SADAR

Semantic Agent Discovery and Attribution Registry

An Authorization Substrate for Agentic AI

Specification Overview

Draft v3.1 — May 2026

Published by Cognita AI Inc. through OpenSemantics.org

under the Community Specification License 1.0

Table of Contents

After opening this document in Word, right-click the table below and choose "Update Field" to populate the table of contents.

About This Document

This document presents the SADAR specification — the Semantic Agent Discovery and Attribution Registry — a proposed authorization substrate purpose-built for the agentic AI paradigm. It is organized to be read alongside the Cloud Security Alliance's "Agentic AI Identity and Access Management: A New Approach" (August 2025). The structure mirrors the CSA paper section-by-section, allowing direct comparison of how each framework addresses the same problem space.

This document is not intended to be a comparison of SADAR versus the CSA model. The organization is to facilitate the reader in that comparison. A companion document, SADAR and the CSA Agentic AI IAM Framework, serves as a comparison of the two architectures.

Where the CSA framework approaches the agentic problem as Identity and Access Management, SADAR approaches it as authorization at the moment of invocation. The shift in framing is consequential: it leads to different architectural choices around what travels with the workflow, where enforcement occurs, how organizations federate, and how observability is owned. This document develops the SADAR position in detail.

Published by Cognita AI Inc., founder and steward of SADAR, through OpenSemantics.org under the Community Specification License 1.0. SADAR™ is a common-law trademark of Cognita AI Inc.

Abstract

Agentic AI systems compose at runtime. A user expresses an intent or establishes a goal for an agent. An orchestrating LLM plans a path using agents, tools, and resources discovered probabilistically. Each step, including the first, may recursively invoke further components across organizational, cloud, and technical boundaries. No party knows in advance what the workflow will traverse — not the user, not the orchestrator, not the operators of the downstream services.

This composition pattern breaks the assumptions traditional Identity and Access Management relies on.

Because the usage pattern and users are pre-defined in traditional software, it is possible to statically define access control lists and roles. This relies on the usage being controlled by traditional programming constructs that dictate what is used, by whom, when, and why. In the traditional model, functions can be scoped to the identity of the executing code because of this structure. Since the code only executes in this known and controlled context, this works. With the dynamic planning and discovery of agents, tools, and resources, it does not.

In traditional systems, authorization can be attributed to an ID because the execution is constrained to pre-defined use cases. With agentic AI, the agent, tool, and resource identity is the wrong point of control.

Since agents, tools, and resources are identified and composed dynamically, their standalone identity lacks the context of traditional software. We no longer are assured who is using it, for what intent or

purpose, or why or for what in that purpose it is being executed. With just an ID, it is impossible to determine authorization — that requires context:

Authorization = originator's scope of authority + macro-process intent + step definition in process context + performing component + accrued state (operations completed and the state of objects acted upon)

The full combination of these elements defines the context from which authorization decisions can be made. Current and proposed models do not address this.

- Static scopes cannot be defined for actions and flows that have not yet been planned.
- Pre-declared rich authorization requests presuppose the chain is known.
- Session-based delegation tracks identity but loses the originating principal's scope of authority for the declared intent.

Identity, however rich, cannot determine whether a specific invocation, discovered and defined at runtime, should be permitted.

SADAR addresses this by separating identity from authorization and propagating both with the workflow.

- Component identity is established through publisher-signed manifests declaring purpose, optionally within a longer process.
- Runtime authentication uses operating IAM standards (OIDC, DPoP, mTLS) directly between the requester and the server, cryptographically bound to each at runtime.
- A SADAR Context Token (SCT) contains the originator's identity, declared intent (e.g., process, goal, task), and a cryptographically accruing chain of executed steps.

The SCT travels in a dedicated SADAR-Context HTTP header alongside the OIDC usage token, not as a token claim. Both are required on every invocation.

The framework specifies bilateral trust between requester and service, with the registry mediating discovery but not occupying the runtime trust path. Cross-organization federation is the first-class deployment model following the same authentication and authorization patterns.

Observability uses OpenTelemetry with the workflow context propagated via baggage and span data repatriated to the originator's collector.

Specification conformance, certification, and authorization are decoupled normative concepts, each independently governable. Certification and authorization are only required for public federation.

This paper presents the contributions in detail:

- It diagnoses why traditional IAM cannot address authorization under runtime composition, and identifies the architectural prerequisites authorization actually requires.

- It defines component identity through publisher-signed manifests with explicit process semantics: what the component performs, what it explicitly does not perform, and what it expects to have been completed before it runs.
- It specifies the SCT as a JWS-inside-JWE artifact that propagates originator authority through every hop, with five chain operations (Open, Continue, Hold, Authoritative Carry, Close) governing how the chain extends, suspends, and terminates.
- It presents the four-layer architecture: registry and discovery, authentication and authorization, enforcement, and observability — with bilateral trust, service-sovereign credential issuance, and federation as primary.
- It details operational use across authorization, audit, monitoring, and incident response, with the Telemetry Record providing per-invocation persistent audit and repatriation providing end-to-end observability owned by the originator.
- It analyzes deployment models, the registry topology, and governance considerations for the decoupled normative concepts.
- It addresses security using the MAESTRO threat model and identifies where SADAR prevents threat classes structurally rather than relying on detective controls.

Keywords: agentic AI, authorization, runtime composition, originator authority, signed manifests, federation, OpenTelemetry, Zero Trust.

Authorization Substrate, Not IAM Framework

Before the technical detail, one framing point shapes how the rest of the document should be read.

SADAR positions itself as an authorization substrate rather than a framework. The distinction matters. A substrate provides an underlying enablement foundation upon which frameworks and solutions can be built. A framework specifies an end-to-end solution including identity, authentication, authorization, and enforcement — typically bundled with assumptions about deployment topology, governance model, and ecosystem participation.

The SADAR substrate provides the prerequisites that authorization decisions require, without specifying how the policy engine evaluates them, what entitlements an organization defines, or how exceptions are handled. SADAR makes the authorization inputs available at every enforcement point. It does not dictate which policies an organization implements against those inputs.

This positioning serves cross-organizational deployment. Different organizations will reasonably make different policy choices: how strictly to enforce, what entitlements to define, what risk tolerance to accept, what exceptions to permit. SADAR ensures that all of them have the inputs they need to make their own decisions. The substrate does not impose a policy model; it supplies the verifiable context the policy model needs.

This positioning also enables IAM frameworks to layer above it. The CSA Agentic AI IAM framework, organizational policy engines, industry-specific compliance overlays, and bespoke enterprise authorization stacks can all consume SADAR's substrate. The substrate is what makes authorization decidable; the layers above decide what to authorize.

The reader should therefore expect this document to specify mechanisms, not policies; verifiable inputs, not policy engines; cryptographic substrates, not deployment products. Where the document references how a policy engine might use SADAR's inputs, the intent is illustrative, not normative.

1. Introduction

Understanding Agentic AI and Its Unique Authorization Challenges

What Makes Agentic AI Different

Agentic AI systems are autonomous software processes that observe, plan, reason, and execute multi-step tasks with limited or no human supervision. Unlike traditional applications, which traverse predetermined workflows bound to a strict domain and range of users and implementation, agentic systems exhibit properties that fundamentally alter the authorization problem:

- **Runtime composition.** The path through agents, tools, and resources is constructed during execution by an LLM-driven planner. The composition is probabilistic, not deterministic. Prior to execution time, the steps are unknown, the user is unknown, the purpose is unknown, the agents, tools, and resources to be used are unknown.
- **Recursive delegation.** Each component in the composition may itself act as an orchestrator, running its own observe-plan-discover-execute loop, presenting the same challenges and growing with each iteration and delegation.
- **Cross-boundary execution.** Components are not bound to a single organization, cloud, or technical stack. A workflow initiated in one organization may invoke services hosted by partners, vendors, or independent providers.
- **Probabilistic discovery.** Component selection involves semantic matching against natural-language capability descriptions. The same request may select different components on different invocations.
- **Probabilistic invocation.** Recent research shows that even with a small number of choices, agents fail to call, and call out-of-order, other agents and tools.
- **Probabilistic data mapping.** The LLM must match data from its memory, prompt, agent and tool descriptions, and what is discovered at runtime, ensuring data is used correctly across execution. This is enough of a challenge in a single invocation, but this consistency must persist across all agents, tools, and resources for the entire chain.

These properties combine to produce a system where, at every point, it is unknown beyond the next step what process flow is ahead, what it will be doing, or why. The originating user does not know. The orchestrating agent does not know. The operators of downstream services do not know — they are not even identified yet. The path is constructed, and continually adjusted, in flight.

Current implementations do not have the tooling to provide the context across the process chain necessary for deciding authorization.

The Scale and Complexity Challenge

Industry projections vary, but the trajectory is consistent: enterprises are moving from dozens of agents to thousands within the next few years.

The challenge is not the number of agent identities to manage. The challenge is that as the number of agents, tools, and resources and hops increases, the distance between the originator's intent and authority and the point of action increases. In a single ReACT agentic system with one or two tools, perhaps this risk is manageable. But in a system with n agents, iterating m times each, and x tools and resources used, the mutation of that intent — without propagated context for grounding — is unknowable, as are the potential impacts.

Authorization decisions scale combinatorially with composition complexity. Context propagation provides grounding and baseline for authorization decisions.

The Originator Authority Problem

Traditional authorization rests on a stable triad:

- a known user with defined authority,
- a known intent, and
- an application that executes that intent within that authority.

The IAM has the user's roles. The operating system, SSO, and other coarse-grained controls determine whether the user can call the application, while the application uses role memberships to apply fine-grained controls within the application.

Agentic systems break this immediately. The originator authenticates to a planning agent. The agent is likely specialized to a task with its instructions in its system prompt. Coarse-grained controls determine whether the user is authorized to call the agent. Beyond that, the model is broken.

The system prompt can mimic some fine-grained controls with hints and instructions, but those are probabilistically applied. Models are getting better at following instructions, but this is still not deterministic, nor are those decisions definitively logged. Not only does this rely on the LLM recognizing the semantic instructions, it requires the LLM to probabilistically match the data to the criteria.

Each step in an LLM-driven workflow — tool selection, data selection, invocation, interpretation — has its own probability of success. The probability that the entire chain succeeds is the product of the individual step probabilities. As steps multiply, total success probability decays geometrically.

Relying on probabilistic matching to find, invoke, map data in and out, and apply constraints from a system prompt is fragile, with any errors propagating exponentially — and silently.

The planner's own reasoning drives every decision: which sub-agents to discover, which tools to invoke, which data to access, which actions to take. None of those decisions is checked against the originator's authority by any external control. Even when the planner runs under the originator's identity — inheriting the originator's permissions through OAuth on-behalf-of, session-context inheritance, or equivalent mechanisms — the identity bounds what could be called within the universe of permitted resources. It does not constrain what the planner chooses to call within that universe.

The originator's general authority is a permission ceiling, not an intent filter. A user authorized to read customer records, modify production data, and send email holds those capabilities; an agent operating under that user's identity inherits them.

The architecture has no mechanism for asking, at each subsequent action, whether this specific action falls within the originator's intent and their authority bound to that intent for this session. The originator's authority is consulted exactly once — at the moment the originator invokes the first agent.

From that moment, authorization is assumed.

This is the structural setup for confused deputy at scale, and it manifests at iteration one. By the fifth hop the visible distance between where authority was checked and where the action is occurring is greater, but the authorization gap has existed since the planner's first observe-plan-discover-execute step.

If the authority context is not propagated and made available at every invocation, nothing enforces the originator's authority, giving rise to privilege escalation and undesirable results. Any view into what decisions were made and how requires reconstruction from logs and application of the scope of authority to each step to determine if authority was exceeded. That reconstruction is difficult in single-organization deployments and impossible across organizational boundaries.

Why a New Approach Is Needed

Extensions to OAuth 2 — Rich Authorization Requests, structured scopes, agent-specific actors, signed delegation chains — are real engineering work. Each addresses a real problem. None addresses this authorization gap.

OAuth requires that the calling party enumerate, at request time, the authorizations the workflow will require. Agentic workflows cannot satisfy that presupposition. You cannot enumerate authorizations for invocations you do not know will occur.

A framework that treats the problem as Identity and Access Management will produce richer identity, but that cannot answer the authorization question that matters: given a runtime invocation, on behalf of an originator whose intent and scope are themselves dynamic, should this action be permitted?

SADAR addresses the question directly. The architecture propagates the originator's intent-bound scope of authority through every invocation as a single accruing credential. The SCT is available to an

enforcement point. That gate evaluates the request against the originator's authority, the declared intent, the invoked component's signed manifest, the fit of the invocation to the overall purpose, and the sequence of steps already executed. The authorization decision is made where the invocation occurs, with the information required to make it correctly, every time.

Note: The architecture and mechanism of the enforcement are outside the scope of SADAR. SADAR's responsibility ends with making that context available.

Contributions of This Paper

This paper makes the following contributions:

- It analyzes the gaps of existing IAM and authorization protocols for runtime composition, identifying the architectural prerequisites authorization actually requires.
- It defines component identity through publisher-signed manifests with explicit process semantics, providing globally unique component identifiers and verifiable publisher attribution.
- It specifies the SADAR Context Token (SCT) and the chain operations (Open, Continue, Hold, Authoritative Carry, Close) that govern authority propagation through composed workflows.
- It presents the four-layer SADAR architecture and details how bilateral trust, service-sovereign credential issuance, and federation-as-primary support cross-organization operation.
- It demonstrates how authorization, auditing, monitoring, and incident response operate in practice, with worked examples in both single-organization and cross-organization scenarios.
- It analyzes deployment models, the registry topology, and the governance implications of decoupling conformance, certification, and authorization as independent normative concepts.
- It addresses security threats using the MAESTRO model and identifies threat classes that SADAR prevents structurally rather than detects after the fact.

The remainder of this paper is structured as follows. Section 2 establishes the imperative for a new authorization paradigm by examining where traditional approaches fail. Section 3 defines component identity. Section 4 presents the SADAR architecture. Section 5 details operational use. Section 6 analyzes deployment models and governance. Section 7 addresses security considerations. Section 8 names the innovative contributions. Section 9 discusses open problems and future work.

2. The Imperative for a New Authorization Paradigm

Agentic AI breaks the assumptions traditional Identity and Access Management was built on. This section examines those assumptions, identifies where extensions to existing protocols fall short, and establishes the architectural prerequisites authorization requires in a system where composition is dynamic and recursive.

2.1 Revisiting Traditional Authorization

The dominant authorization frameworks — OAuth 2.x, OIDC, SAML, and their derivatives — were designed for a world of predictable workflows. A user authenticates to an application; the application requests, in advance, the scopes it will need; the user grants those scopes; the application acts within them. Authorization is a request-time activity. The set of permitted actions is known when the access token is issued.

This model has served the web well for nearly two decades. Its limitations for agentic systems are not minor extensions to be made. They are structural.

Coarse-grained, Static Scopes

OAuth scopes were designed to be human-comprehensible labels: "read email," "modify calendar." They cannot express the kind of fine-grained, resource-specific authorizations that agentic workflows need: "read the customer email thread relevant to ticket 47832," "modify only the next thirty days of the marketing team's calendar entries that involve external attendees."

Rich Authorization Requests address some of this by allowing structured authorization details at request time, but the fundamental problem persists: the calling party must declare at request time what the workflow will do. That is not possible with autonomous agentic flows.

The planner does not know at request time whether it will need to read three emails or three hundred, whether it will invoke the calendar API at all, or whether it will discover a more efficient path through a different combination of components. Each observe-plan-discover-execute iteration may add or remove components and actions. Declarations cannot be made at invocation because the execution plan is derived afterwards. Even with prompt instructions, you cannot be assured of a flow (if you did, you have also broken the value proposition of agents).

Lack of Agent Identity Recognition

Current OAuth implementations make no architectural distinction between human users and AI agents. The token presented to a service identifies the application that received the grant. It does not identify the agent that is acting at the current moment. When an agent can discover hundreds of agents, all sharing the same client credentials, the service has no way to attribute actions to specific agent instances or apply per-agent policy.

Proposals to add agent identifiers to OAuth tokens are a necessary foundation. Every agent, tool, and resource needs a distinct identity. However, an IAM-issued identity only says what the component is called within the scope of that IAM. It:

- Is not globally unique and recognizable across IAM boundaries.
- Is issued by the IAM, not by the publisher or owner of the component, lacking any provenance.
- Does not present a cryptographically signed description of what the component does or any usage concerns (cost, operational, compliance, etc.).
- Does not resolve to each party's authentication endpoint for validation and authorization.

In addition to these gaps, it is simply the wrong architectural point for authorizing an action.

