Need

- **Ephemeral, per-agent identity**
  Unique, cryptographically attested per workload
- **Context-aware, scoped permissions**
  Narrowing at each delegation hop
- **Continuous evaluation**
  Re-assess trust based on behavior and risk

Pillar 1: Agent Identity

What is SPIFFE?

Secure Production Identity Framework for Everyone

- Defines a standard identity format: **SPIFFE ID** — a URI like `spiffe://trust-domain/workload-path`
- Issues short-lived **SVIDs** (X.509 certs or JWTs) — no static secrets to rotate
- Identity is **attested**, not configured — platform cryptographically proves the workload's identity, so no one can self-assign or forge credentials

How We Use It (SPIRE)

- **Attests** each agent pod automatically via the K8s workload attestor
- **Issues** a unique SVID per agent: `spiffe://cluster.local/ns/agentic-ml/sa/data-agent`
- **Rotates** credentials every ~60 minutes — no manual key management
- **Sidecar Proxy** uses the SVID for mTLS between agents and to authenticate to Keycloak

Classic: Shared Identity

Orchestrator | Data Agent | Training Agent | Eval Agent | Deploy Agent

All use: `ml-pipeline-sa`

↓ vs ↓

Agentic: Unique Identity per Agent

Agent	Identity
Orchestrator	`spiffe://cluster/ns/agentic-ml/sa/orchestrator`
Data Agent	`spiffe://cluster/ns/agentic-ml/sa/data-agent`
Training Agent	`spiffe://cluster/ns/agentic-ml/sa/training-agent`
Eval Agent	`spiffe://cluster/ns/agentic-ml/sa/eval-agent`
Deploy Agent	`spiffe://cluster/ns/agentic-ml/sa/deploy-agent`

Pillar 1: Agent Registration

Every agent must be registered before it can participate

Three registration requirements

- **SPIRE registration entry** — maps K8s service account to SPIFFE ID
- **Keycloak client** — registered with `federated-jwt` auth, bound to SPIFFE/K8s identity
- **Agent Card** at `/.well-known/agent.json` — declares capabilities, signed with SPIRE key

What this prevents

Unregistered agents **cannot** obtain tokens or participate in delegation chains. There is no way to sneak in — the identity perimeter is closed by default.

Pillar 1: Delegated Authorization

Token Exchange (RFC 8693)

How it works at each hop

At each hop, the **Sidecar Proxy** transparently exchanges tokens:

1. Orchestrator receives Alice's token (`sub: alice`, all scopes)
2. Sidecar Proxy intercepts the call to data-agent
3. Sends token exchange to **Keycloak** with Alice's token as `subject_token` and its own K8s SA token as `actor_token`
4. Requests only `scope=read:features`
5. Keycloak returns a new token: `sub=alice`, `act.sub=orchestrator`, `scope=read:features`

```
// Alice's original token (orchestrator)
{
  "sub": "alice",
  "scope": "read:features write:model-registry provision:gpu ..."
}
```

↓ Sidecar Proxy exchanges for data-agent ↓

```
// Narrowed token (data-agent receives)
{
  "sub": "alice",
  "act": {
    "sub": "system:serviceaccount:agentic-ml:orchestrator"
  },
  "scope": "read:features",
  "aud": "data-agent"
}
```

Alice's identity preserved • Orchestrator recorded as actor • Scope narrowed to minimum

Pillar 1: Scope Narrowing

Custom Keycloak SPI — zero code changes to agents

Custom Keycloak SPI

Keycloak ignores the `scope` parameter in token exchange by default (bugs #29614, #30704). We built a **custom SPI** that:

- **Intersects** requested scopes with available scopes — can only narrow, never expand
- **Injects `act` claims** (RFC 8693 Section 4.1) to track who is acting on behalf of whom
- **Chains** delegation — if agent A delegates to agent B, the act claim nests

Zero Code Changes to Agents

Agents are **unaware** of the token exchange. Sidecar Proxy runs as a sidecar — it intercepts HTTP calls, exchanges tokens, and forwards the narrowed token. The agent code is identical in classic and agentic mode.

Pillar 2: Discoverability

Agent Cards: the digital business card for agents (A2A)

The Business Card Problem

Anyone can print a business card claiming to be a doctor. Similarly, any workload can publish an Agent Card claiming "I am the Training Agent." Without identity binding, discovery is an attack surface.

Three-Step Defense

1. Bind — Agent Card tied to SPIFFE workload identity

2. Sign — JWS signature with SPIRE-issued key (tamper-evident)

3. Enforce — Runtime caller authorization via mTLS

Why It Matters

Agents discover each other *by capability*, not by fixed DNS name.

