

SADAR™ Semantic Registry Architecture —

Core Security

Version 0.5 — August 2026

© 2026 Cognita AI Inc. All rights reserved. CogniWeave™ and SADAR™ are trademarks of Cognita AI, Inc. All other trademarks are the property of their respective owners.

Contents

Overview	4
1. Pull Request Controls and Charging	4
1.1 Access Control Mechanisms	5
1.2 Charging for Forwarding and Replication.....	6
2. Registry Replication and Forwarding Security.....	6
2.1 Provenance and Integrity	7
2.2 The Registry of Registries as Trust Anchor	11
2.3 Registry Operator Registration.....	14
2.4 Registry-to-Registry Authentication.....	14
2.5 Multi-Hop Forwarding and Identity	15
2.6 Internal Proprietary Registry Exclusion	15
2.7 Security Accountability Summary.....	16
3. Requester Authentication and Authorization	17
3.1 Registration Prerequisite	17
3.2 Query Session Authentication	18
3.3 Authentication TTL and Discovery TTL.....	18
3.4 Request Admission, Expiry, and Re-Authentication.....	19
3.5 Manifest Scope Enforcement	19
3.6 Forwarded Query Scope Propagation.....	20
3.7 The searchAndInvoke Tool	20
3.8 Accountability Table	20
4. Service Invocation Authentication	21
4.1 The Server as Sovereign Token Issuer.....	22
4.2 Manifest Fields Supporting Service Invocation Authentication	22
4.3 The Signed Manifest as Identity Anchor	23
4.4 Identity Granularity and Credential Mapping.....	23
4.5 Agent Credential Isolation	24
4.6 SADAR Context Token (SCT) — Orientation.....	25
4.7 First-Use Provisioning and the Time-of-Use Token	26
4.8 Runtime Authentication Flow.....	27
4.9 DPoP Replay Protection.....	27
4.10 A2A Protocol Interoperability.....	28
4.11 searchAndInvoke Token Management Summary.....	29

4.12 Accountability Table	29
4.13 Authority Establishment and Organizational Boundaries	31
Appendix A: Requirements — Agent Credential Binding and Invocation Authentication	32
A.1 Agent Credential Binding	32
A.2 Invocation Service Authentication	33
A.3 Credential Lifecycle	33
A.4 Non-Normative Implementation Guidance	33

Overview

This document specifies the security architecture for the SADAR™ Semantic Registry. It covers distinct authentication and authorization contexts:

- **Registry access controls** — how registry operators constrain replication and forwarding access between registries (Section 1)
- **Registry-to-registry security** — how registries authenticate to one another for replication and forwarding, using the Registry of Registries as trust anchor (Section 2)
- **Requester authentication** — how agents and other entities authenticate to the local registry to issue discovery queries (Section 3)
- **Service invocation authentication** — how a requesting agent establishes trust with a selected service agent at runtime, after discovery is complete (Section 4)

Sections 1–3 govern interactions in which the registry is an active participant. Section 4 governs agent-to-agent interactions at invocation time, in which the registry is explicitly not a participant — its role ends at discovery. Invocation flows that cross an organizational boundary are additionally treated in the **Cross-Organizational Flows** companion.

1. Pull Request Controls and Charging

The registry containing the canonical definition of an entry is defined as the **Home** registry. This is the registry in which the original publisher make the entry available. As the Home registry, it has full authority to constrain replication access. Just as a service provider may enforce usage limits and access controls on consuming entities, a registry operator may enforce equivalent controls on registries requesting to pull its content. These controls are an operational and commercial accountability of the providing registry administrator, not a protocol-level negotiation.

The requesting registry is defined as the **Host** registry – i.e. it is hosting the entry's definition. When an entry is hosted, its manifest as well as that of its parent entity are replicated to the Host registry. A host **MUST NOT** make a hosted entry available for further federation.

Note: By convention, if a registry administrator directly defines an entry for a resource that it does not own, it creates an entity entry representing the publisher making the resource become a hosted entry. An example where this may occur is if the agentic flows need access to a third party resource that is not hosted in a federated registry.

1.1 Access Control Mechanisms

The following mechanisms may be applied independently or in combination. All mechanisms are enforced by the providing registry before any content is returned.

Control Mechanism	Description	Operational Notes
Rate Limits	Cap the number of pull requests per time window (e.g. requests/minute, requests/day)	Prevents a single subscribing registry from saturating the providing registry's bandwidth. May be differentiated by subscriber tier or commercial agreement.
Payload Size Restrictions	Cap the maximum response payload size per pull request (record count, byte size, or both)	Protects providing registry resources and forces subscribing registries to paginate large pulls. Prevents bulk data exfiltration in a single call.
Allow Lists	Explicit enumeration of registry URNs permitted to request replication from this registry	Positive-control model: only registries explicitly authorized may pull. Required for internal proprietary or commercially restricted registries. Evaluated before any other control.
Block Lists	Explicit enumeration of registry URNs denied replication access, regardless of other criteria	Negative-control model: specific registries are excluded even if they would otherwise satisfy allow-list or authentication criteria. Used to enforce commercial exclusions, competitor restrictions, or revoke previously granted access.
Authentication Requirement	Replication requests must present a valid registry identity token	Ensures only legitimately registered registries may pull. Unauthenticated pull requests are rejected regardless of allow/block list configuration.
Scope Restriction	Limits replication to a declared subset of the registry's content (e.g. specific NAICS branches, providers, manifest versions)	A registry may publish its full catalog for forwarding-mode queries while allowing replication of only a defined subset. Commercial tiers may

		expose different replication scopes at different price points.
--	--	--

Allow lists and block lists are evaluated first, before authentication. The presence of an allow list takes precedence over the block list which is not evaluated in this scenario. A registry on the block list is rejected regardless of whether it holds a valid identity token. A registry not on the allow list (when an allow list is configured) is rejected regardless of authentication status. This ordering prevents authentication infrastructure from becoming a bypass vector for access control.

1.2 Charging for Forwarding and Replication

Registry operators may charge rates to calling registries for forwarding and replication services. The charging model follows exactly the same settlement patterns defined in the [Payment Models](#) — the same models, the same settlement providers, the same mechanics, applied at the registry-to-registry level rather than the entity-to-service level.

- **Advertising:** The providing registry's manifest (held in the Registry of Registries) declares the supported settlement models, pricing tiers, and settlement provider for replication and forwarding access, using the same paymentSupport structure used by service manifests.
- **Negotiation:** At connection establishment between two registries, the providing registry returns settlement configuration — billing model, cost structure, settlement recipient identifier — in the same structure as a service connection establishment response.
- **Settlement:** Financial settlement proceeds through the designated settlement provider (Stripe, PayPal, X.402 network, etc.) without the providing registry or the standards body ever handling payment instruments directly.
- **Enforcement:** Access to replication or forwarding is governed by the requesting registry's account standing with the providing registry, not by per-request payment authorization. This is consistent with the SADAR philosophy of avoiding runtime payment dependencies in the execution path.