The component — agent, tool, or resource — represents a "doer" either executing a process step, facilitating a lower-level function, or returning information. A file-write tool must be able to write to files or it serves no purpose. Asking it to provide authorization context is meaningless. It is the target of the authorization, not the authorizer. Determining if a file write should occur requires process context and originating scope of authority.

Inadequate Delegation

OAuth's delegation model assumes a hierarchical principal-to-principal relationship: user delegates to application, application acts. It does not model multi-hop delegation chains where each of the hops is notionally acting for the original user.

Token exchange flows partially address this, but they re-establish the principal identity at each hop. The user's original scope of authority is not cryptographically present in Agent C's credentials — only Agent B's interpretation of it is.

Because of this, agents and tools may operate beyond the privileges of the originating user. They certainly operate without the context of what the user intends.

This is the structural setup for delegation drift and the confused deputy problem. Each downstream service evaluates the immediate caller's claimed authority and trusts upstream parties to have correctly scoped what was passed forward. As the chain lengthens, the gap between what the originator actually authorized and what the downstream service can verify widens.

Static Trust and Why Rich Authorization Requests Cannot Close the Gap

OAuth's authorization model is grant-time, not invocation-time. Once an entity is authenticated and an authorization is granted, that authorization persists for the token's lifetime regardless of how many actions are taken under it. The grant flows with the token even when the component holding it is now being used for a completely different purpose.

Rich Authorization Requests (RFC 9396) do not change this. They refine what a token permits but do not change when the permission is evaluated. A richly-scoped token can carry detailed authorizations that apply to a process flow the agent is no longer executing.

When the token expires, the component holding it has no basis for deciding whether to refresh or reauthenticate, because OAuth provides it no signal about the calling flow's intent. If it refreshes, it perpetuates the original grant. If it reauthenticates using its own credentials, the resulting token is severed from any originator or prior step. The component is not failing — it is executing exactly the credential-management logic it was built for. The framework simply has no mechanism for process context to enter that decision.

This model works extremely well for users accessing web sites and traditional applications. It does not work in dynamic agent flows.

The grant follows the token rather than the work.

The compounding defects below make this concrete:

- Authorization presupposes the authorizer knows what the executor will do. In agentic flows the plan is emergent — constructed at runtime from the originator's intent and the capabilities discovered in the environment. The authorizer at grant time can enumerate categories of permitted action but cannot determine whether a specific runtime instantiation falls within the originator's actual intent.
- A token issued in the context of one process flow remains valid and unchanged when the same agent is invoked into a different flow. Refreshing the token preserves — or at most narrows — the original grant; it does not re-evaluate whether that grant is appropriate for the flow the token is now servicing. The authorization server has no signal that the operational context has changed.
- The component managing the token has no awareness of process context. It performs actions. When a token expires, its behavior — refresh, or request a new token using its own credentials — is determined by static configuration, not by runtime intent. Neither path captures process context: refresh perpetuates the prior grant, client credentials produce a machine-context token unconnected to any originator or flow.

Tokens satisfy every syntactic validation. Audit logs record grants and invocations with full fidelity. But the authorizations carried by those tokens reflect choices made about a process definition before the LLM ever reasoned over it. There is no mechanism for ensuring RAR authorization is still relevant in token renewal, and the authorization context becomes severed from any prior authorizations if newly minted.

This is an architectural mismatch when used for emergent agent and component execution, and becomes actively dangerous when interpreted for invocation authorization.

Because RAR is not an invocation-time authorization for agentic systems, it can provide latent or incorrect — and therefore exploitable — authorizations.

What is needed is re-evaluation at every invocation — whether this specific action, as part of this specific process flow, on behalf of this specific originator, against the current state of that originator's intent, is appropriate — not whether the session that issued the token remains within its validity window. What such a model requires — invocation-time authorization with propagated originator authority, process context, and chain of custody — is the architecture the remaining sections specify.

Identity Confined to the Issuing Environment

All current authorization approaches bind component identity to the environment in which the component is deployed. An agent deployed in AWS has an IAM role. The same agent deployed in Azure has a different identity. Federation can bridge environments, but the component's underlying identity remains environment-bound; there is no global identifier that survives across boundaries. There is also no attribution to the publisher of the component code, no verifiable definition of capabilities tied to the component's identity, and no tie to the data the component consumes and produces.

Credentials are pre-shared (API keys, certificates) or minted by the requester's IAM solution. Requester and server cannot directly verify each other, and they cannot manage lifecycle aspects — revocation,

TTL, rotation — bilaterally. Captured credentials enable impersonation. Replay protection requires additional protocols layered on top.

SPIFFE represents the most mature attempt to address these gaps. SPIFFE IDs provide a cross-environment identifier in the form of a URI under a trust domain; SVIDs provide sender-bound credentials suitable for bilateral mTLS verification; SPIRE manages lifecycle including rotation and revocation; federated trust domains allow workload identities to be verified across organizational boundaries. Where SPIFFE is deployed, several of the deficiencies above are materially reduced.

What SPIFFE does not address — and does not claim to — is publisher attribution, capability declaration, or data binding. An SVID identifies the workload as deployed. It says nothing about who wrote the code that workload runs, what that code is supposed to do, or what data it is supposed to operate on. Two workloads with valid SVIDs in federated trust domains can authenticate to each other with full cryptographic confidence and still have no basis for deciding whether the invocation between them is appropriate.

Even if every gap in this section were closed — global identifiers, publisher attribution, sender-bound credentials, signed capability declarations — the result would be a better identity layer, not an authorization layer. Identity tells the executor who is calling. It does not tell the executor whether this call is appropriate, which is a separate question with separate inputs.

The gaps in process context identified in the previous section remain. Stronger identity does not supply the originator's current intent, the process definition this invocation belongs to, the transaction instance under that definition, or the chain of custody accrued so far. These are authorization inputs, and no identity standard — however well engineered — carries them. The problem to solve is the authorization of dynamically defined flows, and that problem is unchanged by improvements at the identity layer.

The risk of an agentic flow is not in what you intended it to do but in what it can discover and could do.

Real World Impacts

In April 2026 the founder of PocketOS, an automotive SaaS platform, was using Cursor for a routine task. The agent found what it viewed to be a credential mismatch error and used a long-lived API token to delete the production database — all within 9 seconds.

You can, rightfully, argue that the token was over-privileged. You are right, but that is not the point. The agent deviated from its intended function, ignored previous instructions, created a plan, found the API key, and executed.

A similar example caught Summer Yue, Director of Alignment at Meta Superintelligence Labs. She had been testing OpenClaw. Satisfied with the results in a sandbox, she gave OpenClaw access to her real email — again with strict instructions to protect her email, requiring her confirmation before action. OpenClaw deleted her inbox.

Mind you, this was Meta's alignment director — and she still got hit.

This is precisely the scenario we have been covering:

- Agent activities are not bound, and held, to an intent that survives throughout the chain.
- The user's intent-bound scope of authority does not survive across the chain — in this case the user had the general rights to make the change, but that change **was not a legitimate part of the intent's process.**
- The API token was long-lived. It was not run-time bound to the using agent. There was no authorization event.
- No authentication event between the agent and API. The agent found the API key in a config file (yes, another opportunity for better hygiene). All the API knew was "someone gave me a valid key." There was no opportunity for the API to say "that is a valid key but it is not yours."
- The probabilistic matching of the semantic similarity of the actions to the prompt-written guardrails was ignored. Whether because they had aged out of context or the LLM simply did not see the similarity between the action and the guardrail language as strong enough to trigger.

Every point we have discussed is illustrated in just these two examples. Unfortunately, there are many of these examples.

The last bullet deserves emphasis. The probabilistic use of guardrails was not properly enforced. Period.

As good as the models are becoming, using narrative to define structural requirements for LLM use is simply not reliable or transparent. This includes:

- Steps
- Data meaning and structure
- Checks and guardrails
- Must and must-not constraints

Regardless of why the "hints" were not followed, the impact is the same: unreliable and undesired behavior. SADAR uses defined scopes grounded in existing industry terminology for processes, steps, and data meaning and structure. They do not rely on the LLM probabilistically applying them correctly. They are deterministic for an enforcement layer, outside of SADAR's scope, to apply. This will be discussed in detail later in the document.

2.2 Why Agentic AI Demands a New Authorization Paradigm

Beyond the protocol mismatches, the nature of agentic systems imposes additional architectural requirements that no extension to traditional protocols can satisfy. This subsection enumerates them as the architectural prerequisites SADAR is designed to provide.

The Inputs to an Authorization Decision

In an agentic workflow, determining whether a specific invocation should be allowed requires evaluating, together, at every invocation (allowing for risk-based, caching, and similar policy decisions):

- Originator's scope of authority for the declared intent — itself intent-bound. The same user has different authority in different process contexts.
- Intent — what process is executing, for what purpose. The intent declared at the start of the workflow remains the authority context throughout.
- Component being invoked — agent, tool, or resource. Identity and verified capabilities, established through the publisher-signed manifest.
- Fit-to-purpose — whether this component's capabilities, expected preconditions, and declared exclusions are consistent with the workflow's overall purpose.
- Sequence with preconditions met — whether this action is in the correct order, and whether the component's expects_completed declarations are satisfied by the prior steps.

None of these can be evaluated in isolation. They are not independent gates; they are inputs to a single decision that must be considered at every invocation. A framework missing, or under-representing, any of these cannot make the authorization decision correctly.

Three Properties of These Inputs

First — originator authority is intent-scoped. The user's authority depends on the process being executed. An AP clerk authorizing a wire transfer in the context of vendor payment is not the same authority as in the context of a payroll transfer. The originator's authority must be evaluated against the intent declared at the start of the workflow.

Second — runtime context changes; the authority does not. The originator's authority for an intent does not change when an item goes on backorder, a shipping date slips, or a price moves. What changes is whether the current invocation still fits the original intent. Every observe-plan-discover-execute iteration is a new invocation with potentially new context to weigh against the same authority.

Third — this is a policy decision, not an identity decision. Evaluating the inputs together belongs in a policy engine that can reason about entitlements, runtime context, and process state. SADAR provides the mechanism to make these inputs available to that engine; SADAR is not itself the policy engine.

Architectural Prerequisites

For authorization to operate correctly in agentic systems, the following properties must be present in the architecture. None of them are properties of identity; they are properties of the layer that makes the authorization inputs verifiably available at the enforcement point. These prerequisites are also the architectural principles on which the SADAR architecture in Section 4 is built.

Globally unique component identifiers. Component identifiers must be unique across environments, technologies, and time. The component identifier is composed of the publisher's identifier, the component's identifier within the publisher's namespace, and the component's version. Once published, the identifier is immutable.

Originating authority that travels. The authority of the workflow invocation must propagate with the request across every delegation hop, including hops that cross organizational, cloud, and technical

boundaries. The authority must be cryptographically verifiable at every enforcement point, not interpretively re-issued at each hop. This authorization chain must cryptographically accrue, available end-to-end. The result is a single accruing credential — the SCT — that eliminates the structural setup for confused deputy and delegation drift.

Bilateral trust topology. The requester and the service must authenticate each other directly at invocation. The registry mediates discovery and supplies cryptographic material — it does not occupy the runtime trust path. The service owns its trust relationship with the requester and can revoke unilaterally without coordinating through a third party. This is the most consequential architectural choice in SADAR; multiple downstream properties follow from it. Bilateral matching also applies between requester and service: mandatory capabilities and non-functional requirements must be in agreement between the two parties, giving the server a right of refusal — declining callers whose operational requirements it cannot meet, asserting cost structures that must match the requester's, requiring asserted BAA and HIPAA compliance for HIPAA-bound tasks, and so on.

Manifest-based discovery. Discovery returns each candidate component's signed manifest, not just an endpoint reference. The manifest is the discovery payload. A consumer-side helper, searchAndInvoke, calls a deployment-provided selector that evaluates the manifests, verifies the publisher's signatures, and reads the components' declared process semantics (performs, does_not_perform, expects_completed) along with their authentication endpoints. The selector makes the final decision, which searchAndInvoke then invokes. This is fundamentally different from name-based discovery, and from LLM probabilistic selection: candidate components deterministically match the asserted requirements, with no ambiguity of functionality, non-functional requirements, or data requirements.

Direct authentication with time-of-use credentials. Credentials must be cryptographically bound to the parties presenting them, with the other party able to verify the binding. Credentials must be issued at the time of use, with revocation and expiry semantics enforced by the issuer. Captured credentials must not enable impersonation. Authentication uses existing IAM standards: mTLS, OIDC Client Credentials with RFC 7523 Private Key JWT for service-to-service flows, DPoP for participant binding where applicable. The usage key is delivered as a JWE claim encrypted to the requester's public key, cryptographically binding the key to the requester's identity. Tokens are only usable within the transaction instance for which they were minted.

Participant-managed session lifecycle. Both parties control session validity through clear semantic handshakes for authentication, refresh, expiry, and revocation. Deauthorization at the registry and session revocation at the service are independent acts that can be performed by the appropriate authority without coordination. Lifecycle handshake semantics must be clear.

Publisher-signed manifests with process context. All entries (requester, server, tool, resource, process, publishers, registries, and so on) must be attributed to an owning entity and cryptographically tied to that entity in a standardized manifest. Participants must be able to verify the manifest signature and its integrity. Manifests declare capabilities using process semantics, data meaning and structure, non-functional concerns, verification keys, and authentication endpoints. Capabilities define what the component performs, what it expects to have been completed before it runs, and what it explicitly does

not perform. Invocations must carry a unique transaction identifier and a nonce for replay protection and attribution. The publisher-signed manifest also anchors trust to a real-world accountable party.

Federation as first-class deployment. Cross-organization operation is the case to solve, not the case to accommodate. The architecture is designed around federation from the start: bilateral trust between registries, service-sovereign credential issuance, decoupled lifecycles, repatriated observability. Single-organization deployments are a special case of federation — the same substrate, with the federation set reduced to one party. Registries participating in a public federation must be authorized by the SADAR specification body, finding them credible against operational, safety, and reliability requirements. Private federations are also allowed without authorization and certification.

Observability with repatriation. Observability data is produced by every component as it processes a request: OpenTelemetry spans tagged with the workflow context, structured baggage fields, and per-invocation Telemetry Records. The workflow context propagates through OTel baggage. When span close conditions are met, the Repatriation protocol forwards relevant spans to the originator's collector under bilateral applicability rules. The originator owns the assembled trace as part of normal operation, not as a post-hoc forensic exercise. Privacy controls apply at the producer side; each producer encrypts or obfuscates sensitive fields before forwarding.

These prerequisites supply the authorization inputs. Originator authority and declared intent ride in the propagated workflow context. Component identity and fit-to-purpose derive from signed manifests. Sequence and preconditions come from manifest declarations validated against the propagated context. Trust comes from bilateral authentication and time-of-use credentials.

3. Defining Component Identity

To address authorization in multi-agent systems, the substrate must first answer a question that traditional IAM answers poorly: what the identity actually represents. In SADAR, the answer applies uniformly to all callable entities — agents, tools, resources, process definitions, owning entities, and registries themselves. All are components. All are defined through publisher-signed manifests. This section details the identity model and how it differs from agent-centric identity models in the broader literature.

3.1 Beyond Static Identifiers

Knowing just an identifier for a component does not tell you anything about where the component came from, what it does, what compliance frameworks it complies with, what it needs, where and when it should be called, or its operational concerns. You simply have an ID.

A SADAR component identity is a namespaced, globally unique ID cryptographically anchored to a set of declarations via the manifest. The manifest is published by a verifiable real-world party describing what the component is, what it does, what it does not do, what it expects, and how to authenticate to it. To be globally usable, the manifest is grounded in existing industry standards for categorization, processes,

transactions and steps, data, and non-functional requirements. The manifest supports extension via Internationalized Resource Identifiers, allowing for inclusion of definitions such as APIs of dominant vendors' systems as well as localized extensions.

The identifier is the combination of the entity (publisher's) ID + component ID + version. This identifier is represented in the manifest, signed by the publisher, verifiable by any potential consumer. The identity is the identifier plus the manifest.

The identity of a person is not their SSN or their name; it is who they are. The name and number are how we reference them.

The manifest is the public, machine-readable contract the publisher offers to the rest of the ecosystem. The signature ties that contract to the publisher's verified organizational identity — the same publisher key trust that other SADAR participants rely on for that publisher's other components.