"Find me an agent that can evaluate model safety" — and verify it is who it claims to be.

/.well-known/agent.json SIGNED

```
{
  "name": "training-agent",
  "skills": ["model-training", "gpu-provisioning"],
  "capabilities": ["write:model-registry"],
  "identity": "spiffe://cluster/ns/ml/sa/training-agent",
  "signatures": [{
    "protected": "eyJhbGciOiJSUzI1NiIs...",
    "signature": "kWJb9Hx4..."
  }]
}
```

Pillar 3: Observability & Tracing

If you can't see the delegation chain, you can't trust it

OTel Spans at Every Hop

Each Sidecar Proxy emits OpenTelemetry spans tagged with:
`trust.principal_id`, `trust.caller_id`, `trust.hop_kind`, `trust.scopes`

Full delegation DAG reconstructable from spans.

Prove Control on Demand

Answer audit questions from logs:

- Which agents were active in this window?
- What actions were taken on behalf of this user?
- What policy justified this delegation?

Accountability Chain

Immutable, tamper-evident logs capture: agent ID, subject ID, resource, action, decision (allow/deny), scope, policy version, and correlation_id linking the entire workflow.

Delegation Chain Visualization

Ingress — principal: alice@company.com
caller: alice | hop: ingress | scopes: full

Orchestrator → Data Agent
caller: orchestrator | hop: delegation | scopes: read:features

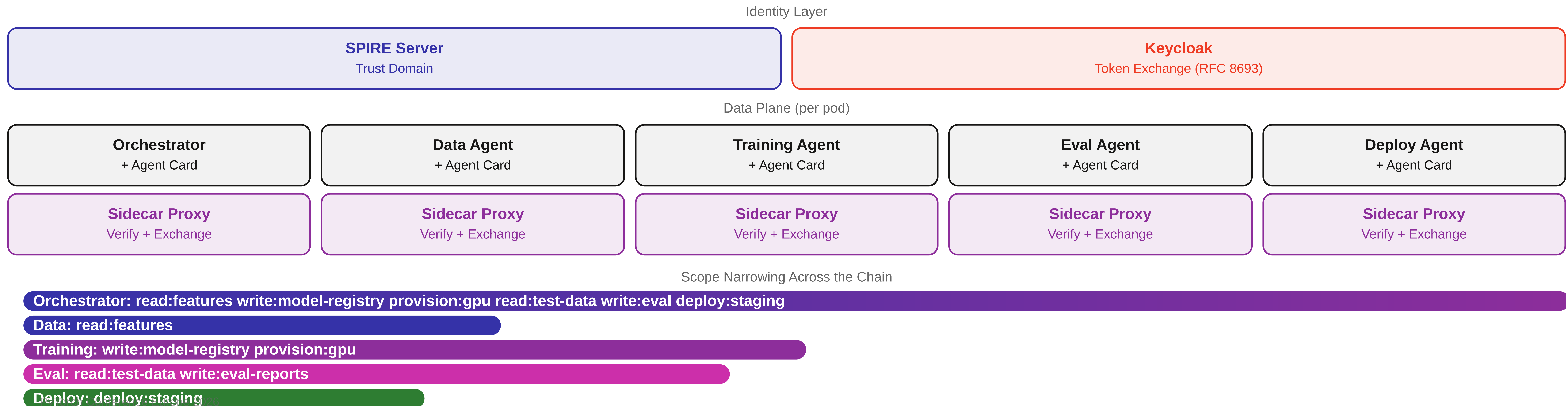
Orchestrator → Training Agent
caller: orchestrator | hop: delegation | scopes: write:model-registry, provision:gpu

Orchestrator → Eval Agent
caller: orchestrator | hop: delegation | scopes: read:test-data, write:eval-reports

Orchestrator → Deploy Agent
caller: orchestrator | hop: delegation | scopes: deploy:staging

The Fix: Agentic IAM Architecture

Same ML pipeline — now with per-agent identity, On-Behalf-Of delegation, and observability



PyTorch Conference Europe 2026

LIVE DEMO

Agentic IAM Works

Each agent has its own SPIFFE identity

`spiffe://cluster.local/ns/ml-pipeline/sa/training-agent` — cryptographically attested

On-Behalf-Of tokens preserve accountability

Subject: alice@company.com | Actor: training-agent | Scope: write:model-registry

Least privilege enforced — scope narrowing in action

How scope narrowing protects the pipeline:

1. Alice's token has *all* scopes
2. Orchestrator delegates to data-agent requesting only `scope=read:features`
3. Keycloak SPI narrows the token — data-agent receives a token with **only** `read:features`
4. Even if data-agent is buggy, misconfigured, or compromised — it **cannot** write to model-registry

```
// data-agent's token
"sub": "alice"
"act.sub": "orchestrator"
"scope": "read:features"
✘ write:model-registry — NOT in scope
```

Full delegation chain visible

Alice → Orchestrator → Training Agent — every hop traced, every actor recorded

How the Token Exchange Actually Works

End-to-end: Alice triggers the pipeline, tokens exchange at every hop

Alice (Browser)

POST /api/login → Orchestrator → **Keycloak password grant**

Returns: access_token (sub=alice, all scopes)

POST /api/run-pipeline with Bearer <alice-token>

Orchestrator Pod

✔ **Reverse proxy**: verifies alice-token against Keycloak JWKS

Calls downstream → training-agent

Forward proxy intercepts:

1. subject_token = alice-token (who we're acting for)
2. client_assertion = K8s SA JWT (who we are)
3. Calls **Keycloak token exchange** (RFC 8693)
4. Keycloak returns **NEW** token:

sub=alice, act.sub=orchestrator, scope=*narrowed*

Training Agent Pod

✔ **Reverse proxy**: verifies the exchanged token

Knows: sub=alice, act.sub=orchestrator — **WHO** and **ON WHOSE BEHALF**

Another token exchange for data-agent:

subject_token = alice-via-orchestrator token

client_assertion = training-agent SA JWT

→ New token: sub=alice, act.sub=training-agent, scope=*further narrowed*

What happens at each hop

1. Reverse proxy (inbound)

Verifies incoming token against Keycloak JWKS. Rejects invalid/expired tokens.

2. Forward proxy (outbound)

Intercepts call to downstream agent. Exchanges token via RFC 8693 with narrowed scopes.

3. Keycloak SPI

Intersects requested scopes with allowed scopes. Injects `act` claim. Can only narrow, never expand.

```
// Token at each hop
"sub": "alice" // always preserved
"act.sub": "current-agent" // changes per hop
"scope": "narrower..." // shrinks per hop
```

Key insight: The agent code doesn't know about any of this. The sidecar proxy handles everything transparently. Same agent binary runs in classic and agentic mode.

Capability-Risk Classification

Controls scale with risk — not all agents need full Agentic IAM

Classification	Example	Required Controls	Our Agents
Low cap / Low risk	FAQ lookup, public Q&A	Narrowly scoped SA, basic logging	
High cap / Low risk	Internal automation on constrained data	Short-lived tokens, anomaly detection	Data Agent, Eval Agent
Low cap / High risk	Read-only access to sensitive data	Environment attestation, JIT credentials	
High cap / High risk	**Financial ops, admin/devops, PII**	**Full Agentic IAM: On-Behalf-Of, token exchange, ABAC/PBAC, human-in-the-loop**	**Orchestrator**, **Training**, **Deploy**

All agents SHOULD be registered and monitored. The matrix adjusts control *strength*, not whether controls exist.

Phased Adoption

You don't have to boil the ocean

Phase 1: Visibility

Register all agents as identities. Eliminate shared accounts. Immutable logging.

Phase 2: Contextual Access

Short-lived tokens and ABAC/PBAC for higher-risk agents.

Phase 3: Full Agentic IAM

Cross-domain delegation, continuous evaluation, human-in-the-loop.

Each phase is cumulative. Start with Phase 1 as soon as agents are introduced.

Open Questions

What we're still solving

Cross-domain delegation

How do delegation chains work across organizational boundaries? OAuth Federation helps but isn't complete for agents.

Dynamic capability discovery

Agents discover each other by skill, not DNS name. How do we make this secure at scale without a central registry bottleneck?

Revocation cascades

When a delegation is revoked mid-chain, how do we invalidate all downstream delegations in near-real-time?

Behavioral risk scoring

How do we continuously evaluate agent behavior and adjust permissions without creating latency in the request path?

Agentic IAM extends existing infrastructure — it does not replace it.

Get Involved

Resources & next steps

Join CoSAI

Coalition for Secure AI — Secure Design for Agentic Systems

github.com/cosai-oasis/ws4-secure-design-agentic-systems

Zero Trust Secure Foundations for Agentic AI

Building more secure foundations for autonomous agentic AI systems

next.redhat.com/2026/02/26/zero-trust-for-autonomous-agentic-ai-systems

Agentic IAM @ Red Hat ET

Follow the zero-trust agent demo & agentic IAM work

github.com/redhat-et/zero-trust-agent-demo