The providing registry may configure different commercial tiers that expose different replication scopes, rate limits, or payload allowances. The manifest advertises available tiers; the requesting registry selects a tier at connection establishment.

2. Registry Replication and Forwarding Security

The security model for registry-to-registry interactions is a direct extension of the entity-to-service security model. The same principles apply: signed manifests, OIDC-based identity, JWE-encrypted usage keys, mutual TLS, and registry isolation. What changes is the level of abstraction: the Registry of Registries (RoR) plays the same trust-anchor role for registries that a local registry plays for entities.

The full treatment of replication protocol mechanics, forwarding controls, charging, and multi-hop identity is documented in SADAR™ Registry Replication Security. This section describes the security architecture; that document describes the operational protocol.

2.1 Provenance and Integrity

2.1.1 Entity Definition

A founding principle of SADAR is that discovered agentic resources – agents, tools, resources, and the like – must have a verifiable provenance that defines their ownership. The verification requires that this be programmatically testable versus attested to and trusted. Furthermore, the integrity of the definition of each resource must be tamper evident. This is enabled through the related entity manifests in the registry.

A party must enroll with the registry to gain access. That process has some verification of the party's identity and the asserted entity manifest they provide. While there are steps to ensure that the entity, and the individual representing it, are in fact representing the company, those steps are not verifiable by the eventual user of a resource. What the user can verify is signatures.

Each entity manifest is signed first using that entity's own key and then countersigned using its parent's key. The exception is the root entity, whose manifest is signed once, using the key published at the organization's own domain — its Fully Qualified Domain Name (FQDN).

Note: For the purposes of this document, **countersign** means the signing of the bytes of the inner content plus the inner signature itself:

```
[ (manifest bytes) (entity key signature) ] (parent signature)
```

This two-level signing validates the root against the FQDN and then every sub-entity against its parent, creating trust for sub-entity keys as each is verifiable back to the FQDN via the hierarchy. A forger claiming to be the organization cannot produce the root signature, because they do not control that domain — short of compromising the domain itself.

Resource manifests follow the same pattern one level down. Each resource's manifest carries the resource's own signature and the countersignature of its owning entity. Ownership is therefore a grant, not a claim. A manifest naming an owner whose countersignature is absent or invalid is rejected.

As such, impersonating an entity requires compromising the organization's domain or its signing keys — the same assets that already protect its website and its email. SADAR introduces no new crown jewel to defend. It anchors trust via the same key the organization already uses for trust.

2.1.2 Entity Structure

Entities can define sub-entities reflecting whatever structure is desired. Common structures might be the parent root entity with legal entities beneath, entities by legal jurisdiction, divisional, and so on. Sub-entities must point to one, and only one, parent entity, with the root having no parent.

```
Parent org entity      - signed by the FQDN key
├─ Sub entity 1        - key k1, countersigned by FQDN key
├─ Sub entity 2        - key k2, countersigned by FQDN key
│   └─ Sub entity 2.1  - key k2.1, countersigned by k2
│       └─ Sub entity 2.1.1 - key k2.1.1, countersigned by k2.1
```

Countersigning always follows the immediate parent edge; a parent countersigns its direct children only — never a grandchild. Only the root key is bound to the FQDN. Three consequences make this the workable shape at scale.

- Divestiture is one cut: Revoking a single parent-child edge severs trust in the entire subtree beneath it.
- The root key is an existing trust mechanism: the FQDN key is a foundation trust principle used throughout the organization already well controlled and maintains.
- Each signer attests only what it actually knows: a division knows its own children; the corporate root need not learn of a team-level reorganization three layers down.

Mechanically, each manifest is a JWS (self-signed, its key material identified within); the countersignature is a detached JWS whose payload is the child's complete serialization. Verification is outside-in — the parent's signature over the artifact first, the child's signature within it second — and tampering with either the content or the child's signature breaks the parent's signature.

2.1.3 Keys and Rotation

Every signature names its own verification key. The signing key's identifier (kid) is carried in the signature's protected header, inside the signed bytes. An artifact therefore permanently declares which key verifies it, and that declaration cannot be altered without breaking the signature. Verification always resolves the named key.

A manifest publishes its key material in one of two ways: **inline** (the key set embedded in the manifest body) or **by reference** (a `jwtks_uri` pointing to a key endpoint the entity controls). A verifier reads whichever field is present acting accordingly. The mode is frozen into the signed manifest, so replication between registries carries it unchanged.

Rotation differs by mode:

By reference, the entity updates its key endpoint. Publishing the new key and retaining the old one, marked superseded rotates the key. As the key value itself is accessed via reference, the manifest remains unchanged and does not require resigning.

Inline, replacing a key changes the manifest body which produces a new manifest version. This requires re-signature of the manifest with a fresh parent countersignature.

By reference is the default posture because a new version should mark a change in declared functional behavior or NFR characteristics versus a routine mechanical key event.

In both modes the key history is append-only. Rotated-out keys are retained, marked, and never deleted, so historical artifacts remain verifiable against the exact key that signed them.. Public keys are immutable for a given pinned key (kid) and may be cached indefinitely. Registries and chain bundles (see 2.1.4) may carry key sets alongside manifests so verification remains possible even when a key endpoint is unreachable.

The two modes trade endorsement granularity for operational cost. Inline, the parent's countersignature endorses each key individually. By reference, it endorses the key channel, the endpoint the child controls.

Both are conformant and MUST be supported. Since the mode is set by each registry entry, it is possible to mix the modes allowing higher-custody nodes to use inline as the stricter posture where required.

Key compromise is a different scenario from normal key rotation. A compromised key is handled by the parent revoking the certification edge through its own signed lifecycle record, severing trust in the subtree beneath it (see the Entity Trust Anchoring companion).

2.1.4 Risk-Based Verification

The validation level a registry consumer elects to invoke may vary to meet its risk posture and appetite. This makes verification depth a posture versus a single switch. Each level reduces residual risk at the expense of processing. As this processing intentionally lives within the consumer itself, the level of verification is opaque to SADAR and other registry participants.