Component Types

SADAR applies the manifest model uniformly to all callable entity types. The primary types are:

- **Entity —** the publisher of a component. Entities can be organized into a hierarchy representing whatever the entity owner wishes (legal entities, geographic, organizational, etc.). Every component must link to one, and only one, entity except for the parent entity, which is a root node.
- **Agents —** autonomous software entities that plan, reason, and execute multi-step tasks. Agents may orchestrate other agents, tools, and resources.
- **Tools —** callable functions exposing well-defined operations. Tools typically execute discrete actions and return results; they do not maintain long-running state across invocations.
- **Resources —** data stores, knowledge bases, or other persistent state accessible through defined operations. Resources are governed by the same manifest model as agents and tools.
- **Process definitions —** machine-readable declarations of intended flows, used by planners as reusable templates and by enforcement frameworks for sequence validation. Process definitions describe steps within the process. Agent and tool manifests declare, using the same semantics, what tasks they perform, which can then be related to an overall process.
- **Registries —** the discovery infrastructure itself. Registries have manifests describing their authoritative scope, their federation relationships, and their conformance posture.

Each type uses the same manifest format, with type-specific fields where relevant. The uniform model simplifies the substrate — the same discovery, signing, and verification logic applies to every component.

Why Publisher-Signed, Not Self-Issued

The choice of publisher signing over self-issued identity is intentional. A self-issued identity — in which the component itself, or its controller, creates and signs its own credentials — places trust in an opaque

party. The relying service learns that the component is who it says it is, but knows nothing about the publisher's accountability, organizational identity, or commitments.

Publisher-signed identity places trust in a real-world accountable party. The publisher is a known entity: a company, an open-source foundation, an individual developer with a verifiable identity. The publisher signs its components with keys it controls and rotates under its own key management. Disputes, vulnerabilities, security advisories, and compliance attestations all attach to the publisher — not to an abstract self-issued identifier.

This model aligns with how software supply chains already work. Code is signed by its publisher; container images are signed by the organization that built them; SBOMs are signed by their authors. SADAR extends the same model to runtime component identity. The publisher who signed the code is the same publisher who signs the manifest.

3.2 Essential Components of a Component Identity

A SADAR component identity, expressed in the manifest, contains the following elements.

Cryptographic Anchor

- **Publisher identifier —** a globally unique URN identifying the real-world party publishing the component. Publisher identifiers anchor the trust chain; they are verifiable through public key infrastructure or equivalent verification mechanisms. The registry is the Source of Record for publisher identifiers within SADAR — the authoritative place to resolve a publisher's SADAR identity. The underlying System of Record (e.g., D&B, S&P, stock symbol) remains the authoritative source for the publisher's real-world identifier; the registry binds the SADAR identity to one or more such external identifiers.
- **Component identifier —** a URN within the publisher's namespace identifying the specific component. The combination of publisher identifier, component identifier, and version forms the globally unique identifier.
- **Version —** a semantic version following standard conventions. Versions are immutable once published; updates require a new version with its own signed manifest.
- **Publisher signature —** a JWS signature over the manifest, with ES256 as the default algorithm. Other algorithms are permitted under the cryptographic baseline and defined within the manifest for consumption.
- **JWKS endpoint reference —** the publisher's key set, exposing at minimum one signing key and one encryption key. JWKS endpoints are required at every entity, agent, tool, and registry. Key rotation is handled by JWKS endpoint update without manifest republication.

Component Identification Metadata

- **Registry Entry Type —** entity, agent, tool, resource, process definition, or registry.

- **Publisher attestation —** the publisher's organizational identity, including any third-party verification (extended validation certificates, audited identity claims, regulatory registration).
- **Timestamps —** creation date, last update, and expected lifecycle bounds.
- **Software and model attestations —** cryptographic hashes of the component's code, model parameters, and material dependencies. SHA-3 family hashes are recommended.
- **Dependencies —** a normative reference to AIBOM (or equivalent supply-chain attestation) listing critical dependencies. Optional metadata but strongly recommended.
- **Lifecycle status —** active, deprecated, suspended, or revoked. Status changes are signed events propagated through the registry layer.

Capability, Scope, and Process Semantics

This is the most consequential section of the manifest for authorization. The SADAR manifest declares process semantics: what the component performs, what it expects to have been completed before it runs, and what it explicitly does not perform.

- **performs —** a list of IRIs identifying the operations the component performs. These IRIs are drawn from a shared taxonomy of agent capabilities, allowing semantic matching across organizations and ecosystems.
- **does_not_perform —** a list of IRIs identifying operations the component explicitly does not perform. This is not an absence of capability declaration; it is a positive declaration of non-action. The expects_completed list cannot contradict the does_not_perform list; manifests with such contradictions are invalid.
- **expects_completed —** a list of IRIs identifying operations that must have been completed before the component runs. Enforcement frameworks use this to validate sequence and prerequisites.
- **data interfaces —** declarations of data consumed and produced, with semantic annotations. A future schema extension will support a "one-of" pattern, allowing manifests to declare equivalent acceptable forms across standards under a component-chosen local name.
- **non-functional concerns —** SLAs, costs, compliance attestations, and other operational characteristics relevant to authorization decisions.

The manifest content specified here may be carried in any cryptographically verifiable envelope — W3C Verifiable Credentials, JWS over JSON, or other equivalent mechanism — provided publisher signature, key resolution, and revocation status are addressable. The envelope is an implementation choice; the schema, the process semantics, and the verification protocol defined in this specification are normative. SADAR manifests declare what the component is and does, with positive declarations of non-action and prerequisite completion, rather than asserting what the component may be permitted to do.

Authentication and Authorization Endpoints

- **Service endpoint —** the URL or set of URLs at which the component receives invocations.

- **Authentication endpoint —** the OAuth/OIDC endpoint at which a requester obtains a usage credential for the component. Each component's serving organization controls this endpoint; the registry is not the issuer.
- **Supported authentication methods —** mTLS, OIDC Client Credentials with Private Key JWT (RFC 7523), and other methods compatible with the cryptographic baseline.
- **Scope namespace —** the urn:sadar:scope:v1 namespace, plus any component-specific scopes the serving organization defines for fine-grained authorization.

Verifiable Attestations

Beyond the manifest itself, components may carry verifiable attestations about their operational properties, supply chain, or compliance posture. Attestations are signed by accredited third parties — not the publisher — and reference the component identifier. Examples include:

- **Conformance attestations —** attesting that the component implements specific SADAR conformance levels.
- **Certification attestations —** attesting compliance with industry-specific standards (HIPAA, SOC 2, PCI, EU AI Act risk categories).
- **Supply chain attestations —** signed by build pipelines, attesting to provenance, dependencies, and integrity.
- **Audit attestations —** from third-party auditors, attesting to operational properties.

Conformance, certification, and authorization are decoupled normative concepts in SADAR. A component can be conformant without being certified; it can be certified without being authorized for a specific use; deauthorization is independent of decertification. Section 6 develops these decouplings in detail.

3.3 Component Identity Ownership and Control

SADAR distinguishes three roles in the lifecycle of a component identity: the publisher, who creates and maintains the component and its manifest; the serving organization, which operates the component's runtime endpoints under publisher authorization; and the requester, which invokes the component at runtime. Each role has a distinct cryptographic identity and a distinct sphere of accountability.

Publisher Role

The publisher is the real-world accountable party for the component. The publisher:

- Signs the component's manifest with the publisher's signing key.
- Maintains the JWKS endpoint and rotates keys under the publisher's key management practices.
- Owns the publisher's component namespace; component identifiers are scoped within it.
- Authorizes endpoint URLs by listing them in the signed manifest. Endpoints not present in the manifest's authorized list are not publisher-authorized; consumers verify endpoint authorization

by validating the publisher's signature over the manifest. Endpoint list changes require a manifest version increment.

- Publishes manifest updates as new versions; once published, manifests are immutable.
- Issues lifecycle status changes (deprecation, revocation) as signed events.

The publisher does not necessarily operate the component's runtime endpoints. A component may be published by Publisher X and operated by Serving Organization Y under license, deployment partnership, or other arrangement. The publisher's signed manifest establishes what the component is, what it does, which endpoints are authorized to serve it, and what constraints apply to its use. The serving organization establishes how it runs at the URLs the publisher authorized.

Serving Organization Role

The serving organization is the party in control of a DNS name listed in the publisher's signed manifest. Its authorization to operate the component derives from inclusion of its URL in that signed list; its identity to consumers is established at the transport layer through standard TLS certificate practices.

The serving organization:

- Hosts the service endpoint and authentication endpoint at one or more URLs listed in the publisher's signed manifest.
- Presents valid TLS certificates for the listed hostnames using whatever certificate source is appropriate to its deployment — public CA, internal PKI, SPIFFE/SPIRE, or other. SADAR specifies no SADAR-specific PKI; certificate validation at the endpoint follows standard practice for the deployment context.
- Issues usage credentials to authenticated requesters — service-sovereign token issuance, in SADAR's terminology. The registry is not the issuer; each serving organization issues its own credentials under its own IAM.
- Defines and enforces local policies for which requesters are authorized to invoke the component.
- Manages session lifecycle within the bounds permitted by the manifest and by SADAR's lifecycle bounds (minimum 60 seconds, maximum 24 hours, recommended 15 minutes for usage tokens).
- Revokes credentials when its trust relationship with a requester is compromised.

Endpoint Trust Model

The mTLS trust model is anchored entirely in the publisher's signed manifest. The publisher's choice of URL is the trust declaration; the consumer's TLS handshake confirms that the party controlling the URL's DNS name is what answers at that URL. SADAR does not specify, require, or restrict the certificate provider or PKI used by operators. It requires only that:

- Publishers MUST list authorized endpoint URLs in the signed manifest.

- Consumers MUST validate that any endpoint they invoke appears in the publisher-authorized list for the manifest version they are operating against.
- TLS validation at the endpoint MUST follow standard practice for the deployment context.

If the manifest lists https://serving-partner.example/..., the publisher is attesting that the party controlling serving-partner.example is authorized to operate the component at that URL. The strength of this attestation is bounded by the publisher's diligence in URL selection. SADAR verifies that the publisher authorized the URL; it does not adjudicate whether the operator at the URL is honest, competent, or aligned with the publisher's intent. That transitive trust is the publisher's responsibility, maintained through its choice of operators and its discipline in keeping the authorized endpoint list accurate.

Consumers participating in operator-specific trust frameworks may apply additional validation when both sides are equipped to do so. Endpoints operated under SPIFFE/SPIRE, for example, may present SVIDs to SPIFFE-aware consumers without any change to the SADAR-level trust model — SADAR requires only that the URL appear in the publisher-signed list; the cryptographic substance of the TLS handshake at that URL is a transport-layer concern below SADAR.

This trust model preserves two foundational principles of the SADAR registry:

- **No runtime dependency on the registry beyond discovery.** Discovery results may be cached within manifest TTL bounds. The registry is not in the trust path at invocation time; the consumer's runtime validation against the publisher's signed manifest is sufficient.
- **No exploitation vector through registry contents.** The registry holds only publicly visible discovery content. It issues no credentials, holds no signing keys, and serves no role as a CA or trust authority. The registry's compromise affects discovery freshness but does not propagate into runtime trust decisions.

Accountability Separation

The separation of publisher, serving organization, and requester roles has practical consequences for accountability. A vulnerability discovered in the component's code is the publisher's responsibility to address. A compromise of a serving organization's deployment is the serving organization's responsibility. A breach in a requester's use of the component falls on the requester. Each role has identifiable accountability and a corresponding cryptographic identity in the SADAR architecture.

Requester Role

The requester is the party invoking the component. In agentic flows, the requester may be a human-initiated agent platform, an upstream agent, an orchestrating planner, or another component acting in the workflow. The requester:

- Resolves the component's manifest through the registry.
- Verifies the publisher's signature on the manifest.
- Selects an endpoint from the publisher-authorized list in the manifest.

- Authenticates to the serving organization's authentication endpoint and obtains a usage credential.
- Carries the appropriate SCT context with the invocation.
- Receives the response and emits the required observability spans.

In SADAR, the requester is not the originator. The originator is the human or system that initiated the overall workflow. The originator's identity and intent-scoped authority are carried in the SCT and propagate through every invocation, regardless of how many requesters have acted between the originator and the current invocation.

3.4 Identity Generation, Assignment, and Lifecycle Management

Component identities are generated at publish time and persist for the lifetime of the component version. Updates produce new versions with new identities; the old identity does not change. This section describes the lifecycle in detail.

Publication

A new component is published by:

1. Assigning a component identifier within the publisher's namespace.
2. Assigning a version, following semantic versioning conventions.
3. Constructing the manifest with all required elements: cryptographic anchor, identification metadata, capability and process semantics, authorized endpoint URLs, authentication endpoint references, and attestations.
4. Signing the manifest with the publisher's signing key.
5. Publishing the signed manifest to one or more registries.

Registries validate the manifest signature at ingestion. Registries authorized to publish discoverable content propagate the manifest to registries that have subscribed to or are federated with them. Replication is governed by the manifest's replication_seconds field (the registry-to-registry replication bound) and bilateral federation agreements between registries.

Lifecycle Axes

SADAR governance recognizes two independent lifecycle axes — one governing manifest cache validity, the other governing credential validity. They are distinct in mechanism, in authority, and in vocabulary, and must be kept distinct in implementation.

Manifest cache validity — the consumer-side lifecycle. The manifest itself does not expire; it is an immutable signed artifact, durable for as long as the publisher chooses to leave it published. The discovery_seconds field bounds how long a cached copy of the manifest may be relied upon before re-discovery is required. Past that bound, the holder must re-fetch from a registry. Re-discovery returns whatever the publisher currently has published — possibly the same manifest, possibly a new version,

possibly a signed deauthorization event indicating the component should no longer be used. Re-discovery is the mechanism through which all publisher-side state changes propagate to consumers; the cache validity bound is the worst-case propagation delay between a publisher action and consumer awareness.

Registry replication — a separate replication_seconds field bounds how long downstream registries may serve cached content from upstream registries before refreshing. Replication semantics shape the registry topology rather than the consumer cache; in deployments where the two bounds need not differ, replication_seconds defaults to discovery_seconds and need not be separately declared.

Credential validity — the serving organization's runtime lifecycle. Usage credentials issued by the serving organization carry an embedded expiry. Past expiry the credential is invalid; the consumer must reissue through fresh authentication. Credential validity bounds are subject to SADAR's session limits (60 seconds minimum, 24 hours maximum, 15 minutes recommended).

The two axes are decoupled. A serving organization may revoke credentials immediately when its trust relationship with a requester is compromised, without coordinating with any registry. A publisher may deauthorize a component without affecting in-flight credentials — the serving organization decides how to handle the deauthorization in its own time. A registry may distribute updated manifests without any change to credential validity at services that have already authenticated requesters under the prior version. Each authority operates on its own timeline under its own policy. No centralized coordinator must be online for any of them to act.

Soft and Hard Bounds

Both manifest cache validity and credential validity use a soft and hard bound model:

Soft bound — the point at which the cached value is considered stale and refresh is recommended. Past the soft bound, the holder should attempt refresh but may continue using the cached value if refresh fails.

Hard bound — the point at which the cached value must not be used. Past the hard bound, refresh is mandatory; the cached value is no longer valid.

This model provides resilience against transient registry or IAM unavailability without compromising freshness guarantees over longer periods.

Re-discovery and Reissuance

The vocabulary distinction matters because the mechanisms are different.

Re-discovery is the manifest-side mechanism. The consumer fetches the current manifest from a registry. Nothing is being renewed or reissued — the manifest is immutable per version, and what re-discovery returns is whatever state the publisher has currently published. The cached copy is replaced with the current state. If the publisher has issued a new version, the consumer now holds the new version. If the publisher has deauthorized the component, the consumer now holds that signal. If

nothing has changed, the cached copy is effectively refreshed with identical content. Re-discovery is a state observation, not a renewal.

Reissuance is the credential-side mechanism. The consumer authenticates to the serving organization through a fresh authentication event and receives a new credential with a new expiry. Reissuance is not renewal: the old credential is gone, a new credential is created, and the new credential is anchored in the state of identity, scope, and chain at the moment of reissuance.

This is the inverse of OAuth's refresh-token model. OAuth refresh preserves the authorization context — the grant captured at the original consent event — and has no concept of workflow context to preserve. SADAR reissuance preserves workflow context and re-evaluates authorization. The SCT and OTel baggage propagated with the invocation represent the originator's intent, the transaction instance, the accrued chain of custody, and the business process identifier — properties of the work being done, not of the credential being used to do it. These travel through reissuance unchanged. What reissuance re-derives is the authentication and authorization context for the current moment.

The consumer may reissue at any time before the credential's expiry. The operational characteristics of doing so — rate limits, authentication costs, minimum intervals — are declared as NFRs in the manifest and enforced by the serving organization. The protocol permits early reissuance; deployments tune practicality through declared NFRs.

Revocation Propagation Through the Runtime Error Path

On deprecation or revocation, the called endpoint returns a well-known error code indicating that the endpoint has been deprecated or revoked. The searchAndInvoke helper re-issues the discovery for either the same component with a new version or the full discovery lifecycle, per implementation choice.

searchAndInvoke similarly respects the publisher's discovery_seconds by performing a new search as needed.

Re-discovery is the proactive propagation path; the runtime error path is the backstop that ensures revoked components are not invoked even when the consumer's cache has not yet refreshed.

Credential Bindings

SADAR usage credentials carry three cryptographic bindings that together produce the protection properties the rest of the architecture relies upon:

- **Agent binding.** The credential is issued to a specific authenticated agent and is not usable by a different agent. Sender-bound proof-of-possession (DPoP, mTLS, or signed assertions) prevents bearer-token reuse by an unauthorized holder.
- **Originator binding.** The credential is bound to the originating user or system on whose behalf the agent is acting. The originator identity travels in the SCT.
- **Transaction binding.** The credential is bound to the transaction instance under which it was issued, identified by the transaction UUID carried in the SCT.

Because the credential is bound to a transaction UUID, it is valid only in the context of that transaction. It cannot be long-held and applied later to a different transaction. It cannot be replayed within the same transaction, because the SCT carries a per-invocation nonce that the serving organization tracks for the duration of the transaction. A different credential issued for the same user, the same agent, and the same declared intent cannot be substituted because the transaction UUIDs do not match. This protects integrity: even with identical starting context, the operational state matters for authorization of the next step.

The combination of these bindings is what makes SADAR credentials structurally different from OAuth tokens. An OAuth token authorizes "agent X may do Y" — bound to the agent and the scope, not to a specific workflow context. A SADAR credential authorizes "agent X, acting for originator O, in transaction T, at this point in the chain, may do Y." Credential capture by an attacker gives them the ability to invoke under that exact context only; they cannot generalize the credential to a different transaction, a different originator, or a different point in the chain. The blast radius of credential capture is structurally smaller than it is in OAuth.

Consumer Controls

The consumer may locally define a re-discovery interval and a reissuance interval for its own purposes. These operate independently of the publisher's discovery_seconds and the serving organization's credential expiry — neither side coordinates with the other. Each enforces its own clock against its own bounds.

The publisher's discovery_seconds is the upper bound on how long a consumer may treat a cached manifest as current. Past that bound, the consumer must re-discover, which propagates any publisher-side change to the consumer. The consumer's local re-discovery interval may be stricter — re-discovering hourly when the publisher permits 24 hours — for higher-assurance environments that want faster propagation of publisher actions. Local intervals longer than the publisher's bound have no effect, because cached content past discovery_seconds cannot be relied upon regardless of local configuration.

The serving organization's credential expiry is the upper bound on how long a credential remains valid. The consumer may force reissuance more frequently than the credential's expiry would require. Local intervals longer than the credential's expiry have no effect; the credential simply expires when its expiry says it does.

Neither side acknowledges the other's actions. When the publisher's bound triggers re-discovery, the consumer's local discovery interval is unaffected and continues counting on its own schedule (or may, at the consumer's discretion, be reset to align with the re-discovery event). When the consumer's local interval triggers re-discovery early, the publisher's bound is unaffected — the publisher does not know and does not care that an early re-discovery occurred. The same independence applies to credentials. The two systems operate in parallel under their own clocks; the effective behavior at any moment is governed by the earliest applicable bound.

Deauthorization and Revocation

Deauthorization at the registry layer signals that a component, a version, or a specific endpoint should no longer be discoverable or invoked. Publisher actions include:

- Retiring a component as a whole, by issuing a signed deauthorization event for the component identifier.
- Retiring a specific endpoint, by removing the URL from the manifest's authorized endpoint list in a manifest version increment.
- Retiring a component version, by issuing a signed deprecation event for that version.

Deauthorization events propagate through the registry topology with normal replication semantics and reach consumers at their next re-discovery.

Revocation at the credential layer is performed by the serving organization. The serving organization may revoke individual credentials (one requester), all credentials for a specific requester (broad revocation), or all credentials it has issued (component-wide revocation). Revocation propagates to the requester at the next invocation through standard OAuth introspection or token revocation flows.

The two acts are decoupled. A component may be deauthorized at the registry while ongoing credentials continue to be honored — the serving organization decides how to handle the transition. A serving organization may revoke all credentials while the component remains discoverable at the registry — indicating, for instance, that the deployment is in maintenance but not retired. The independence allows each authority to act under its own logic without forcing the other to react in lockstep.

3.5 Layered Controls and Runtime Composition

SADAR governance composes three layers of control. Each is owned by a different party, scopes a different aspect of the question "is this invocation appropriate?", and contributes to the authorization decision made by the enforcement layer at the moment of invocation. The layers are not redundant copies of one another, and they are not adversarial. They are complementary, with deliberate separation of concerns that allows each layer to evolve independently while producing coherent authorization decisions when composed at runtime.

3.5.1 The Three Layers

Process control is published by the owner of a process definition. It defines the sequencing of steps in a flow, the data that moves between them, and the expected ordering across the transaction as a whole. The vocabulary is process-classification IRIs drawn from established taxonomies such as APQC PCF, supplemented by enterprise-specific extensions where standard taxonomies are insufficient. A process definition expresses what should happen across a flow, in what order, with what data. It does not describe the components that execute the steps, and it does not encode enterprise policy; both are addressed by separate layers.

Agent manifest control is published by the component's publisher and travels with the component wherever it is deployed. The manifest advertises the capabilities of the entry it describes. For an agent

or tool, it describes capabilities using the same taxonomy that is used for processes — which is what makes planning possible: a planner reasoning over a process step x.y.z can discover candidate components through deterministic semantic matching, returning every component whose manifest advertises performs: [x.y.z]. Alongside positive capability declarations, the manifest carries declarations of non-action (does_not_perform) and prerequisite expectations (expects_completed), along with data interface and non-functional declarations. These describe the component itself and intentionally exclude any assumption about the surrounding process or the deploying enterprise's policy. This information hiding allows the same component to be discovered and used across many processes, in many enterprises, carrying its self-description and self-protection unchanged across all of them.

Enterprise control is owned by the enterprise operating the enforcement point. It expresses organization-specific policy that applies regardless of the process being executed or the components participating in it: entitlement constraints on the originator, asset-class restrictions, regulatory thresholds, monetary limits, separation-of-duties rules. Enterprise controls are the local sovereign's standing policies, applied to every invocation in the deployment regardless of source.

3.5.2 Composition at the Enforcement Point

At the moment of invocation, the enforcement layer evaluates all three control layers against the invocation context, together with the originator's current scope of authority. The composition is conjunctive: each layer can reject; no layer can unilaterally permit.

A component cannot grant itself permission by declaring no preconditions. A process cannot grant permission for a step that enterprise policy forbids. An enterprise cannot unilaterally invoke a component in violation of the component's declared preconditions or the process's declared sequencing. All three must permit, and the originator's authority must cover the invocation, for it to proceed.

This composition is what makes the architecture work. The originator is protected from any one layer being misconfigured, compromised, or maliciously authored, because no single layer can authorize an invocation in isolation. The control layers can be developed, deployed, and updated independently — each by its own owner — because the composition logic lives at the enforcement point and reconciles them at the moment of decision rather than at the moment of publication.

3.5.3 The Manifest's Two Audiences

The agent manifest serves two audiences with different temporal relationships to the invocation.

At plan time, a planner — an LLM-based reasoning loop, a deterministic selector, or any other component that constructs an execution plan — uses manifest declarations as discovery and selection criteria. The planner reasons forward: it needs something that performs x.y.z; the registry returns candidates whose manifests advertise that capability; among the candidates, this one requires a.b.c to have been performed first, so the planner either includes a.b.c earlier in the plan or selects a different candidate without that prerequisite. The manifest gives the planner the information needed to build a feasible plan.

At invocation time, the enforcement layer uses the same declarations as validation criteria against the actual invocation context. The chain of custody — the accrued SCT — shows what has been performed in this transaction. The manifest's expects_completed declarations are evaluated against that chain. Data inputs are evaluated against the manifest's data interface declarations. The operation being invoked is evaluated against the manifest's performs list. The enforcement layer does not re-do the planner's reasoning. It validates that what is actually happening is what the manifest declared could appropriately happen.

The same manifest data feeds both audiences; the audiences use it independently. The planner uses it to build something that should work. The enforcement layer uses it to verify that what is actually happening is consistent with what the publisher declared could appropriately happen. Enforcement does not trust the plan; it trusts the manifest and the chain of custody.

This separation has architectural value beyond the component's self-protection. A plan may be wrong. The originator may interrupt. An adversary may attempt to redirect a chain of invocations away from the planner's intended path. In none of these cases does the component's defense depend on the plan being correct, because enforcement is evaluating the actual chain, not the planner's intended one.

3.5.4 Taxonomy, Not Ontology

A natural question is why SADAR grounds capability declarations in taxonomies — APQC PCF for processes, O*NET-style classifications for agent actions, domain-specific classifications for specialized industries — rather than ontologies with formal relational structure.

Taxonomies describe the decomposition of work: what tasks exist, how they relate hierarchically, what identifiers refer to what operations. Ontologies layer relational structure over that: prerequisites, postconditions, data dependencies, sequencing constraints, equivalence relations.

The relational structure an ontology would express is real and important. But in SADAR's architecture, those relationships are not properties of the operations themselves. They are properties of how operations compose in a particular flow, in a particular deployment, under a particular enterprise's policies. The same operation may have one set of prerequisites in Enterprise A's procurement flow and a different set in Enterprise B's — not because the operation has changed, but because the surrounding composition has.

Baking those relationships into a fixed ontology would freeze one composition pattern as authoritative. Components would no longer be portable, because their ontological position would presuppose the flow they appear in. Open discovery and re-composition — finding a component that performs x.y.z and using it in a new context with new prerequisites — would require either re-ontologizing or working around the ontology. The very property that makes SADAR components discoverable and composable across contexts would be lost.

SADAR's design keeps the decomposition layer stable — the taxonomy of what tasks exist — and lets the composition emerge at runtime through the three control layers. Process control defines composition within a flow. Manifest control declares composition constraints from the component's perspective. Enterprise control imposes composition constraints from the deployment's perspective.

The enforcement layer composes them at invocation, against the chain of custody for this particular execution. This composition is dynamic, contextual, and reconciled at the moment of decision — which is exactly what a fixed ontology cannot be without becoming brittle.

This is also why the manifest's performs, does_not_perform, and expects_completed lists are framed as publisher-attested declarations rather than ontological assertions. The publisher describes the component's behavior in shared semantic terms drawn from a stable taxonomy. SADAR's three-layer composition does the relational work that an ontology would otherwise carry. The taxonomy supplies the vocabulary; the control layers supply the relationships.

4. The SADAR Architecture

SADAR is structured to make the authorization inputs (originator authority, intent, component identity, fit-to-purpose, sequence) verifiably available at every enforcement point, and to do so across organizational, cloud, and technical boundaries without requiring centralized coordination. The architectural principles on which SADAR rests were enumerated in Section 2.2 as the prerequisites authorization requires; this section presents the foundational pillars and the four-layer model that realize those prerequisites.

4.1 Foundational Pillars

The architecture rests on five technological pillars, all of which are operating standards that enterprise IAM already supports. SADAR does not introduce new cryptographic primitives, new identity protocols, or new transport layers; it composes existing standards in a specific way and adds three SADAR-specific artifacts (the manifest, the SCT, and the Telemetry Record).

OIDC and OAuth 2 with Extensions

OpenID Connect and OAuth 2 provide the authentication and credential issuance foundation. SADAR uses:

- OAuth 2 Client Credentials grant for service-to-service authentication.
- RFC 7523 Private Key JWT for assertion-based client authentication — the requester proves its identity by signing a JWT assertion with its private key, rather than presenting a shared secret.
- DPoP (RFC 9449) for participant binding where applicable, ensuring captured tokens cannot be replayed by other parties.
- Standard token introspection and revocation flows.
- The urn:sadar:scope:v1 scope namespace, defining the standard SADAR scopes for search, manifest resolution, registry listing, health, telemetry repatriation, and target invocation. Serving organizations may define additional component-specific scopes.

The IdP infrastructure each organization already operates serves as the issuer. Standard .well-known discovery semantics enable standard tooling. Grant types, token formats (JWT), and endpoint conventions are unchanged.

mTLS as the Authentication Baseline

Mutual TLS provides the channel-level authentication for SADAR interactions. The serving organization's endpoint and the requester both present X.509 certificates; channel-level authentication is established before any application-level credential is presented. This baseline holds across all SADAR participant interactions: registry-to-registry, requester-to-registry, requester-to-service, and serving-organization-to-IdP.

The certificate identity tied to a SADAR participant is verifiable against the participant's declared publisher identity — the same trust chain that anchors manifest signing extends to channel authentication.

JWS and JWE for Signed Artifacts

Manifests are signed with JWS (RFC 7515), using ES256 as the default algorithm. Other algorithms are permitted under the cryptographic baseline. The signature ties the manifest to the publisher's signing key, verified through the publisher's JWKS endpoint.

The SCT uses a JWS-inside-JWE structure: the SCT's claims are signed by the entity adding them, then the result is wrapped in JWE encryption. The encryption layer protects sensitive claims from intermediate parties; the signature layer ensures integrity and non-repudiation. Selective encryption — different claims encrypted to different parties — supports the privacy requirements of cross-organization workflows.

Usage keys are delivered as JWE-encrypted claims to the requester's public key, cryptographically binding the key to the requester's identity. JWKS endpoints are required at every entity, agent, tool, and registry, exposing at minimum one signing key and one encryption key. Key rotation is handled by JWKS endpoint update without manifest republication.

Cryptographic parity is required across the artifacts: baggage fields, span attributes, SCT claims, and Telemetry Record fields all use compatible signing and encryption algorithms, so an enforcement point that verifies one artifact can verify others without additional infrastructure.

OpenTelemetry for Observability

OpenTelemetry provides the observability transport, instrumentation API, and data model. SADAR adds normative requirements above OTel's baseline:

- Normative span attribute telemetry.origin.environment carrying entity_urn:agent_urn:environment_id, with both ends of every interaction emitting attributed spans for tamper-evident reconstruction.

- Normative baggage fields including the live Risk Score Adjustment list under urn:sadar:baggage:v1:risk_adjustments and the business process identifier.
- The per-invocation Telemetry Record — a separate persistent audit artifact populated through the SADAR Helper API, with phase-immutability across the Search, Selection, Invocation, and Outcome phases.
- The Repatriation protocol for governed cross-trust-boundary observability, triggered on span close when bilateral applicability is satisfied.

The OTel SDK, OTLP transport, W3C Trace Context propagation, sampling, batching, and exporter configuration are unchanged. Storage backend choice remains the implementer's decision.

Signed Manifests and the SCT

Two SADAR-specific artifacts complete the foundation:

Signed manifests — described in Section 3. The manifest is the component's identity, capability declaration, and authentication discovery payload, signed by the publisher and verified by the consumer.

The SADAR Context Token (SCT) — the propagated authorization artifact. The SCT carries the originator's identity, the declared intent, the intent-scoped scope of authority, and a cryptographically accruing chain of prior steps. Five chain operations govern how the SCT extends, suspends, and terminates through the workflow:

- **Open —** the framework signs the initial SCT at workflow inception, establishing the originator's identity, declared intent, and authority context. The Open operation is the trust root.
- **Continue —** each processing step adds an attestation to the SCT and re-signs it. The Continue operation extends the chain; the new SCT cryptographically incorporates the prior state.
- **Hold —** the SCT is suspended pending an asynchronous response. The Hold operation freezes the chain at a known state, allowing resumption when the response returns.
- **Authoritative Carry —** a designated authority (typically the framework or a policy authority) attests to the SCT's state in a way that downstream evaluators may rely on without re-verifying the upstream chain. Used in long-running workflows where chain length would otherwise grow unbounded.
- **Close —** the SCT terminates. Subsequent invocations require a new Open.

The SCT is bounded in size: it carries authorization context only, not data payloads. Data details live in OTel spans and the Telemetry Record, linked to the SCT by the workflow context. This separation is what keeps the SCT propagatable across deep compositions while supporting full audit detail through the observability layer.

4.2 Core Architectural Layers

The SADAR architecture is organized in four layers.

Layer 1 — Registry and Discovery

Responsible for component publication, manifest distribution, capability-aware discovery, and federation between registries.

- Registries ingest, validate, store, and serve component manifests. Six registry types are defined: Provider, Marketplace, Industry, Community, Internal Proprietary, and Registry of Registries (Section 6).
- Manifest distribution is governed by replication TTL, bilateral federation agreements, and the four distribution patterns (full subscription, partial subscription, forwarding, multi-hop forwarding) or by direct submission to the registry for home content.
- Discovery API — capability-aware search returning manifests matched against the requester's query criteria. Bilateral manifest matching: both the requesting party's requirements and the publishing party's declarations are evaluated for compatibility.
- Federation between registries uses mTLS plus OAuth Client Credentials. Each registry decides independently which registries to federate with and on what terms.