Level	Verifier checks	What is verified
0 — Registry deference	Nothing; entries are consumed as the registry serves them	Nothing; trusts the registry's ingestion validation and its ongoing integrity
1 — Entry integrity	The entry manifest's own signature, against the key its signature names	That the entry's manifest is unchanged; ownership is unverified
2 — Publisher endorsement	Level 1 plus the owning entity's countersignature over the complete signed artifact	That the owning entity endorsed this exact manifest — association and content — in addition to Level 1's integrity; the endorsing entity's own key is unverified, so a forged parallel hierarchy still passes
3 — Full tree	The whole chain: root at the FQDN, every countersignature and lifecycle record down to the entry	Verifies every entity starting with the root, thus proving the chain belongs to the organization whose FQDN was used and is unchanged; all that remains trusted is freshness (the staleness window) and the anchor infrastructure itself (DNS / web PKI)

Direct verification (Levels 1–2):

1. Resolve the key named by the manifest signature's kid — from the inline key set or the referenced key endpoint — and verify the manifest's signature with that public key. Success proves the manifest is unchanged since signing.
2. For Level 2, additionally verify the owning entity's countersignature over the complete signed artifact — content plus the entry's signature.

Full entity chain verification (Level 3) — performed root-downward, because each verified step is what makes the next step's key trustworthy:

1. **Root:** fetch the key published at the organization's FQDN via its declared mechanism — a well-known HTTPS path, a certificate binding, or a DNS record (see the Entity Trust Anchoring companion). This is the one step that goes around the registry. Verify the root manifest's signature with it — this simultaneously proves the root manifest is unchanged and that it was issued by the domain holder. The root's key is the FQDN key.
2. **Child:** verify the parent key's countersignature over the child's complete signed artifact — content plus the child's own signature. That one check proves the artifact is unchanged and that the parent endorsed this exact self-signed declaration. Then verify the child's own signature within, against the key its manifest declares — proving the child actually held the key it claims. Only now treat the child's key as trusted to endorse the next level.

3. Repeat down each edge to the target entity. Each verified step promotes the next step's key from data to trust anchor. At each edge, check the current lifecycle record — signed by its administering authority — for revocation, within a freshness window suited to your risk tolerance.

4. Fail closed at the first broken step: nothing beneath an unverified edge is evaluated. Verified steps may be cached — verifying a second entity under the same parent starts at the already-trusted key and costs one step.

Why there is no intermediate level. The tempting middle rung — verify the entry and also check the root at the FQDN, without walking the interior — is deliberately absent. It adds work for the verifier without adding burden for the attacker: the genuine root manifest is a public artifact, so a fabricated interior entity presented alongside the real, borrowed root passes both end checks while never having been granted anything. Checking the root proves the organization exists and yields its authentic key; it does nothing to verify the chain. Likewise, walking parent references to find the root is resolution, not verification — pointers are claims anyone can write; only countersignatures make edges true. An assurance step whose checks an attacker passes for free is a decoy, not a level. Two notes keep full-tree verification affordable: in the very common case where the owning entity is the root, the full tree is exactly two checks; and the /chain bundle below makes any depth a single fetch, with verified steps cacheable.

To make full verification practical, a SADAR registry **SHOULD** provide a /chain endpoint that returns, for a given entity, every manifest in its hierarchy with countersignatures and current lifecycle records:

```
{
  "bundle_for": "urn:sadar:entity:acme:div2:team1:proj-x:v3",
  "assembled_at": 1786550000,
  "path": [
    { "manifest": "<root entity manifest – complete signed artifact>",
      "anchor": { "fqdn": "acme.com", "mechanism": "well_known",
"location": "/.well-known/sadar-entity-key" },
      "lifecycle": "<signed lifecycle record or absent = active>" },
    { "manifest": "<Sub entity 2 – complete signed artifact>",
      "countersignature": "<detached JWS by the root/FQDN key over
those octets>",
      "lifecycle": "..." },
    { "manifest": "<Sub entity 2.1 – signed artifact>",
      "countersignature": "<detached JWS by k2>",
      "lifecycle": "..." },
    { "manifest": "<Sub entity 2.1.1 – signed artifact>",
      "countersignature": "<detached JWS by k2.1>",
      "lifecycle": "..." }
  ]
}
```

Nothing in the bundle is asserted by the registry. Every element is signed by the party that could truthfully sign it — the child, the parent, the lifecycle authority, the domain — and the verifier checks those signatures itself, terminating at an anchor the registry does not control. The registry is a courier of evidence, never a witness: it cannot fabricate a bundle that verifies, and it could be replaced by an untrusted mirror without weakening verification. Its only real power is staleness — serving yesterday's bundle without today's revocation — which is why the bundle carries `assembled_at`, lifecycle records are sequenced, and a consumer applies the freshness window its risk tolerance demands.

2.2 The Registry of Registries as Trust Anchor

The standards-body-operated Registry of Registries is not merely a discovery directory for registry descriptors. It is the trust anchor for the federation model. For registries to be available for *public* federation, they must be both **certified** and **authorized**.

Certification is an independent assessment of conformance to the SADAR standards.

Authorization is an independent assessment of certain TBD operational and security commitments.

To be Authorized, a registry must first be Certified. In this way, the RoR ensures that it only advertises fully compliant and reliable registries for discovery and federation. The canonical RoR is the Directory of Authorized Registries, maintained by the standards body: a registry's current listing is the federation eligibility credential, and conformant registries reject forwarding and replication requests from registries not listed. Non-authorized registries may consume from authorized registries; they may never serve content into the public federation.

The RoR holds registry descriptors only — signed by their publishers and countersigned under their owning entities like any other manifest (§2.1). This ensures that the content of the RoR is fully verifiable at runtime, if desired — at whatever assurance level the consumer elects (§2.1.4) — versus merely trusted to be legitimate and unaltered.

Replicated content preserves this property end to end through three signature roles: the publisher's signature never changes wherever a manifest travels; the home registry issues a separate signed binding attesting "this manifest was published through me"; Host registries serve bytes faithfully and sign nothing. Federation is bilateral and non-transitive — each link is its own evaluated, consented relationship — which contains the blast radius of any compromised registry.

Registries authenticate to one another with the same manifest-anchored first-use provisioning and time-of-use tokens as any two SADAR parties (§2.4, §4.7). The RoR issues no runtime credentials; its contribution is the eligibility check. Revocations and lifecycle changes propagate as signed messages over the mandatory, mutually authenticated push channel.

Federation establishment workflow, eligibility modes, replication mechanics, and policy administration are covered in the SADAR Federation Establishment and Policy and Replication and Manifest Provenance documents. Enforcement responsibilities split across three layers:

Layer	Decides	Enforced at
Federation membership	Which home registries flow content into and out of this registry; the bilateral agreements in force	Registry admin console; bilateral federation assertions exchanged through push
Per-entity / per-entry ACLs	Which entries participate in which federations, in either direction	Registry serving layer; admin-configured ACLs evaluated at discovery and replication time
Requester-specific authentication	Which specific principals can authenticate to specific endpoints	IAM at the authentication call; failure returned before the request reaches the registry

This symmetry is intentional. The same tooling, the same cryptographic primitives, and the same operational patterns that govern entity-to-service interactions govern registry-to-registry interactions. The only difference is the scope at which they operate.