Layer 2 — Authentication and Authorization

Responsible for credential issuance, SCT generation and validation, and authorization context propagation.

- Framework — the workflow originator's infrastructure that issues the initial SCT under the Open chain operation. The framework binds the originator's identity, intent, and authority to the SCT root.
- Authentication endpoints — each serving organization's IdP, issuing usage credentials under service-sovereign token issuance. The registry is not the issuer.
- Credential binding — DPoP and the JWE-encrypted usage key bind credentials to the requester. Captured credentials cannot be replayed.
- SCT chain operations — the five operations described in Section 4.1 govern how authorization context extends through the workflow.

Layer 3 — Enforcement

Responsible for evaluating the authorization inputs and making the allow/deny decision at every invocation.

- Service policy engine — the serving organization's authorization decision point. Evaluates originator authority (from the SCT), declared intent (from the SCT), component identity (from the resolved manifest), fit-to-purpose (manifest vs. request), and sequence (SCT chain vs. manifest expects_completed).

- Helper API — the searchAndInvoke helper on the requester's side that orchestrates discovery, authentication, SCT chain extension, and invocation. The helper is not in the trust path; it facilitates the requester's side of the call.
- Policy authority — organizational policy governing what entitlements apply, what exceptions are permitted, and how the policy engine evaluates the inputs. SADAR provides the inputs; the organization defines the policy.

Layer 4 — Observability

Responsible for end-to-end audit, runtime monitoring inputs, and forensic reconstruction.

- OpenTelemetry spans — every component emits attributed spans for every invocation. Spans carry the normative telemetry.origin.environment attribute, workflow context, and SCT identifier.
- Baggage propagation — the workflow context, business process identifier, and Risk Score Adjustment list propagate through OTel baggage.
- Telemetry Record — the per-invocation persistent audit artifact, populated through the Helper API with phase-immutability across Search, Selection, Invocation, and Outcome phases.
- Repatriation protocol — governed cross-trust-boundary forwarding of spans to the originator's collector, triggered on span close when bilateral applicability is satisfied. Producer-side encryption and obfuscation applied before forwarding.

The four layers operate independently but are coupled by the workflow context. The SCT identifier appears in OTel baggage, span attributes, and the Telemetry Record. The manifest identifier appears in the SCT chain attestations and in span data. This linkage allows any layer's artifacts to be correlated with the others without coupling the layers operationally.

4.3 Zero Trust as Architecture, Not Principle

Zero Trust is commonly stated as a principle to be applied: never trust, always verify; least privilege; assume breach. Most frameworks adopt these principles and then design around them with varying degrees of compromise.

SADAR's position is that Zero Trust is not a choice in agentic systems; it is architecturally required. Three structural conditions make this so:

There is no application boundary between the LLM's planner and the components it calls. Components are directly invocable by whatever planner the LLM chooses. The traditional perimeter where authorization logic could be placed does not exist.

Composition is dynamic; pre-defined policies cannot cover it. Static policies presuppose known flows. There are no known flows. Each invocation must be evaluated on its own merits, with the originator's authority, the declared intent, and the runtime context as inputs.

Cross-organization operation is the case to solve. Mutual trust assumptions across organizations do not exist. Each party must authenticate the other directly and evaluate the request against its own policy, with verifiable inputs.

Given these conditions, Zero Trust is not an option — it is the only architecturally available position. SADAR's prerequisites are Zero Trust applied: bilateral authentication, time-of-use credentials, participant-managed session lifecycle, publisher-signed component contracts, and authorization decided at every invocation against verifiable inputs propagated with the workflow.

Explicit verification, least privilege, assume breach, micro-segmentation, data-centric security — each Zero Trust principle maps directly to a SADAR architectural property. The principles are not layered on top of SADAR; they are the architecture.

5. SADAR in the Authorization Process

This section details how SADAR operates in practice across the four major operational domains: fine-grained authorization at invocation, audit and provenance, real-time monitoring, and incident response. Each domain is illustrated with worked patterns showing how the architectural principles manifest in actual flow.

5.1 Fine-Grained Authorization at Invocation

Authorization in SADAR is assessed at every invocation, against the inputs propagated through the workflow context. The word "assessed" is intentional: it does not mandate a full authorization at every step, but it enables it for every step. The decision of what to authenticate and when is a risk-based policy decision within each organization. There is an inherent tradeoff between deep authentication of every action and establishing policies, practices, and implementations that balance risk with performance.

SADAR is not an authorization engine and not the decision-maker. SADAR's role is to provide the substrate that preserves the necessary context to support authorization across the agentic flow. The architecture and specific nature of how authorization occurs is outside SADAR's scope.

The subsections that follow trace the full path from workflow initiation through downstream invocation, illustrating how each input arrives at the enforcement point.

5.1.1 Workflow Initiation

A workflow begins when an originator — a human user, a scheduled task, or an external system — expresses an intent to a SADAR-aware framework. The framework:

6. Authenticates the originator through the organization's standard IdP.
7. Establishes the declared intent. The intent identifies the business process, the purpose, and any user-supplied constraints.

8. Resolves the originator's intent-scoped scope of authority. The scope of authority is what the originator is permitted to do, in this context, for this intent. It is determined by the organization's entitlements and the process definition for the declared intent.
9. Constructs the initial SCT under the Open chain operation. The SCT contains: originator identity, declared intent (URN reference to the process definition), intent-scoped scope of authority, workflow identifier, framework signature, and lifecycle bounds.
10. Initiates the workflow by invoking the first component — typically an orchestrating planner agent — with the SCT in context.

The Open operation is the cryptographic root of the workflow. Every subsequent invocation in the chain references back to this signed origin. Every enforcement point in the chain can verify that the originator's authority established at this moment has not been re-issued or reinterpreted downstream.

5.1.2 The searchAndInvoke Helper

searchAndInvoke is the SADAR-aware infrastructure component that performs the protocol-level work of discovery, authentication, and SCT construction on behalf of planner agents and other invoking components. Its purpose is to encapsulate SADAR-specific complexity so that agents themselves, and agentic frameworks such as LangChain and others, are abstracted from SADAR mechanics. Agents see a clean invocation interface and pass requests in their own vocabulary; searchAndInvoke translates those requests into registry queries, signed SCT chain extensions, authenticated invocations against serving organization endpoints, and verified responses.

The exception to this encapsulation is OpenTelemetry. The calling system must handle OpenTelemetry spans as normal and construct SADAR-compliant OTel baggage using a separate SADAR helper. OTel is a cross-cutting observability concern that cannot be hidden inside searchAndInvoke without breaking the calling system's tracing model.

This encapsulation is intentional. Agent implementations vary widely across platforms, frameworks, and reasoning approaches. Requiring every agent to implement SADAR cryptography, manifest validation, and chain signing would push SADAR-specific concerns into every agent codebase, undermine portability across platforms, and create a large attack surface composed of many independent SADAR implementations. By concentrating SADAR-specific behavior in searchAndInvoke, the trust model converges on a single component per deployment that the operating entity registers, publishes, and maintains under its own accountability.

searchAndInvoke as a Registered SADAR Component

searchAndInvoke is itself a SADAR-registered component. Its publisher — typically the entity operating the deployment in which it runs — registers searchAndInvoke in the registry with a signed manifest, a JWKS endpoint, and the keys under which it will operate.

The keys searchAndInvoke uses for chain signing, credential proof-of-possession, and identity assertions are the keys declared in its manifest. Trust in searchAndInvoke is the same kind of trust extended to any

registered component: anchored in publisher accountability, verifiable through manifest signature, and resolvable through JWKS at any time.

searchAndInvoke has no privileged trust role within the SADAR architecture. Its credibility derives from registration and publisher accountability, the same as any other component. A chain extension signed by searchAndInvoke is verifiable by the same registry lookup any verifier would perform for any other signature. There is no special "infrastructure key" or out-of-band trust extension. searchAndInvoke participates as a normal registered component performing a specific role.

The operating entity may register a single searchAndInvoke for an entire deployment, separate instances per cluster or tenant, or any other granularity appropriate to its operational model. Each registered instance has its own identity, its own keys, and its own manifest. Chain signatures identify which specific instance produced which link, with the same trust attribution available for any other component.

How Agents Reach searchAndInvoke

Agents must be able to find searchAndInvoke in order to use it, and the mechanism for that depends on the agent platform. In most deployments, the connection between an agent and the local searchAndInvoke is established by the platform itself: an MCP server reference, an agent framework configuration, a sidecar injection, a service mesh route, or similar platform-specific wiring. The agent receives a handle to searchAndInvoke as part of its operational environment and treats it as a standard tool. It does not need to perform SADAR discovery to find its own helper and, in fact, cannot, because searchAndInvoke encapsulates the discovery mechanics.

This wiring approach also allows the implementation of searchAndInvoke to evolve without affecting agent code. For conformance, searchAndInvoke must return consistent responses; changes to searchAndInvoke methods must be applied consistently across a federation to ensure consistent discovery and invocation behavior.

5.1.3 Discovery Patterns

Discovery in SADAR is the act of finding components — agents, tools, or resources — whose manifests advertise the capabilities a current need requires. The capabilities are expressed in shared taxonomies, which is what makes discovery deterministic: a request for a component that performs operation x.y.z resolves to every manifest advertising performs: [x.y.z] in the consulted registries.

All SADAR discovery flows through searchAndInvoke. The mechanism is uniform regardless of what initiated the discovery request. What varies between patterns is what drives the need, not how the need is fulfilled.

SADAR supports two discovery patterns: agents discovering within their own scope of reasoning, and orchestrators discovering against a SADAR process definition.

Agent's Scope

Each agent has a manifest. That manifest defines its capabilities to consumers and the semantic context in which those capabilities operate. For example, an agent defined to place an order might declare:

- APQC.org:CF PCF:Element ID 10279:vx — 4.2.4 Order materials and services
- APQC.org:CF PCF:Element ID 10295:vx — 4.2.4.4 Create/Distribute purchase orders
- X12.org:850:v5010.x

This advertises that the agent performs the step identified as 10295 in version x of the American Productivity and Quality Center's Cross-Functional Process Classification Framework. The agent uses the data definitions (meaning and structure) defined by X12.org in their 850 transaction (purchase order) in version 5010.x. The x in the version indicates compatibility with any minor version under major version 5010. (This is not the full normative structure and is illustrative only.)

These declarations definitively describe what the agent does and the semantic context in which it operates. The manifest may additionally declare:

- *Operations the component expects to have been completed before it can be invoked* (expects_completed). These are conditions on the chain of custody — "the chain must show these things were done before I can be invoked appropriately." A purchase order agent might require pricing verification (PCF 4.2.4.3) and inventory verification (4.2.4.1) to have been completed earlier in the chain.
- *Capabilities the component will invoke during its own work* (requires). These are dependencies the calling agent must satisfy by making the required capabilities available. An agent that intends to verify pricing during its own operation declares a requires entry for the pricing capability, allowing the calling agent to discover and provide a pricing component before invocation.

The distinction matters. expects_completed is a precondition on the chain of custody, enforced by the executor's policy engine against the SCT. requires is a dependency the calling agent satisfies by discovering and providing the needed capability at orchestration time.

Orchestration Against a Process Definition

A SADAR process definition specifies a flow as a directed graph of capability requirements. Each node identifies a capability and data-semantics requirement that must be filled by a SADAR component at runtime; edges express flow relationships between nodes — sequence, parallel fan-out, and convergence. The process definition is itself a SADAR-registered artifact, published, signed, and discoverable like any other component, but it describes a flow rather than a capability.

The notation is graph-theoretic, expressed as a set of directed edges in a Cypher-style syntax. The base pattern is:

```
(source) -> (target)
```

Source and target are nodes; the arrow indicates flow direction. Multiple targets in a single edge execute in parallel. Named join points let parallel branches converge before subsequent flow continues. Named subgraphs let pieces of process logic be defined once and reused —

```
(node) -> (:processOrder) -> (next)
```

substitutes the flow defined as processOrder at that point, allowing process definitions to be composed and decomposed.

The full process definition language is specified elsewhere; the notation above is enough to read the example that follows.

Each node carries the IRIs identifying the capability and data-semantics required for that step. To avoid syntactic clutter, IRI lists are wrapped in square brackets:

```
(self/start) -> ([APQC.org:CF PCF:Element ID 10294:vx – Solicit supplier quotes,
X12.org:840:v5010.x], [APQC.org:CF PCF:Element ID 10359:vx – Inventory check,
SAP.com:BAPI_MATERIAL_AVAILABILITY:vx])

([10294], [10359]) -> ([APQC.org:CF PCF:Element ID 10295:vx – Create purchase
order, X12.org:850:v5010.x])
```

This reads as: the originator initiates the process; quote solicitation and inventory verification execute in parallel; both must complete before purchase order creation begins.

The process definition does not name specific components. It specifies the capability required at each node. When the process executes, the orchestrator produces a registry query for each node as it becomes the next step to execute; candidates are returned; the selector chooses; the chosen component is invoked. The process definition is a template; the actual components are bound at runtime, and the bindings may differ between executions depending on which components are available, authorized, and selected for that execution.

This pattern composes with expects_completed declarations naturally. A component fulfilling a node may declare that it expects upstream operations to have been completed. The chain of custody produced by upstream execution will contain entries for those operations. The executor's policy engine validates expects_completed against the chain at invocation; the match succeeds; the invocation proceeds. The agent did not need to know the process definition would supply its preconditions, and the process definition did not need to know about the agent's preconditions. Each declared what it knew; the runtime composition reconciled them through the chain.

Regardless of which pattern initiated a discovery — agent's scope or orchestration — the downstream protocol mechanics are identical. The next subsection walks them.

5.1.4 Executing a Discovery and Invocation

The planner agent decides what to do next. It identifies the kind of capability it needs — drawn from its own reasoning, from a requires declaration, or from the next node in a process definition — and asks searchAndInvoke to find it. The request to searchAndInvoke carries the capability requirement and any agent-level preferences; searchAndInvoke translates the request into a registry query. The query carries:

- The required capability IRI.
- Optional preferences: provider, version constraints, compliance attestations, non-functional requirements.

- The SCT in baggage so the registry can scope discovery to authorized components for this originator and intent.

The registry returns one or more matching manifests, each signed by their respective publishers. searchAndInvoke validates manifest signatures and then calls the deployment's selector component, passing the candidate manifests for selection. The selector component is provided by the operating entity; every SADAR-compliant deployment must supply one. The selector makes the final selection from among the candidates and returns its choice to searchAndInvoke.

This division is intentional. Each using organization has its own perspective on risk tolerance and best understands its own risk landscape. The selector is the appropriate place for that judgment because it operates within the deployment's policy context. searchAndInvoke remains a mechanism helper, not a decision-maker: it validates, presents candidates, and acts on the selector's choice.

Once the component is selected, searchAndInvoke performs the protocol work to invoke it. searchAndInvoke authenticates to the selected component's authentication endpoint (the URL declared in the component's manifest, drawn from the publisher-authorized endpoint list). Authentication uses mTLS plus OIDC Client Credentials with RFC 7523 Private Key JWT, using the signing key declared in searchAndInvoke's own SADAR manifest. The serving organization's IdP issues a usage credential bound to searchAndInvoke as the calling party, with proof-of-possession via DPoP against searchAndInvoke's key and a JWE-encrypted usage key for response binding.

The credential is the physical authentication artifact: it establishes that this searchAndInvoke instance is permitted to communicate with this serving organization. Authorization scope for the specific work being done is carried separately in the SCT.

searchAndInvoke constructs the SCT chain extension for this invocation under the Continue chain operation. The extension carries:

- The originator identity and current authority, propagated from the incoming context.
- The transaction UUID.
- The planner identity, as a claim attesting to which agent initiated this step.
- The manifest reference for the selected component.
- The operation being invoked.
- A fresh nonce.
- A hash of the prior chain state, so the extension is bound into the chain as a tamper-evident sequence.

searchAndInvoke signs the extension with its own key. The serving organization will validate the credential, the proof-of-possession against searchAndInvoke's registered key, and the signed SCT chain — including the new extension's signature and its reference to prior chain state — as a unit. The credential establishes who is calling; the SCT establishes what work is being done, for whom, in what transaction, and at what point in the chain. The two are validated together; neither alone is sufficient.

If authentication and authorization succeed, searchAndInvoke invokes the selected component using the issued credential and the extended SCT, and returns the response to the planner. The planner does not see the credential, the SCT, or any protocol-level artifact; it sees the invocation result in its own vocabulary.