Dimension	Entity ↔ Registry (existing model)	Registry ↔ Registry (extended model)
Trust anchor	The Semantic Registry acts as trust anchor for entities (agents, services) within its scope. It assigns entity IDs, holds signed manifests, and is the authoritative source for entity discovery.	The Registry of Registries acts as trust anchor for registry operators across the ecosystem. It assigns registry IDs, holds signed registry manifests, and is the authoritative source for registry discovery.
Identity issuance	An entity registers with its local registry and receives a registry-issued entity URN.	A registry operator registers with the standards body RoR and receives a RoR-issued registry URN.
Manifest signing	Entities sign their own service manifests (JWS/ES256) prior to upload. The registry stores manifest hash + signature; it never holds the entity's private key.	Registry operators sign their own registry manifests (JWS/ES256) prior to upload to the RoR. The RoR stores manifest hash + signature; it never holds the registry's private key.
Token assertion at runtime	At service invocation, the client entity asserts a signed identity token to the provider service. The provider validates the token against the registry's JWKS.	At replication or forwarding time, the requesting registry authenticates directly to the providing registry under the universal model — manifest-anchored proof of key possession validated against the requesting registry's descriptor-published JWKS — with current Directory listing checked as eligibility.
License and payment	Service manifests advertise supported settlement models and licensing terms. Usage keys and settlement metadata are exchanged at connection establishment per the Payment Architecture Specification.	Registry manifests advertise supported replication models and licensing terms for access. License keys and settlement metadata for replication fees are exchanged at pull/forwarding establishment using identical mechanisms.
Right of refusal	A provider service may refuse a connection from a client entity whose manifest characteristics are incompatible (compliance posture, governance framework, entity type).	A providing registry may refuse a replication or forwarding request from a requesting registry whose registry manifest characteristics are incompatible, or which does

		not appear on the providing registry's allow list.
--	--	--

The RoR's blast radius is bounded identically to a local registry's blast radius: it is limited to identity issuance and descriptor storage, and it issues no runtime credentials. It is not a runtime participant in replication or forwarding data flows. Once two registries have authenticated to one another, all subsequent replication and forwarding interactions occur directly between them — the RoR is not in the data path.

2.3 Registry Operator Registration

Before a registry instance may participate in inter-registry replication or forwarding, its operator must register with the standards body through the RoR. This process mirrors entity registration with a local registry:

- **Operator authentication:** The registry operator authenticates to the RoR using the organization's credentials (mTLS + OIDC Client Credentials flow).
- **Registry ID assignment:** The RoR assigns a globally unique registry URN to the registry instance (e.g. urn:sadar:registry:acme-corp:primary).
- **Registry manifest publication:** The operator prepares a registry manifest describing the registry instance's scope, NFRs, supported replication models, licensing terms, and settlement configuration. The manifest is signed locally by the operator's private key (JWS/ES256) and uploaded to the RoR. The RoR stores the manifest hash and signature; it never holds the operator's private key.
- **OIDC endpoint declaration:** The operator's registry OIDC endpoint URI is embedded in the signed registry manifest, meaning the endpoint address is covered by the operator's signature and cannot be redirected undetected.

2.4 Registry-to-Registry Authentication

Registry-to-registry authentication uses the universal SADAR model — the same manifest-anchored first-use provisioning, sovereign token issuance, and per-request proof of possession defined in §4 — applied at registry scope. There is no special-purpose mechanism and no central token authority: the RoR contributes an eligibility check (current Directory listing), never credentials.

2.4.1 Authentication Flow

- The requesting registry resolves the providing registry's descriptor — from the Directory or its federation configuration — and verifies it at its elected assurance level (§2.1.4), confirming the provider's current Directory authorization.
- The providing registry likewise confirms the requester's current Directory listing and evaluates its allow and block lists before any exchange proceeds.

- First-use provisioning proceeds per §4.7: the requesting registry presents a signed assertion (RFC 7523) produced with its manifest-anchored key; the provider resolves the requester's `oidc_issuer` and JWKS from the requester's signed descriptor and confirms proof of key possession — the presenter controls the key that signed the descriptor.
- The providing registry, as sovereign issuer, issues a time-of-use token bound per §4.7 — requesting registry identity, intent, and intent instance — with dual TTLs, the most stringent triggering refresh.
- Each subsequent request carries the token plus a DPOP proof (§4.9), over mTLS. Push-channel messages are signature-verifiable under the same model.

2.4.2 Relationship to the Directory

The token itself has no registry-specific claim profile: it is the §4.7–§4.9 model applied with the registry as the actor. What is registry-specific is the eligibility layer: Certification (conformance) and Authorization (listing) are institutional acts of the standards body, checked at establishment and re-checked per the parties' policies. Conformant registries reject forwarding and replication requests from registries not currently listed; between listed registries, requests may still be declined under the receiving registry's declared NFRs, allow and block lists, and commercial terms (§1).

2.5 Multi-Hop Forwarding and Identity

When a query propagates across multiple forwarding hops, the identity and trust model must remain coherent across the chain. Two principles govern this:

- **Each hop authenticates independently:** Each registry in the forwarding chain authenticates to the next-hop registry under its own credentials per §2.4. A forwarded request does not carry the originating registry's token through the entire chain — each hop is a distinct authenticated registry-to-registry interaction. This prevents a compromised intermediate registry from forwarding with a token it did not earn.
- **Originating entity scope propagates with the query:** The scope asserted by the originating entity's manifest is included in the forwarding request payload (not in the registry-to-registry authentication token). Intermediate and terminal registries may use this scope to enforce their own access policies for the entity's query, independent of the inter-registry authentication.

A providing registry receiving a forwarded query validates two independent identity assertions: (1) the forwarding registry's authentication per §2.4, establishing that it is a legitimately registered, currently authorized registry; and (2) the originating entity's manifest scope claim, establishing that the query was initiated by an entity with an active, in-scope manifest. Either assertion may independently result in the query being refused.

2.6 Internal Proprietary Registry Exclusion

An internal proprietary registry does not register with the RoR and does not participate in the registry-to-registry authentication protocol. It has no registry URN in the RoR namespace, no RoR-held descriptor,

and no Directory listing for other registries to validate. It cannot be a replication source or a forwarding target for any registry outside the enterprise boundary.

This exclusion is categorical and enforced at the network and protocol boundary, not merely by configuration. An internal proprietary registry that is unreachable from outside the enterprise network satisfies this requirement architecturally. Additional enforcement through the registry's access control layer is defense-in-depth.