If the selected component is a resource rather than an executable capability, searchAndInvoke returns the resource to the planner without further authentication or authorization, because resources are not directly actioned. The planner may then select a tool and pass the resource to it, at which point a new invocation cycle begins through searchAndInvoke.

Selection is the deployment's decision through its selector; the registry returns candidates, not a binding answer. The planner identifies what it needs; the selector identifies which candidate to use; searchAndInvoke handles the protocol work that makes the invocation happen. Each layer has its own responsibility, and each operates within its own scope of authority and competence.

5.1.5 Enforcement at the Service

The service receives the invocation. Its policy engine evaluates the authorization inputs:

- **Originator authority —** the policy engine inspects the SCT root: who initiated the workflow, what intent was declared, what scope of authority was established. The signature on the Open operation is verified against the framework's key.
- **Intent —** the declared intent (URN reference) tells the policy engine what process this invocation is part of. The process definition (resolvable from the URN) describes the expected flow and constraints.
- **Component identity —** the service knows its own manifest. The policy engine confirms that the invocation is to the correct component and that the manifest version matches the version under which the requester authenticated.
- **Fit-to-purpose —** does this component's declared performs list align with what the workflow needs at this step? Does the invocation respect the component's does_not_perform exclusions?
- **Sequence —** does the SCT chain show that the component's expects_completed IRIs have been satisfied by prior steps?

If all inputs check out under the service's policy, the invocation proceeds. If any input fails, the service denies the invocation. The denial is logged and signaled to the requester through standard error response conventions.

The inputs compose conjunctively. Each can reject; none can unilaterally permit. The service's policy may evaluate additional inputs beyond these — enterprise policy, runtime risk signals, contextual factors — but the SADAR-supplied set is the minimum. They establish the baseline authorization decision; the service builds whatever policy it requires on top.

5.1.6 Multi-hop Workflows

When the invoked component itself needs to invoke a further component, the same flow repeats. searchAndInvoke at the invoking component's deployment:

- Discovers candidates through the registry for the required capability.
- Authenticates to the selected sub-component's endpoint.
- Extends the SCT under the Continue chain operation, signing the addition with its own key.
- Invokes the sub-component with the extended SCT and the usage credential.

Critically, the originator's identity, declared intent, and scope of authority remain cryptographically present at the SCT root throughout. The sub-component's enforcement point evaluates against the originator's authority — not against the immediate caller's claimed authority and not against any upstream interpretation.

This is the structural answer to the confused deputy problem. Each enforcement point evaluates the originator's actual authority for the declared intent against the specific invocation. Authority does not drift across hops because authority does not re-issue at each hop. A component cannot grant itself permissions it does not have by virtue of having been called by another component that does — the originator's authority is the cryptographic anchor, propagated unchanged through every Continue operation.

Long-running Workflows and the Authoritative Carry

Some workflows traverse many hops and would otherwise produce unbounded SCT chain length. The Authoritative Carry chain operation allows a designated authority — typically the framework or a designated policy authority — to attest to the SCT's state in a way that downstream evaluators rely on without re-verifying the full upstream chain.

Authoritative Carry is not a replacement for the chain; it is a compression mechanism. The authority signing the Authoritative Carry has, by virtue of its trust position, established that the chain to that point is valid. Downstream parties trust the authority's attestation. This is bounded use: only specific authorities may issue Authoritative Carry, and the framework's root signature remains the trust anchor.

5.1.7 Service-Sovereign Token Issuance

A consequence of how SADAR distributes authentication is that each service's usage credentials are scoped to that service, governed by that service's policies, and revocable by that service unilaterally. When a planner invokes ten different services across a workflow, searchAndInvoke holds ten different usage credentials on its behalf — each issued by the respective service's IdP, each bound to searchAndInvoke via DPoP, each subject to that service's lifecycle.

The registry does not see, hold, or mediate these credentials. The framework does not aggregate them. Each service's trust relationship with the calling deployment is its own. If service A revokes the deployment's credentials, services B through J are unaffected. If service B becomes compromised and must terminate all sessions, services A and C through J continue operating.

This independence is what makes SADAR's federation model practical at scale. No central coordinator must be online for any service to authorize, invoke, revoke, or refresh credentials. Each authority operates on its own timeline under its own policy, and the substrate composes their independent decisions through propagated workflow context rather than through coordinated state.

5.2 Audit, Provenance, and Repatriation

SADAR observability operates as a workflow property, not a component property. Components emit data about their participation; the workflow context propagates the linkage; the originator's collector receives the assembled trace through the Repatriation protocol.

OpenTelemetry Spans with Workflow Context

Every component participating in a SADAR workflow emits OpenTelemetry spans. Spans carry the normative telemetry.origin.environment attribute (entity_urn:agent_urn:environment_id), the workflow identifier from the SCT, the business process identifier, and standard OTel context (trace ID, span ID, parent span ID).

Both ends of every interaction emit attributed spans. This produces tamper-evident reconstruction: if a requester's span claims to have invoked a service and the service's span does not corroborate it, the discrepancy is detectable. The bilateral attestation pattern is fundamental to SADAR's audit guarantees.

The Telemetry Record

In addition to OTel spans, every SADAR invocation produces a Telemetry Record: a persistent per-invocation audit artifact populated through the SADAR Helper API. The Telemetry Record is structured in four phases, with phase-immutability — once a phase's data is recorded, it cannot be modified.

- **Search phase —** records the registry queries the requester issued, the candidates returned, and the selection criteria applied.
- **Selection phase —** records the chosen candidate, the verification of its manifest, and the requester's decision rationale.
- **Invocation phase —** records the authentication, the SCT extension, the request payload reference (typically a content hash), and the response.
- **Outcome phase —** records the final state of the invocation: success, error, or partial outcome, including any policy decisions made by the service.

Each phase is sealed when its data is complete. Sealing produces a cryptographic hash over the phase's contents, signed by the recording party. Any attempt to modify sealed phase content invalidates the seal. Phase-immutability provides cryptographic non-repudiation: the requester cannot later claim to have searched differently, the service cannot later claim to have decided differently. The Telemetry Record stands as a signed artifact of what occurred.

Repatriation

The Repatriation protocol forwards spans to the originator's collector under bilateral applicability rules. Repatriation triggers on span close — when a component finishes processing an invocation, it evaluates the bilateral applicability conditions and, if satisfied, forwards the span to the originator's designated collector.

Bilateral applicability has two conditions:

Producer policy permits repatriation. The producing organization's policy must allow forwarding spans to the originator. Producer policy may impose constraints: encryption requirements, field-level redaction, aggregation thresholds for k-anonymity, or outright denial for sensitive operations.

Originator collector accepts forwarding. The originator must have declared, in advance, that it accepts repatriated spans for workflows initiated under specific frameworks. This prevents unauthorized parties from injecting spans into an originator's collection.

When both conditions are met, the span is forwarded — encrypted, signed by the producer, and tagged with the workflow context. The originator's collector receives spans from all participating organizations as the workflow executes, producing a unified trace owned by the originator. The trace is correlated by workflow identifier; no post-hoc joining across separately-owned log stores is required.

Privacy Controls at the Producer

Sensitive data — PII, financial details, business confidential information — lives in the producing organization's spans and Telemetry Records. SADAR places privacy controls at the producer because the producer is the party that can correctly assess what is sensitive.

Producer-side controls include:

- Field-level redaction — specific fields removed or replaced with placeholders before forwarding.
- Field-level encryption — sensitive fields encrypted to keys held only by authorized originator-side analysts, with the rest of the span available for normal observability.
- Aggregation — individual invocations replaced with aggregated counts for forwarding, with raw spans retained only at the producer.
- Selective denial — spans for highly sensitive operations not forwarded at all; the producer retains them under its own audit policy.

The originator receives the trace its policy permits. Producer-side controls protect producer-side concerns. The split is honest: the producer cannot expose data it should not, and the originator cannot demand data it is not entitled to.

Forensic Reconstruction

When an incident requires forensic reconstruction, the originator's collector already holds the repatriated trace. Reconstruction queries the trace by workflow identifier and retrieves the complete sequence of invocations, decisions, and outcomes across all participating organizations. The Telemetry

Records add per-invocation detail — search candidates considered, manifests verified, authentication outcomes, policy decisions.

Cross-organization correlation is automatic because the workflow identifier appears in every span and every Telemetry Record. No legal process is required to obtain logs from each participating organization. No log joining is required across separately-owned log stores. The trace was assembled as the workflow executed; forensic reconstruction is a query, not an investigation.

5.3 Real-time Monitoring as Backstop

SADAR's preventive controls do the primary work of authorization. Detective controls — anomaly detection, behavioral baselines, trust scoring — operate as backstops for misconfiguration, novel attacks, and operational anomalies that fall outside the authorization model's scope. This subsection details how monitoring operates in SADAR and why its role is supplementary rather than primary.

What Monitoring Detects

Real-time monitoring is most valuable for:

- Policy misconfiguration — entitlements that grant too much, exclusions that fail to exclude what they should, intent definitions that allow invocations the organization did not intend.
- Implementation defects — components whose manifest declarations do not match their actual behavior; enforcement points that fail to evaluate all inputs correctly.
- Novel attack patterns — attacks that the architecture did not anticipate, particularly those that work within authorized authority and intent but exploit unforeseen composition patterns.
- Operational anomalies — performance degradation, error rate spikes, unusual traffic patterns that may indicate failure or compromise.

In each of these cases, monitoring is detecting departures from expected operation. The signal exists. Behavioral baselines, anomaly detection, and trust scoring can find it.

What Monitoring Does Not Detect Well

Monitoring is poorly suited to detecting authority-scope failures — the class of failures that arise when the originator's actual scope of authority for the declared intent was exceeded but the immediate component-level behavior was within profile.

Consider: a market analyst agent has permission to read market data. The agent reads market data — specifically, a competitor's confidential strategy report. The behavior is in-profile. The action type is correct. The credential is valid. The only thing wrong is that the originator's actual authority did not cover reading this specific data for this specific intent. There is no behavioral signature for this failure; it looks identical to legitimate market analysis.

Monitoring frameworks that depend on behavioral signals to catch authority-scope failures are asking the detection layer to do work it structurally cannot. SADAR prevents this class of failure at the

enforcement point by propagating the originator's intent-scoped authority and evaluating it against every invocation. Detection is unnecessary because the failure does not occur.

The Risk Score Adjustment List

SADAR specifies a normative baggage field, urn:sadar:baggage:v1:risk_adjustments, carrying a live Risk Score Adjustment list. The list is updated as the workflow executes; downstream policy engines may consult it as one input to their decisions.

Risk adjustments capture runtime context that the SCT itself does not carry: elevated risk from recent anomalous activity, organizational risk posture changes, regulatory restrictions activated by a current event. The risk_adjustments baggage allows the framework's real-time view of risk to inform downstream policy engines without requiring those engines to query a centralized risk service.

Risk adjustments are advisory, not authoritative. The policy engine decides how to incorporate them into its decision — perhaps by requiring stronger authentication when risk is elevated, perhaps by lowering permitted action thresholds, perhaps by routing decisions to human review. The risk_adjustments baggage provides the signal; the policy engine decides what to do with it.

SIEM and SOC Integration

SADAR's observability data — OTel spans, Telemetry Records, and the assembled trace — feeds standard SIEM and SOC tooling. The repatriated trace at the originator's collector is the primary source for security operations: cross-organization correlation is already done, workflow context is preserved, and standard observability tooling can query and analyze.

Behavioral analytics, anomaly detection, and threat hunting all operate against this data. The architectural value is that the data is already correlated and centralized at the originator, rather than fragmented across producing components and organizations. SOC analysts work with traces, not log joins.

5.4 Incident Response

Incident response in SADAR draws on the substrate's separation of authorities and the workflow-centric observability model. This subsection details how identification, containment, eradication, and recovery operate.

Identification Through the SCT

When an alert fires — from monitoring, from a manual report, or from external threat intelligence — it typically references a specific workflow or invocation. The SCT identifier and the workflow identifier in the alert tie immediately to:

- The originator who initiated the workflow.
- The declared intent and scope of authority.
- The complete chain of components that have processed the workflow.

- The repatriated trace and Telemetry Records for forensic detail.

Identification is immediate. The full workflow context is in the SCT; the full operational detail is in the trace. No multi-organization log discovery is required to understand what happened.

Containment Through Service-Sovereign Revocation

Each serving organization can revoke usage credentials it has issued without coordinating through any central authority. When a service determines that a requester's credentials should be revoked — because of detected anomalous behavior, because of an external alert, because of an internal policy decision — the service revokes immediately.

Revocation propagates to the requester on the requester's next attempted invocation. Standard OAuth token revocation flows apply: the requester's usage credential fails introspection, the request is denied, and the requester must re-authenticate to obtain new credentials — if the service's policy permits.

The two-axis lifecycle supports nuanced containment. A serving organization can revoke session credentials while leaving the component itself discoverable at the registry (indicating the deployment is in maintenance, not retired). Or the publisher can deauthorize the component at the registry while serving organizations continue handling in-flight sessions under their own policy. Each authority acts on its own timeline under its own logic.

Containment Through SCT Lifecycle

In addition to service-side revocation, the framework that issued the original SCT can terminate the workflow under the Close chain operation. Closing the SCT signals to all participating components that the workflow is no longer authorized; subsequent invocations referencing the closed SCT are denied.

The Close operation is propagated through normal observability channels. Participating components see the closure in baggage on subsequent invocations and through the standard SCT validation that occurs at every enforcement point. Closure is not instantaneous — in-flight invocations may complete — but no new invocations are admitted under the closed SCT.

Eradication and Recovery

Eradication addresses the root cause: a vulnerability patched, a misconfigured policy corrected, a compromised credential rotated, a compromised component replaced. SADAR's component identity model supports clean eradication because components are immutable; a vulnerability in version X is addressed by publishing version Y, and existing sessions under version X can be revoked while version Y becomes the active deployment.

Recovery returns the system to normal operation. The forensic trace at the originator provides the basis for confirming that all affected workflows have been identified, all compromised credentials revoked, all impacted data inventoried. Recovery is informed by complete forensic detail, not by partial logs gathered through cross-organization discovery.

Closing the Worked Example

Returning to the PocketOS incident introduced in Section 2: an incident of that class is structurally prevented under a SADAR substrate. The Cursor-hosted agent attempting the destructive deletion would have carried an SCT whose Open operation declared the originator's intent (a routine staging task) and the intent-scoped scope of authority (exploration and analysis, not production data destruction). The Railway service receiving the volumeDelete invocation would have evaluated the request against the propagated authority context. Deletion of production data exceeds the declared intent's scope. The request would have been denied at the enforcement point before any data was touched — regardless of the breadth of the API token the agent found in the config file.

The agent's reasoning would not have mattered. The breadth of the API token would not have mattered. The instructions in the system prompt would not have mattered. Each of those is a property of the calling side; the enforcement decision is made by the service holding the data, against the originator's actual authority for the declared intent, carried in a credential the service can verify directly. Prevention happens at the enforcement point because the inputs prevention requires have been propagated to the enforcement point. Detection becomes a backstop for misconfiguration and novel attacks, not the primary defense against the routine authority-scope failures the current architecture admits.

5.5 Other Uses of the SADAR Substrate

The SADAR substrate — verifiable component identity, propagated authority, bilateral trust, and repatriated observability — supports a range of capabilities beyond core authorization. This subsection sketches several.

Reputation and Trust Brokering

Component identifiers are stable anchors for reputation. A consuming organization can record its experiences with components — reliability, accuracy, conformance to declared semantics — and share these records with peers or specialized reputation services. Reputation attestations are themselves signed and verifiable, and they can be discovered alongside manifests through the registry.

This produces a richer discovery experience: requesters can prefer components with attestations from sources they trust, can avoid components flagged by peers, and can incorporate reputation as one input to their selection decisions. The substrate provides the infrastructure; specialized reputation services build on top.

Billing and Quota Enforcement

For commercial components, the Telemetry Record provides the per-invocation audit trail that supports usage-based billing. The signed record establishes that the invocation occurred, when, and at whose request — sufficient evidence for billing reconciliation.

Quota enforcement operates through the same mechanism. A serving organization can issue usage credentials with embedded quota constraints; the service's policy engine enforces against the quota; the Telemetry Record captures the usage for retrospective verification.

Supply Chain Attestations

Components carry attestations beyond the publisher's manifest signature. AIBOM attestations declare the supply chain (training data provenance, dependencies, build pipeline). Conformance attestations confirm SADAR conformance level. Certification attestations confirm regulatory compliance (HIPAA, SOC 2, EU AI Act risk category).

These attestations are discoverable alongside the manifest. A requester whose policy requires HIPAA-compliant components can filter discovery results to components carrying current HIPAA attestations from accredited auditors. The substrate carries the evidence; the requester's policy decides what to require.

Process Definitions for Planner Agents

Process definitions — themselves manifests — serve as reusable execution-plan templates for planner agents. A planner whose declared intent matches a known process definition can use that definition as a starting plan, adapting as it discovers components and runs. The process definition's declared preconditions, sequencing, and exclusions inform the planner's decisions.

This provides a bridge between procedural enterprise process libraries (BPMN models, process maps, compliance flows) and the dynamic composition of agentic systems. The process definition is not enforced as a static flow; it is consumed as guidance. The planner may follow it, deviate from it when discovery suggests a better path, or compose multiple definitions together.

Ethical and Governance Services

Specialized governance components — ethical compliance oracles, bias review services, fairness audit services — publish their own manifests. Workflows that should consult such services for high-stakes decisions can invoke them through the same SADAR substrate, with the same authorization, audit, and trust properties. The substrate does not specify which governance services exist; it provides the infrastructure on which they operate.

6. Deployment Models and Governance

SADAR is designed around federation as the primary deployment mode, with bilateral trust between participating registries enabling cross-organization operation without centralized coordination. This section presents the registry topology, the bilateral trust model, and the governance considerations that shape how organizations adopt and operate SADAR.

6.1 Federation as Primary Deployment Mode

Where many frameworks present federation as one deployment option among several, SADAR treats federation as the default. The architectural choices that support this — bilateral trust, service-sovereign

credential issuance, repatriated observability, decoupled authority lifecycles — all converge on the same point: cross-organization operation is the case to solve.

Three observations motivate this position:

Single-organization deployments are a special case of federation. An organization operating an entirely internal SADAR deployment is, architecturally, a federation of one. The same substrate that supports cross-organization operation also supports internal-only operation — with the same primitives and the same operational characteristics.

Real agentic workflows cross boundaries. Production agentic systems compose across cloud providers, vendor APIs, partner services, and increasingly cross-organization collaborations. A framework that treats these crossings as edge cases will accommodate them with friction. A framework that treats them as the default will operate them natively.

Sovereign deployment is non-negotiable in many sectors. Regulated industries (healthcare, finance, government) and security-sensitive sectors (defense, critical infrastructure) require deployment where the organization controls its own infrastructure end-to-end. Cross-organization participation must be possible without compromising sovereignty. SADAR's bilateral model supports this directly.

6.2 The Registry Types

SADAR specifies six registry types, each serving a different role in the ecosystem. The types are not exclusive — an organization may operate multiple types — and they reflect the actual structure of how component discovery happens in real-world deployments.

Provider Registries

Operated by infrastructure providers — cloud platforms, model providers, agent runtime vendors. Provider registries list the components the provider hosts or operates. They are the natural starting point for component publication when the publisher and the runtime host are the same organization.

Marketplace Registries

Commercial registries listing components for purchase, subscription, or licensed use. Marketplace registries may aggregate components from many publishers, handle commercial discovery (pricing, terms), and broker access between consumers and publishers. They are the analog of app stores or API marketplaces.

Industry Registries

Vertical-specific registries operated by industry consortia or specialized providers. A healthcare industry registry might list HIPAA-attested components; a financial services registry might list components with relevant regulatory compliance. Industry registries impose membership criteria and quality bars appropriate to their domain.

Community Registries

Consortium-operated registries serving a specific community: an open-source ecosystem, a research collaboration, a standards body's reference components. Community registries typically have more permissive publication criteria than industry registries and serve discovery within a defined community.

Internal Proprietary Registries

Single-organization registries listing components developed and operated within one organization. Internal Proprietary registries are not discoverable from outside the organization — they are not federated for outbound discovery — but they may consume from registries the organization has authorized to receive from. This allows an organization to use external components internally without exposing its own components externally.

Internal Proprietary registries are not listed in the Registry of Registries and cannot serve as home registries for publicly discoverable content. The governance boundary is explicit: private deployments participate as consumers but do not contribute to the public ecosystem unless they publish through another registry type.

Registry of Registries

The Registry of Registries (RoR) is the bootstrap mechanism: a registry whose entries are themselves registries. The RoR allows discovery of authoritative registries for specific domains, industries, or communities. It is the well-known starting point for agents and organizations new to a particular SADAR ecosystem.

The RoR is not centralized in the sense of a single global authority. Multiple RoR instances may exist, operated by different parties for different purposes (an industry RoR for healthcare, a national RoR for a country's government services, etc.). RoRs themselves are registries and are discoverable through ordinary SADAR mechanisms.

6.3 Bilateral Trust Establishment

Federation between any two registries is established bilaterally. There is no consortium-level governance to join, no shared trust list to participate in, no federation-wide policy to comply with. Each registry decides independently which other registries to federate with and on what terms.

The Bilateral Trust Mechanism

Trust between two registries is established through:

- Mutual identity verification — each registry validates the other's real-world organizational identity through whatever means it considers sufficient (publisher key trust, extended-validation certificates, out-of-band verification).
- mTLS plus OAuth Client Credentials — the runtime authentication mechanism between the registries.

- Bilateral federation agreement — a signed agreement specifying what each registry will share with the other: full subscription, partial subscription, forwarding rules, multi-hop forwarding permissions.
- Replication parameters — the operational parameters for federation: replication frequency, conflict resolution, attestation propagation.

Once established, federation operates without further coordination. Each registry propagates updates under the federation agreement; either side may terminate the federation by ending the agreement and revoking the runtime credentials.

Multi-hop Federation

Federation may be multi-hop: Registry A federates with Registry B, which federates with Registry C, allowing A's consumers to discover components published in C. Multi-hop is governed by forwarding permissions in the federation agreements; A's agreement with B specifies whether B may forward A's queries onward and under what conditions.

The bilateral model means multi-hop chains can be arbitrarily complex without requiring any global coordination. Each link in the chain is governed by the agreement between its endpoints. Trust composes through the chain: A trusts B, B trusts C, so A may trust C's answers about components C lists — but A may choose to verify C's entries against C's direct attestations rather than relying on B's forwarding.

6.4 Three Decoupled Normative Concepts

SADAR treats conformance, certification, and authorization as three independent normative concepts. Each is governed separately, evaluated separately, and revoked separately. This decoupling has substantial consequences for how organizations adopt and operate SADAR.

Conformance

Conformance is a structural property: the component implements the SADAR specification correctly. A conformant component:

- Publishes a valid signed manifest with all required fields.
- Honors the publisher signature, including key rotation through JWKS.
- Exposes the required SADAR endpoints (authentication, invocation, JWKS).
- Issues and validates SCTs correctly under the chain operations.
- Emits OpenTelemetry spans with the normative attributes and baggage.
- Participates in repatriation under the bilateral applicability rules.

Conformance is a yes-or-no determination. A component either implements SADAR correctly or it does not. The R-CONF normative requirements specify what conformance entails. Conformance attestations may be issued by accredited test suites or by self-attestation.

Certification

Certification is an external attestation of compliance with a specific standard or regulatory regime. A component may be certified for HIPAA, SOC 2, PCI DSS, EU AI Act risk categories, or industry-specific standards. Certification is issued by an accredited certifying body and references the specific component identifier and version.

Certification is independent of conformance. A component may be SADAR-conformant without being certified for any external regime. A component may be HIPAA-certified for use in healthcare contexts but not certified for financial services. Certifications are scoped to specific regimes and may have their own expiration, renewal, and revocation lifecycles.

R-CERT normative requirements specify how certification attestations are structured, signed, discovered, and validated. Multiple certifications may attach to a single component; each is independently revocable.

Authorization

Authorization is the determination by a specific consuming organization, registry, or service that a specific component may be used within a specific context. Authorization is local to the deciding party and may be revoked unilaterally.

A component may be conformant and certified but not authorized for use by a particular organization — because the organization has chosen a different vendor, because the component does not meet the organization's additional requirements, or because the organization is in a quiet period before adopting it. Authorization reflects deployment decisions, not structural properties.

R-AUTH normative requirements specify how authorizations are issued, communicated, propagated through the registry topology, and revoked. R-ELIG — the related eligibility requirements — specifies what conditions must be true for a component to be eligible for authorization (typically including conformance, possibly including specific certifications).

Why Decoupling Matters

Conflating these three concepts — as some frameworks do — produces governance and operational problems.

If certification is bundled into identity (as in capability VCs that include compliance attestations), then revoking certification revokes identity, which may have unintended consequences for in-flight workflows. SADAR's decoupling allows certification to be revoked without disrupting the component's identity — the component remains discoverable and operable; only its claim to specific compliance is rescinded.

If authorization is bundled into conformance (as in framework models where deployment-eligible means specification-compliant), then organizations cannot easily restrict their internal use to a curated subset of conformant components. SADAR's decoupling allows an organization to authorize only specific components for its internal use, regardless of whether others are conformant and certified.

If conformance is bundled into authorization (as in deployment models where being authorized requires being conformant), then conformance becomes a gate for participation rather than a structural property. SADAR's decoupling allows non-conformant components to exist in a registry for purposes of experimentation, development, or migration — without being authorized for production use — and allows authorization to evolve independently of conformance status.

Each of the three concepts can be revoked independently. Each is governed by the appropriate authority: conformance by the testing or certifying body, certification by the regulatory authority, authorization by the deploying organization. The R-CONF, R-CERT, R-AUTH, and R-ELIG requirement families specify the normative interactions.

6.5 Governance Considerations

Beyond the decoupling of normative concepts, several governance considerations shape SADAR adoption.

Publisher Accountability

The publisher of a component is a real-world accountable party. The publisher's identity is verifiable, the publisher's signing key is auditable, and the publisher bears responsibility for the component's declared behavior. This is different from self-issued identity models where the component's controller may be opaque.

Publisher accountability supports incident response (the publisher is the party to engage for vulnerability disclosure), regulatory compliance (the publisher is the party of record for compliance questions), and reputation (consumers know who they are relying on).

Registry Governance

Each registry is governed by its operating organization under that organization's policies. Industry registries may impose membership criteria, quality bars, or certification requirements. Community registries may use community governance models. Marketplace registries operate under commercial terms. Provider and Internal Proprietary registries operate under their respective single-organization governance.

No global registry governance exists or is required. Cross-registry interactions are governed by bilateral federation agreements between the participating registries. This avoids the lowest-common-denominator problem that often arises in federated systems where shared governance must satisfy the most restrictive participant.

Disputes and Dispute Resolution

Disputes — over component behavior, attestation accuracy, registry decisions, or federation terms — are resolved bilaterally between the parties involved. SADAR does not specify a global dispute resolution

mechanism because none is appropriate; the substrate operates across many legal jurisdictions, regulatory regimes, and commercial relationships.

What SADAR does provide is the evidentiary substrate. Signed manifests, signed SCTs, signed Telemetry Records, and repatriated traces produce non-repudiable evidence of what occurred. Parties to a dispute have verifiable evidence to support their positions. Resolution happens through whatever forum the parties agree to or are subject to: contract law, regulatory adjudication, industry arbitration.

Privacy and Data Governance

Producer-side privacy controls allow each organization to govern what data flows out under repatriation. This aligns with data sovereignty requirements: each organization decides what crosses its boundary, in what form, to whom, and under what conditions.

Sensitive data minimization — redaction, encryption to authorized recipients only, aggregation — happens at the producer. Originators receive the data their producers' policies permit. This makes data privacy governable at the source rather than requiring centralized control over what is forwarded.

Evolution and Versioning

The SADAR specification itself versions. Components declare which specification version they conform to in their manifest. Registries declare which versions they accept and propagate. This allows controlled evolution: new versions can be adopted incrementally without breaking existing deployments, and deprecation can proceed under clear timelines.

Cognita AI Inc., as the founder and steward of SADAR, governs the specification through the OpenSemantics.org publication channel under the Community Specification License 1.0. The license includes royalty-free patent grants for Necessary Claims. Future governance may include membership models for organizations contributing to the specification's evolution.

7. Security Considerations

Securing SADAR involves both the substrate's own integrity and the threats SADAR is designed to address in the agentic systems built upon it. This section uses the MAESTRO threat model for structure, mirroring the CSA paper's approach, but emphasizes where SADAR's architectural choices prevent threat classes structurally rather than addressing them through layered detective controls.

This section is not an exhaustive threat-modeling exercise. It is a first-pass application of a third-party threat model — MAESTRO — to the SADAR substrate, identifying where SADAR's architectural choices prevent threat classes structurally and where SADAR provides the substrate for detective controls. A similar exercise has been conducted against the FINOS AI Governance Framework v2 risk catalogue; that analysis is published at https://www.opensemantics.org/documents/sadar-addresses-governance-gaps-identified-in-the-finos-ai-governance-framework-for-discovery-and-authorization. Both exercises are first-cut mappings rather than complete adversarial analyses; deeper threat modeling — including

formal analysis under adversarial composition and producer-side compromise — is identified as future work in Section 9.

7.1 The MAESTRO 7-Layer Reference Architecture

MAESTRO decomposes agentic AI ecosystems into seven layers: foundation models, data operations, agent frameworks, deployment and infrastructure, evaluation and observability, security and compliance (vertical), and agent ecosystem. SADAR addresses threats across these layers through a mix of preventive controls (where the architecture eliminates the threat class) and detective controls (where the substrate provides the data feed for monitoring and response).

The defining characteristic of SADAR's threat model is that it prevents whole classes of threats at the architectural level rather than layering controls on top of an architecture that admits them. This is most visible in the authority-related threat classes: confused deputy, delegation escalation, scope creep, and impersonation through delegation. Each of these has a structural prevention in SADAR; detective controls operate as backstops.

7.2 Threat Analysis

The following table maps key threat classes to their SADAR mitigations, identifying which threats are prevented architecturally and which are addressed through additional controls.

Threat	SADAR Mitigation	Control Type
Component impersonation	Publisher signature on manifest; verifiable publisher identity; key rotation through JWKS	Preventive — invalid signatures are detected at manifest resolution
Credential replay	DPoP-bound credentials; JWE-encrypted usage keys bound to requester's public key	Preventive — captured credentials cannot be replayed by other parties
Delegation escalation	SCT propagates originator's intent-scoped authority directly; no re-issuance at hops	Preventive — authority does not drift across delegation
Confused deputy	Originator's authority evaluable at every enforcement point	Preventive — enforcement evaluates originator's actual authority, not caller's claim
Scope creep / prompt injection effects on authorization	Authority surface bounded by SCT; LLM-driven actions cannot escalate beyond the originator's scope	Preventive — LLM compromise does not enable authority exceedance
Manifest tampering	Manifests signed; tampering detected at signature verification	Preventive — modified manifests fail validation
Registry compromise	Bilateral trust; downstream registries verify publisher signatures directly	Compromise scope limited — a compromised registry can deny service but cannot forge attestations

Threat	SADAR Mitigation	Control Type
Token theft	Service-sovereign issuance; service can revoke unilaterally; lifecycle bounds limit window	Containment — stolen tokens have bounded validity and can be revoked without coordination
Cross-org data exfiltration	Producer-side encryption and redaction in repatriated spans	Preventive — producer governs what crosses its boundary
Audit log tampering	Signed spans and Telemetry Records; bilateral attestation pattern	Detective — discrepancies between requester and service attestations are detectable
DDoS against IAM services	Distributed enforcement; no single coordinator must be online	Mitigated — individual service or registry outages do not stop the system
Policy misconfiguration	Detective via monitoring and risk score adjustments	Detective — SADAR provides observability; organization defines monitoring
Novel attack patterns	Detective via observability and SIEM integration	Detective — backstop for threats outside the architectural model
Data poisoning of agent training	Out of scope for the authorization substrate	Addressed at the model lifecycle layer, not in SADAR

7.3 Cross-Layer Threats

Several threat classes span multiple MAESTRO layers and require coordinated address.

Supply chain attacks on components. A compromised publisher could publish a malicious component. SADAR's defenses: publisher identity verification, supply chain attestations (AIBOM, build-pipeline signatures), reputation systems, and the inability of any single registry to forge publisher signatures. Defense in depth combines publisher accountability with attestation verification.

Privilege escalation across enforcement points. A compromised enforcement point could grant invocations its policy would otherwise deny. SADAR's defenses: enforcement points evaluate against the SCT (not against locally-cached state), repatriated observability provides bilateral attestation of every invocation, and the originator can detect inconsistencies between expected and actual workflow behavior. Compromise at one enforcement point is bounded; it cannot rewrite history.

Goal misalignment leading to authorized misuse. An agent that has been compromised through prompt injection or training data poisoning may act with valid credentials in ways that exceed its actual authority. SADAR's defenses: the SCT bounds the authority surface regardless of what the agent attempts; the does_not_perform manifest declarations bound legitimate component behavior;

observability detects departures from expected operation. The compromised agent's actions cannot escalate beyond the authority propagated to it.