2.7 Security Accountability Summary

Concern	Owner	Mechanism / Notes
Registry private key	Registry operator	Never leaves the registry operator's environment. The RoR never holds registry private keys.
Registry manifest integrity	Registry operator + RoR	JWS signature + hash storage in RoR. Any post-upload tampering is detectable by any validating party.
Registry-to-registry authentication	RoR OIDC endpoint	Universal model (§2.4): manifest-anchored first-use provisioning; the providing registry is the sovereign issuer of the time-of-use token.
Token validation (R2R)	Providing registry	Requester's JWKS resolved from its signed descriptor; proof of key possession; DPoP per request; current Directory listing checked as eligibility. The RoR is not involved at runtime.
Time-of-use token issuance (R2R)	Providing registry	Issued by the providing registry at establishment, bound to (registry identity, intent, intent instance) with dual TTLs per §4.7.
Replication license enforcement	Providing registry	Providing registry validates the time-of-use token and DPoP proof on each pull request; rejects expired or invalid presentations.

Allow/block list enforcement	Providing registry	Applied before any authentication exchange proceeds. A blocked registry cannot proceed even with valid credentials.
Rate and payload limit enforcement	Providing registry	Applied per authenticated requesting registry URN. Limits may vary by commercial tier.
Requester entity registration	Local registry	Entity must be registered and hold an active manifest before issuing queries.
Requester manifest scope enforcement	Local registry	Registry validates query criteria against entity's manifest scope at every query.
RoR blast radius	RoR architecture	Limited to registry discovery and identity issuance. RoR is not a runtime participant in replication or forwarding data flows.

3. Requester Authentication and Authorization

Every entity that issues a query to a registry — whether a direct consumer agent, an orchestrator, or a pipeline component — must be a registered entity of that registry with an active, valid manifest. Anonymous or unauthenticated query access is not permitted.

This requirement serves two governance functions. First, it creates an auditable record of who queried the registry and under what manifest terms, supporting compliance reporting and forensic investigation. This serves as the basis for the bilateral trust that each party can validate and attribute the other.

Second, it enables the registry to enforce manifest-scope constraints on query criteria — an entity's registered manifest defines the scope of what it is permitted to search for, not merely what it is permitted to invoke.

3.1 Registration Prerequisite

Before an entity may issue queries:

- The entity must be registered in the local registry and assigned an entity URN.

- The entity must have an active, signed manifest on file with the registry. The manifest declares the entity's identity, its operational scope, its compliance posture, and any governance constraints it operates under.
- The entity's manifest must be in the Active lifecycle state. An entity whose manifest is deprecated, past its deprecated_date, or superseded may not issue queries until it publishes and activates a replacement manifest.

3.2 Query Session Authentication

The authentication flow for a querying entity mirrors the client-side flow in the entity-to-service security model:

- The entity authenticates to the local registry's OIDC endpoint using its private key (Client Credentials flow, mTLS).
- The registry validates the entity's credentials and confirms its registration status and manifest currency.
- The registry issues a signed query session token scoped to the entity's manifest-declared query permissions. This token carries the Entity URN, Agent ID, and granted operation scope as standard claims, along with a jti (JWT ID) and iat/exp timestamps. The jti + exp combination prevents replay: each token is unique, short-lived, and targeted to this registry instance via the aud claim.
- The entity presents this token with each query request.
- The registry validates the token and enforces scope constraints against the query criteria before processing.

3.3 Authentication TTL and Discovery TTL

There are two independent TTL concepts in discovery that must not be conflated.

The **Authentication TTL** defines the period for which the requesting agent's identity has been authenticated. Once expired, it must reauthenticate to access the registry. Because Client Credentialing is a non-interactive, machine-to-machine flow, token refresh is a silent, automatic operation: the agent detects expiry (proactively by inspecting exp, or reactively on a 401 response), re-presents its private key credentials to the registry's OIDC endpoint over mTLS, and receives a new signed JWT with a fresh jti, iat, and exp. Re-authentication does not restart the Discovery TTL.

The **Discovery TTL** is the period for which the discovery results are valid. Both the requester and the server may define a Discovery TTL. Both are mechanically enforced with the most stringent triggering first. The rationale for allowing both to define the Discovery TTL is for each to bound the discovery entry based on the specifics of their use cases. A server may be satisfied with a relatively long Discovery TTL knowing that they have a stable service whereas a sensitive consumer may want to proactively validate the services being used or may want to set a Discovery TTL to force the agentic flow to look for alternatives, potentially with more favorable terms.

The server registry can protect itself from excessive discovery refreshes by a specific requester by establishing rate limits and other such controls.

3.4 Request Admission, Expiry, and Re-Authentication

Two clocks govern a query session, and they answer different questions. The Authentication TTL (§3.3) governs the session between requests: when it expires, the agent silently re-authenticates — a non-interactive Client Credentials refresh, proactive when exp is within the configured threshold, reactive on a 401 — and continues. Admission-time validation governs each individual request: the registry validates the token at the moment it accepts the request, and that admission decision is final for that request.

The governing principle for an admitted request is commit-or-reject at admission, not mid-execution. If the token is valid at admission, the request executes to completion even if the token expires during processing; the registry does not re-validate mid-execution, avoiding the race in which a long-running forwarding chain spans a token boundary and produces a partial result with no defined compensating transaction. Expiry therefore never aborts work in flight — it surfaces on the next request, which triggers re-authentication per §3.3.

- Token valid at admission → the request completes, including any forwarding sub-queries, under the authentication that was valid when the request entered.
- Token expired before admission → the request is rejected. The registry returns a 401 before any processing begins; the agent re-authenticates and resubmits.
- Token expires during a forwarding chain → intermediate registries continue processing; the originating registry assembles and returns the complete aggregated result; the agent's next request triggers re-authentication.

Note: admission finality is an authentication rule only. It does not preclude an enforcement layer from acting mid-flow on other grounds — declining continuation, applying policy, or terminating a flow — nor does it extend any authorization beyond what the admitted request carried.

Implementers note: a very short Authentication TTL combined with a high-latency forwarding chain could result in the agent needing to re-authenticate before it can act on a result still being assembled. The searchAndInvoke reference implementation handles this by proactively refreshing the authentication token when exp is within a configurable threshold — defaulting to 30 seconds — so that the token presented on the next request is already fresh before the prior result is returned.

NEW:

Both the requester and server manifests contain the location of the authentication endpoints. When the requester has selected the server to invoke, it must authenticate to the endpoint in the server's manifest.

3.5 Manifest Scope Enforcement

The registry enforces that the entity's query criteria fall within the scope declared in its active manifest. An entity registered with a manifest scoped to NAICS 62 (healthcare) and APQC PCF node 8 may not issue queries targeting NAICS 52 (finance) or PCF node 3 — these are outside its declared scope.

Scope violations are returned as explicit errors, not silently narrowed results. The entity's query is rejected with a scope-violation response identifying which criteria exceeded the entity's manifest scope. Silently narrowing a query would produce a result set the consumer could mistake for complete.

3.6 Forwarded Query Scope Propagation

When the local registry forwards a query outward, it includes the originating entity's manifest scope in the forwarding request payload. This enables intermediate and terminal registries to enforce their own access policies with respect to the originating entity, independent of the inter-registry authentication.

A providing registry receiving a forwarded query may accept the query as presented, narrow the query to the intersection of the forwarded scope and its own allow policy, or reject the query if the originating entity's manifest characteristics are incompatible with its governance requirements.

3.7 The searchAndInvoke Tool

In practice, the searchAndInvoke tool is provisioned to the requesting agent through the agent's native tooling mechanism — MCP, A2A, or a framework-specific means. This tool encapsulates the full discovery lifecycle: authentication, token refresh, request construction, result caching, TTL tracking, and selection acknowledgment. The calling agent interacts with a single search interface without managing any of these mechanics directly.

A reference implementation of this tool with MCP bindings is published as part of the SADAR open-source reference implementation.

3.8 Accountability Table

Control	Owner	Description
Requester entity registration	Requesting entity (agent/orchestrator)	Entity must be registered in the local registry and assigned an entity URN before it may issue queries.
Requester manifest validity	Requesting entity + registry	The entity's manifest must be active (not deprecated, not superseded) at the time of the query.
Query token issuance	Local registry	The registry issues a signed query session token to the requesting entity on authentication. This token scopes the entity's permitted query operations.

Manifest scope enforcement	Local registry	The registry enforces that query criteria fall within the entity's registered manifest scope. Queries outside manifest scope are rejected with a scope-violation error, not silently narrowed.
Forwarding token propagation	Local registry	When the registry forwards a query outward, the forwarding token includes the scope asserted by the originating entity's manifest. The providing registry may use this to enforce its own access controls.
Right of refusal at providing registry	Providing registry	A providing registry receiving a forwarded query may inspect the originating entity's forwarded manifest scope and refuse the request if incompatible with the providing registry's access policy.

4. Service Invocation Authentication

Service invocation authentication governs how a requesting agent establishes trust with a selected service agent at runtime. This section is distinct from Sections 1–3, which govern contexts where the registry is an active participant. At invocation time, the registry is not in the path.

The registry's role ends at discovery: it surfaces the service's invocable endpoint and OIDC endpoint as part of the manifest returned by `searchAndInvoke`. All subsequent authentication and invocation occurs directly between the requesting agent and the service agent. The registry is neither contacted nor involved for the duration of the invocation session.

The authentication lifecycle described in this section is encapsulated by the `searchAndInvoke` helper. Calling agents interact with a single invocation interface and do not manage the mechanics below directly. The normative requirements for this layer are maintained in the SADAR Invocation Authentication companion; this section describes the architecture narratively, and on any conflict the companion governs. Invocation flows that cross an organizational boundary are additionally governed by the enforcement gradient described in §4.13 and the **Cross-Organizational Flows** companion.

4.1 The Server as Sovereign Token Issuer

SADAR does not designate a central token authority for service invocation. Each service agent is the sovereign issuer of its own usage tokens. This design:

- Keeps the registry out of the runtime invocation path
- Ensures each service maintains sovereign control over its own access decisions and policy
- Eliminates single points of failure in the authentication chain
- Allows each service to apply its own governance policy at token issuance, independent of any central authority

This stands in contrast to registry authentication (Section 3), where the registry is the token issuer for query session tokens. At invocation time, the service issues the token; the registry's prior contribution was surfacing the service's OIDC endpoint at discovery.

4.2 Manifest Fields Supporting Service Invocation Authentication

Every SADAR-compliant service manifest carries the following fields that are directly relevant to invocation authentication. All required fields are covered by the manifest's JWS signature; any post-upload alteration is detectable by any validating party.

Field	Required	Purpose at Invocation
invokable_endpoint	Required	The direct URI at which the service agent accepts invocation requests. Used by <code>searchAndInvoke</code> for all runtime calls. Surfaced by the registry at discovery time; the registry is not involved at invocation time.
oidc_issuer	Required	URI identifying the service agent's OIDC authority. Serves as the root from which the requester derives the service's token endpoint. Also serves as the stable identifier for the service's identity domain. Present in the requester's manifest to identify the requester's own OIDC authority to the server.
jwks_uri or jwks	Required	The entity's public key set. Serves three functions: (1)

		verifying the manifest's JWS signature; (2) validating the JWT presented at first-use provisioning; (3) verifying DPoP proof signatures on every subsequent request.
a2a_card_uri	Optional	URI pointing to the service agent's A2A Agent Card at its well-known endpoint. Navigational only — for A2A-native clients who want the full A2A interaction contract. SADAR-aware clients use <code>invokable_endpoint</code> directly.

4.3 The Signed Manifest as Identity Anchor

A SADAR manifest is publicly accessible with its hash. Possession of the manifest alone proves nothing — any party can retrieve and present a public document. The manifest's function as an identity artifact depends on combining it with runtime proof-of-key-possession.

The signing key creates the binding: the manifest is signed with the entity's private key (JWS/ES256); the manifest's `jwtks` field contains the corresponding public key; at runtime, the entity presents a JWT also signed with its private key; the relying party verifies the JWT against the JWKS derivable from the manifest. Successful verification proves the presenter controls the key that signed the manifest — i.e., is the entity the manifest declares.

This means the JWKS in the manifest is load-bearing for three distinct verification operations: manifest signature verification, JWT validation at first-use provisioning, and DPoP proof verification on every subsequent request. One key set serves all three functions without additional key material.

4.4 Identity Granularity and Credential Mapping

The direct model for service invocation authentication — in which every requesting agent registers individual credentials at every service it calls — creates a credential management problem that scales as the product of the number of requesters and the number of services. SADAR resolves this through a registry-mediated indirection: the credential used to authenticate a request is not bound directly to the individual agent, but to a configured identity scope that may represent an entity, a project, a division, or the individual agent. The registry's `credential_scope` field on each agent registry entry defines this mapping.

When `searchAndInvoke` makes the OIDC authentication call on behalf of an agent, it presents the agent's registry identity. The OIDC server resolves the `credential_scope` mapping and issues a token reflecting the configured scope — not necessarily the individual agent. The service being called therefore registers

a trust relationship with the issuer endpoint of the mapped identity (an entity, project, or other scope) rather than with each individual agent. Multiple agents within the same entity or project share a single registered trust relationship at each service, without sharing credentials.