7.4 Zero Trust as Architecture

Section 4.3 develops the position that Zero Trust is architecturally required in agentic systems, not a chosen principle. The security consequences of this architectural commitment are concrete:

- Explicit verification — every invocation triggers verification of identity, authority, fit-to-purpose, and sequence. No invocation is admitted on the strength of prior verification.
- Least privilege — usage credentials are scoped narrowly to specific invocations under the SCT's authority context. Broad credentials do not exist; every credential is bounded.
- Assume breach — the architecture operates correctly even if any single component, registry, or service is compromised. Bilateral attestation makes compromise detectable; service-sovereign revocation contains it.
- Micro-segmentation — each service operates its own enforcement perimeter. Compromise at one service does not propagate to others.
- Data-centric security — producer-side privacy controls govern what data crosses trust boundaries. Sensitive data is controlled at its origin, not by downstream parties.

These properties are not Zero Trust principles applied to a SADAR architecture; they are SADAR architectural properties that happen to satisfy Zero Trust requirements. The substrate cannot operate any other way.

7.5 Enterprise Security Use Cases

SADAR's observability and substrate properties support a range of enterprise security operations.

Use case	SADAR support
Agentic red teaming	Repatriated traces provide ground truth for offensive campaign analysis; the originator sees the actual workflow paths attackers exercised.
Security chaos engineering	Bilateral attestation pattern allows controlled failure injection (denied invocations, revoked credentials, corrupted manifests) with verifiable impact assessment.
Agent trust boundary validation	The SCT chain provides verifiable evidence of where authority crossed boundaries and how it was scoped at each crossing.
Data leakage risk analysis	Producer-side privacy controls and repatriation policy together govern what data crosses boundaries; risk analysis evaluates the policy and the observed flows.
LLM plugin and tooling risk assessment	Manifest declarations (performs, does_not_perform) make the intended behavior of each component explicit; deviations from declared behavior are detectable through observability.

Use case	SADAR support
AI identity abuse simulation	Service-sovereign revocation and SCT lifecycle allow controlled simulation of credential compromise, with verifiable containment outcomes.
Compliance evidence generation	Telemetry Records and repatriated traces produce signed, non-repudiable evidence supporting compliance audits across regulatory regimes.
Cross-organization incident response	The repatriated trace at the originator's collector is the unified incident view; cross-organization log discovery and joining are not required.

8. Innovative Contributions

SADAR's contributions are organized around the central architectural commitment: authorization is the load-bearing problem in agentic systems, and the substrate must propagate the inputs authorization requires to every enforcement point. This section names the innovations that follow from that commitment.

Authorization Substrate, Not IAM Framework

SADAR is positioned as an authorization substrate, not a complete Identity and Access Management framework. The distinction is consequential: a substrate supplies the prerequisites that authorization requires without specifying policy, deployment, or governance. This allows SADAR to interoperate with diverse policy engines, deployment topologies, and governance models — including operating on top of an IAM framework like CSA's, where the IAM framework supplies identity and the substrate supplies authorization.

The contribution is the clean separation of concerns. Identity handles attribution and authentication. Authorization handles decisions at invocation. Policy handles entitlements and exceptions. Each layer does its own work, and each can evolve independently.

The SCT and the Chain Operations

The SADAR Context Token is the load-bearing innovation. The SCT propagates the originator's intent-scoped scope of authority through every invocation in the workflow as a single accruing credential. The five chain operations — Open, Continue, Hold, Authoritative Carry, Close — govern how the chain extends, suspends, and terminates.

What the SCT enables: every enforcement point evaluates the originator's actual authority for the declared intent against the specific invocation. Authority does not drift across hops because it does not re-issue at hops. Confused deputy and delegation escalation, the dominant authority-related failure modes in multi-hop agentic workflows, are structurally prevented.

What the SCT does not carry: data payloads, observation data, audit detail. These live in OpenTelemetry spans and the Telemetry Record. The separation keeps the SCT propagatable across deep compositions while preserving full audit detail through the observability layer.

Signed Manifests with Process Semantics

SADAR manifests declare not only what a component is, but what it does, what it does not do, and what it expects to have been completed before it runs. The process semantics — performs, does_not_perform, expects_completed — give enforcement points the inputs they need to evaluate fit-to-purpose and sequence.

The publisher-signed manifest also anchors trust to a real-world accountable party. Publisher accountability is verifiable, auditable, and supports incident response and compliance in ways that self-issued identity models cannot.

Bilateral Trust Topology

The architectural choice to put the registry outside the runtime trust path — letting requester and service authenticate directly — enables a cascade of operational properties. No central coordinator must be online for authorization decisions. Each service owns its trust relationship with its requesters. Revocation is unilateral. Federation is bilateral, not consortium-governed. Sovereign deployments operate as first-class participants without compromise.

This contribution is structural, not novel in primitives. Bilateral trust uses standard mTLS and OAuth. The innovation is the architectural commitment to keep the registry out of the runtime trust path — and the design discipline to maintain that property across all the layers.

Two-axis Lifecycle Separation

Manifest cache validity (consumer-side) and credential validity (serving-organization side) are independent. Each authority operates on its own timeline. Deauthorization at the registry and credential revocation at the serving organization are decoupled acts. This separation is what makes federation practical: each party in the federation acts under its own logic without requiring lockstep coordination with others.

Federation as Primary Deployment

Cross-organization operation is the case to solve, not the case to accommodate. The architecture is designed around federation from the start: bilateral trust between registries, service-sovereign credential issuance, decoupled lifecycles, repatriated observability. Single-organization deployments are special cases of federation — the same substrate, with the federation set reduced to one party.

OpenTelemetry-based Observability with Repatriation

Observability is a workflow property, not a component property. Spans tagged with workflow context flow to the originator's collector through the Repatriation protocol, producing a unified trace owned by the originator as the workflow executes. Producer-side privacy controls govern what crosses trust boundaries. Cross-organization correlation is automatic; forensic reconstruction is a query against the originator's collection, not an investigation across separately-owned log stores.

The Telemetry Record adds per-invocation persistent audit with phase-immutability across Search, Selection, Invocation, and Outcome phases. The combination of repatriated traces and phase-immutable Telemetry Records produces non-repudiable evidence for compliance, incident response, and dispute resolution.

Three Decoupled Normative Concepts

Conformance, certification, and authorization are three independent normative concepts in SADAR. Each is evaluated separately and revoked separately. Each is governed by the appropriate authority. The decoupling enables nuanced governance: a component may be conformant without being certified, certified without being authorized, authorized despite limited certification. R-CONF, R-CERT, R-AUTH, and R-ELIG normative requirement families specify the precise interactions.

Operating Standards as the Technology Stack

SADAR is built on standards enterprise IAM already operates: OpenID Connect, OAuth 2, DPoP, JWS, JWE, mTLS, OpenTelemetry. The SADAR-specific artifacts — manifests, SCTs, Telemetry Records — are structured forms of standard artifacts. No new cryptographic primitives, no new protocols, no new infrastructure requirements beyond what current enterprise IAM already supports.

This is a deliberate contribution. Many proposed agentic IAM frameworks rely on Decentralized Ledger Technology, Decentralized Identifiers, Verifiable Credentials, or Zero-Knowledge Proofs — each of which has substantial operational overhead and ecosystem adoption requirements. SADAR uses these primitives optionally where they add value, but the substrate operates fully on standards already in production at every large enterprise.

Measurable Outcomes

To evaluate the substrate's effectiveness in deployment, the following metrics are useful.

Metric	What it measures
Authorization decision time	Time from invocation receipt at the service to allow/deny decision. Lower latency indicates effective input availability at the enforcement point.

Metric	What it measures
Cross-organization workflow success rate	Percentage of workflows traversing multiple organizations that complete without authorization failures. High success rates indicate substrate properties operating correctly across boundaries.
Confused deputy incident rate	Number of detected confused-deputy events per million invocations. Should approach zero in a properly deployed SADAR substrate; positive numbers indicate policy or implementation issues.
Mean time to revocation propagation	Time from a service's decision to revoke credentials to the requester's next attempted invocation being denied. Lower times indicate effective service-sovereign revocation.
Cross-organization audit reconstruction time	Time from incident identification to assembled forensic view. With repatriated traces, this should be query latency — not investigation duration.
Federation establishment time	Time from initial agreement between two registries to first successful cross-registry discovery. Bilateral establishment should be hours to days, not months.
Producer privacy compliance rate	Percentage of repatriated spans that pass producer-side privacy checks. High rates indicate effective producer-side controls.

9. Discussion and Future Work

SADAR addresses what the authors believe is the central architectural problem in agentic AI: making authorization decidable at every invocation under runtime composition, across organizational boundaries, with prevention rather than detection as the principal defense against authority confusion. The framework as specified is sufficient for production deployment of straightforward agentic systems and for the cross-organization federations that increasingly characterize enterprise agentic operations. Several areas warrant continued development.

Performance and Scalability

The SCT grows with workflow depth, accruing attestations under the Continue chain operation. While the size remains bounded — each attestation is a small signed claim, and the Authoritative Carry operation compresses long chains — production deployments will benefit from performance benchmarking across realistic workflow patterns. Future work should establish quantitative bounds for SCT size, signature verification time at enforcement points, and propagation overhead across hops.

Caching strategies for manifests and JWKS endpoints, with the soft and hard bound model, mitigate registry and IdP lookup overhead. Optimal cache parameters depend on workflow patterns; production deployments will tune them based on observed behavior.

Process Specification (Part 2)

The process specification was paused in April 2026 with two open questions on resume: (a) whether expects_completed IRI matching should be literal or transitive, and (b) whether does_not_perform validation against performs should occur at manifest ingestion as a validation error on contradiction. Both questions affect how strictly process semantics are enforced and how flexibly composition can occur within those bounds. The Process L1 page documents the manifest model; companion document 15 (Process Flow Specification) handles the enforcement framework integration.

The model clarification is that manifests carry only three IRI lists. SADAR propagates and grounds process semantics; it does not enforce flow, match definitions at discovery, or validate manifests against definitions. Process definitions are consumed by planners as reusable execution-plan templates and by enforcement frameworks for runtime ordering, prerequisites, exclusions, and compensation. The process identifier in baggage gives Guardian-style enforcement engines operational context for Zero Trust policies.

Manifest Schema Extensions

The "one-of" pattern for data field declarations is a future schema extension. It allows a manifest to declare equivalent acceptable forms across standards under an agent-chosen local name (e.g., time:[iso:timestamp:xxx, ansi:datetime:xxx]). The pattern generalizes to non-functional requirements: for discrete-valued NFRs, one-of allows manifests to declare acceptable values on either side; matching becomes set intersection. This cross-cutting manifest pattern reduces the burden of standards alignment without requiring ecosystem-wide schema convergence.

Standardization and Interoperability

SADAR is a candidate for standardization through IETF, OpenID Foundation, or related bodies. Standardization would establish neutral governance, accelerate adoption across vendors, and ensure long-term interoperability. The Community Specification License 1.0 publication at OpenSemantics.org positions SADAR for standards-track contribution; future work includes formal engagement with appropriate standards bodies.

Reference implementations and conformance test suites are essential for ecosystem development. Open-source reference implementations of the registry, the Helper API, the SCT operations, and the Telemetry Record format would accelerate adoption and provide ground truth for conformance verification.

Threat Model Evolution

Adversaries will adapt. As SADAR deployments grow, attackers will probe specific implementation choices, key management practices, and operational gaps. Continued threat modeling — including formal analysis of the SCT chain operations under adversarial composition, the bilateral attestation

pattern under partial-trust scenarios, and the Repatriation protocol under producer-side compromise — will surface refinements to the specification.

Specific areas warranting deeper analysis include quantum-resistant signature schemes (preparing for the post-quantum transition), formal verification of the SCT chain operations' security properties, and adversarial composition attacks where multiple compromised parties collude to forge attestations or exfiltrate authority.

Tooling and Developer Experience

Adoption will be paced by developer experience. Production-quality SDKs for the major implementation languages, IDE integration for manifest authoring and validation, debugger support for SCT chain inspection, and operational tooling for registry management all need to mature. Cognita AI Inc. is developing reference tooling alongside the specification.

Ethical and Societal Considerations

Verifiable component identity and propagated authority enable accountability that has been absent from many AI deployments. They also enable surveillance and control: if every invocation is observable, the same observability that supports compliance audit could be used for invasive monitoring of legitimate behavior. SADAR's producer-side privacy controls and the Repatriation protocol's bilateral applicability address this in part — producers control what crosses their boundary — but the broader question of what observability is appropriate, and who decides, remains an ongoing societal conversation.

The substrate is policy-neutral, but the policies organizations build on top of it have significant implications. Continued engagement with regulators, civil society, and affected communities will shape how SADAR-enabled systems operate in practice.

Closing

SADAR is presented as a proposed authorization substrate for the agentic era. The framework addresses what the authors believe is the actual problem: not better identity, but authorization that operates correctly under runtime composition across organizational boundaries. The design choices — bilateral trust, propagated originator authority, signed manifests with process semantics, federation as primary deployment, observability with repatriation, decoupled normative concepts — follow from this central commitment.

The CSA framework and SADAR are not in opposition. They address overlapping concerns from different starting points, and they can compose: the CSA framework as the IAM layer atop a SADAR substrate. The authors of SADAR view the CSA framework as the most substantive recent work in agentic IAM and welcome continued dialogue across both efforts.

Production deployments, real-world threat data, regulator engagement, and broader ecosystem adoption will continue to inform the specification. SADAR is offered as a contribution to the broader effort of making agentic AI deployable, governable, and trustworthy in production environments. The work is ongoing.

10. References

Cloud Security Alliance. (2025, August). Agentic AI Identity and Access Management: A New Approach. Huang, K., Narajala, V. S., Yeoh, J., et al.

Cognita AI Inc. (2026). SADAR Addresses Governance Gaps Identified in the FINOS AI Governance Framework for Discovery and Authorization. OpenSemantics.org. https://www.opensemantics.org/documents/sadar-addresses-governance-gaps-identified-in-the-finos-ai-governance-framework-for-discovery-and-authorization

Cognita AI Inc. (2026). SADAR Scope (Companion Document 1). OpenSemantics.org.

Cognita AI Inc. (2026). SADAR Governance and Conformance Specification. OpenSemantics.org.

Cognita AI Inc. (2026). SADAR Replication and Manifest Provenance Specification. OpenSemantics.org.

Cognita AI Inc. (2026). SADAR Federation Establishment and Policy Specification. OpenSemantics.org.

Cognita AI Inc. (2026). SADAR Risk Score Specification. OpenSemantics.org.

Cognita AI Inc. (2026). SADAR Features V2.0. OpenSemantics.org.

FINOS. (2025, October). AI Governance Framework v2. Fintech Open Source Foundation.

Hardt, D. (Ed.). (2012). The OAuth 2.0 Authorization Framework (RFC 6749). Internet Engineering Task Force.

Hardt, D., Parecki, A., & Lodderstedt, T. (2025). The OAuth 2.1 Authorization Framework (Internet Draft). IETF.

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force.

Jones, M., & Hildebrand, J. (2015). JSON Web Encryption (JWE) (RFC 7516). Internet Engineering Task Force.

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Signature (JWS) (RFC 7515). Internet Engineering Task Force.

Jones, M., Campbell, B., & Mortimore, C. (2015). JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants (RFC 7523). Internet Engineering Task Force.

Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., & Waite, D. (2023). OAuth 2.0 Demonstrating Proof of Possession (DPoP) (RFC 9449). Internet Engineering Task Force.

Lodderstedt, T., Richer, J., & Campbell, B. (2023). OAuth 2.0 Rich Authorization Requests (RFC 9396). Internet Engineering Task Force.

OpenID Foundation. (2014). OpenID Connect Core 1.0 incorporating errata set 1.

OpenTelemetry Project. OpenTelemetry Specification. Cloud Native Computing Foundation.

W3C. (2020). Trace Context. W3C Recommendation.

Huang, K. (2025). Agentic AI Threat Modeling Framework: MAESTRO. Cloud Security Alliance.

European Parliament and Council. (2023). Regulation on Artificial Intelligence (AI Act).

Cloud Security Alliance. (2024). The State of Non-Human Identity Security.

OWASP. (2025). OWASP Top 10 Non-Human Identities Risks.

Apono. (2026, April). Nine Seconds to Delete a Database: What the PocketOS Incident Teaches Us About AI Agent Privilege Management.

PCMag. (2026). Meta's Director of Alignment Loses Inbox to AI Agent OpenClaw.

Cermi, et al. Why Do Multi-Agent LLM Systems Fail? arXiv:2503.13657v3.