The `credential_scope` field accepts the following configured values:

- **agent** — the authentication credential is bound to the individual agent registry entry. Services register a trust relationship with this agent's identity specifically. This scope is appropriate for agents with highly differentiated access requirements or regulatory obligations that mandate agent-level accountability.
- **entity** — all agents within the entity share a single service-facing identity. Services register a trust relationship with the entity's OIDC issuer once; all agents within that entity are authenticated under the entity's credential.
- **project** — agents grouped under a common project or business process share a single service-facing identity. Useful for organizing trust relationships by functional boundary rather than organizational boundary.
- **division** — agents within an organizational division or business unit share a service-facing identity scoped to that division.

The `credential_scope` mapping is an administrative property of the agent's registry entry, established at provisioning time and not modifiable at runtime by the calling application. The scope of the issued authentication token is determined by the registry; the calling application cannot assert a different scope. The individual agent's identity within that scope — and all other authorization context — is carried separately in the SADAR Context Token (see Section 4.6).

4.5 Agent Credential Isolation

SADAR is designed to overlay any agentic framework without requiring framework-specific credential management code in the agent itself. Agents are not expected to know how to authenticate; authentication is an infrastructure concern managed by `searchAndInvoke` on the agent's behalf. The governing security property that follows from this design principle is agent credential isolation: the private credential used to authenticate on behalf of an agent must not be directly accessible to the agent's application logic.

This requirement does not prescribe how isolation is achieved. It constrains only the outcome: the credential must not be retrievable by code executing within the agent's own logic boundary. `searchAndInvoke`, operating in the agent's infrastructure layer rather than its application layer, accesses the credential via the deployment's provisioned trust boundary and uses it to produce a signed assertion. The agent's application code never touches the credential material.

The normative requirements for credential binding and isolation are stated in Appendix A. Non-normative guidance on conformant implementation patterns (platform workload identity, credential broker sidecar, and framework-level injection) is provided in Appendix A, Section A.4.

Each conformant agent deployment is provisioned with a private credential at registration time, prior to any invocation. The credential is scoped to the individual agent registry entry regardless of the `credential_scope` mapping — the scope mapping governs how the agent's identity is presented to services, not how many credentials are provisioned. An agent whose `credential_scope` is `entity` still has its own provisioned credential; the entity mapping is resolved by the OIDC server at token issuance, not by sharing credentials across agents.

4.6 SADAR Context Token (SCT) — Orientation

The OIDC JWT obtained during first-use provisioning establishes authentication at the configured `credential_scope` granularity. It answers the question of who is permitted to call the service at the infrastructure level. It does not carry the full context of what is being requested, why, on whose authority, and as part of what business process. That context is carried in the **SADAR Context Token (SCT)**.

The SCT is a per-invocation authorization-context token constructed and signed by `searchAndInvoke` and transmitted in the dedicated **SADAR-SCT** HTTP header on every invocation, alongside the usage token in the `Authorization` header — never embedded within it. Both are required on every request: the usage token provides authentication and replay protection; the SCT provides authorization context and chain-of-custody. The separation is deliberate — the authentication layer is stable across an invocation session and represents a registered trust relationship; the SCT is specific to each invocation and represents the runtime assertion of what is being done. Keeping them separate prevents the authentication credential from becoming overloaded with policy context that varies per call, and allows the `credential_scope` mapping to operate independently of the authorization policy enforced at each service.

Structurally, the SCT is a **signed segment chain**: each segment is a JWS signed by its **custodian of record** (the environment's `searchAndInvoke`/platform identity, or the home registry for boundary and non-conformant segments), with all claims carried in the clear under signature so that admissibility is evaluable at every enforcement point, and with each segment transitively bound to its whole path by a signed chain manifest. The acting component is named in the segment's `actor_id` and `operation_id` claims and, where it declares the capability, embeds its own bound **actor attestation**. The flow's scope of authority travels as a **producer-signed authority grant** (§4.13). **The SCT carries no secrets** — no credentials, no authentication evidence artifacts, no business inputs; evidence and rich narrative belong to the Telemetry Record under the repatriation regime.

The complete normative treatment lives in the SCT document set, which this section only orients:

Companion	Defines
SCT Structure	The canonical segment schema, claim definitions, enumerations, and literal serialization

9. SCT Operations (Appendix B companion)	The five chain operations (Initiate Chain, Validate, Extract Claims, Append Segment, Verify Chain Integrity), error contract, and cryptographic suites
SCT Propagation	Transport, cross-hop custody, session-store security, boundary crossing, and the custody signing model
10. Trust Models	The four originator trust models and their negotiation
Cross-Organizational Flows	Authority, enforcement, and verification across organizational boundaries

4.7 First-Use Provisioning and the Time-of-Use Token

When a requesting entity invokes a service agent for the first time, a provisioning exchange establishes the bilateral trust relationship. This exchange is orchestrated transparently by `searchAndInvoke` and occurs **once per requester↔service pair**:

- `searchAndInvoke` authenticates to the requester's OIDC token endpoint using a signed JWT assertion (RFC 7523 — JSON Web Token Profile for OAuth 2.0 Client Authentication). The assertion is signed with the agent's provisioned private credential, accessed via the agent's trust boundary as described in Section 4.5. The credential is never passed through agent application logic. The OIDC server validates the assertion against the agent's registered public key and issues a JWT whose identity scope reflects the agent's `credential_scope` registry mapping (see Section 4.4).
- `searchAndInvoke` presents the JWT plus a manifest reference (Entity URN + `oidc_issuer` URI) to the service agent's invocable endpoint.
- The service agent resolves the requester's `oidc_issuer` URI to its discovery document (`{issuer}/.well-known/openid-configuration`) and retrieves the `jwt_uri`.
- The service fetches the requester's JWKS and validates the JWT signature. Proof-of-key-possession confirmed: the presenter controls the private key corresponding to the JWKS public key, which is the same key that signed the manifest.
- The service validates JWT claims: `iss` must match the `oidc_issuer` declared for the mapped identity; `sub` must match the identity established by the agent's `credential_scope` mapping — which may be an entity URN, project URN, or other configured granularity rather than the individual agent URN. The service does not require the individual agent URN in the authentication credential.
- The service provisions access for the mapped identity and issues a **time-of-use usage token**.

Time-of-use token binding. The usage token is TTL-bound and bound to the tuple (**requesting actor identity, intent, intent instance**) — the actor's registry identity (presented at the `credential_scope` granularity, with the individual actor carried in the SCT), the intent under execution, and the intent instance (nonce; aligning with the SCT's `intent_instance_id`). The token is therefore scoped to a specific transaction instance, not merely to a requester↔service pair: a cached token is not presentable for a

different flow, and a new intent instance triggers acquisition of a fresh token (a silent OIDC re-authentication — *not* a repeat of first-use provisioning, which remains one-time per pair).

Dual TTL, most stringent first. Both the client and the server set a TTL on the time-of-use token; **both literally apply**, and whichever is shorter triggers refresh first. The **client owns the caching decision** — most scenarios do not warrant per-invocation re-authentication within the same transaction instance; the TTL is the refresh driver, sized for long-running flows where re-authentication mid-instance is warranted.

After successful first-use provisioning, the service may cache the requester's JWKS for the duration of the manifest's declared TTL, refreshing on manifest version change or TTL expiry. The service does not call the requester's OIDC endpoint again unless the requester's manifest changes — subsequent token acquisition uses the established relationship.

4.8 Runtime Authentication Flow

After first-use provisioning, subsequent invocations from the same requesting entity are authenticated using the cached time-of-use token:

- `searchAndInvoke` presents the cached token as a Bearer token in the Authorization HTTP header, provided the current invocation belongs to the intent instance the token is bound to and the token is within both parties' TTLs.
- A DPoP proof accompanies each request (see Section 4.9) — the per-request, single-use artifact; the time-of-use token itself is reusable within its binding and TTL bounds.
- The service validates the usage token (including its intent-instance binding) and DPoP proof before processing the request.
- Token refresh: proactive refresh when the effective (most stringent) exp is within a configurable threshold (default 30 seconds); reactive refresh on 401 response; fresh acquisition on a new intent instance. The service's OIDC endpoint is called only at token acquisition and refresh — not on every request.

4.9 DPoP Replay Protection

Standard JWT Bearer tokens are not inherently replay-proof: a captured token remains valid until its expiry timestamp. SADAR mandates DPoP — Demonstration of Proof of Possession (RFC 9449) — to eliminate this window.

For each request, the requester generates a DPoP proof: a short-lived JWT containing the HTTP method and URI of the specific request (htm and htu claims), a unique jti not previously seen by this service, an iat timestamp with a short validity window, and signed with the requester's private key.

The service validates all of the following before processing:

- The access token (standard Bearer token validation, including the time-of-use binding of §4.7)
- The DPoP proof signature, using the same JWKS used at first-use provisioning
- That the jti has not been seen before within the expiry window (the service maintains a jti seen-set)
- That the htm and htu claims match the actual request method and URI

A captured usage token is useless without a valid DPoP proof — and useless outside the intent instance it is bound to. A captured DPoP proof is useless because its jti is already in the seen-set and its htm/htu are bound to a specific request. The combination provides per-request, method-and-URI-bound replay protection without a challenge-response round trip. The requester's JWKS — derivable from the manifest — serves all three verification functions: manifest signature, first-use JWT validation, and per-request DPoP proof. No additional key material is required.

4.10 A2A Protocol Interoperability

The SADAR authentication model is fully compatible with the Agent2Agent (A2A) protocol's authentication declaration model. A2A Agent Cards declare authentication requirements using OpenAPI 3.2 Security Scheme Objects, which support the same schemes SADAR uses: openIdConnect, oauth2 (Client Credentials flow), and mtls. The service's OIDC endpoint declared in the SADAR manifest corresponds directly to the openIdConnectUrl in an A2A Agent Card's openIdConnectSecurityScheme — they reference the same identity authority.

A2A explicitly requires that credential acquisition be out-of-band — the A2A protocol does not define how clients obtain tokens. searchAndInvoke is that out-of-band process, standardized by SADAR. There is no conflict between the two models; they operate at complementary layers.

4.10.1 The a2a_card_uri Field

A SADAR manifest may optionally carry an a2a_card_uri field pointing to the service agent's A2A Agent Card at its well-known endpoint (e.g. <https://agent.example.com/.well-known/agent-card.json>). The scope of this field is strictly navigational:

- SADAR-aware clients use invokable_endpoint in the manifest directly. No additional hop to the A2A card is required.
- A2A-native clients who encounter the manifest via a registry query may follow the a2a_card_uri to retrieve the full A2A interaction contract — skills, streaming capabilities, push notification support, and other A2A-specific interaction details not carried in the SADAR manifest.
- The a2a_card_uri also serves as a verifiable cross-reference: a consumer can confirm that the SADAR manifest and the A2A Agent Card describe the same service agent.

There is no change to the A2A Agent Card itself. SADAR does not modify, extend, or replace the A2A card. The `a2a_card_uri` in the SADAR manifest is a one-way pointer from the SADAR discovery artifact to the A2A invocation artifact. The SADAR registry stores manifests. It does not store A2A Agent Cards.

4.10.2 Relationship to A2A Discovery

A2A's own discovery model requires the client to already know the agent's domain before fetching its card from `/.well-known/agent-card.json`. SADAR provides the prior step: finding which agent to contact in the first place, based on semantic capability matching via `searchAndInvoke`. SADAR is the semantic discovery layer that A2A explicitly leaves undefined. Once discovery resolves a service, the invocation proceeds — via SADAR's `invokable_endpoint` or via A2A's interaction contract, at the client's discretion.

4.11 `searchAndInvoke` Token Management Summary

The `searchAndInvoke` helper encapsulates the full invocation authentication lifecycle:

- Retrieves the service's OIDC endpoint and invokable endpoint from the manifest returned by discovery
- Calls the requester's own OIDC endpoint to obtain the initial JWT for first-use provisioning
- Manages the first-use provisioning exchange with the service agent
- Caches the issued time-of-use token per (service endpoint, intent instance), within the dual-TTL bounds; the client owns the caching decision
- Generates and attaches a DPoP proof to each request (RFC 9449)
- Proactively refreshes the token when the effective exp is within a configurable threshold (default: 30 seconds); acquires fresh tokens on new intent instances
- Handles reactive token refresh on 401 responses
- Optionally follows `a2a_card_uri` for A2A-native invocation when requested

The calling agent interacts with a single invocation interface. It does not manage authentication tokens, DPoP proofs, OIDC endpoints, or JWKS resolution directly.

4.12 Accountability Table

Control	Owner	Mechanism / Notes
Requester private key custody	Requesting entity	Never leaves the requesting entity's environment. Used to sign the JWT presented at first-use provisioning and to generate DPoP proofs on every request.

Requester manifest integrity	Requesting entity + local registry	JWS signature over manifest content, including <code>oidc_issuer</code> and <code>jwtks</code> fields. The signed manifest binds the Entity URN to the OIDC issuer URI and public key. Post-upload tampering is detectable by any validating party.
First-use identity verification	Service agent	Service fetches requester's JWKS via <code>oidc_issuer</code> discovery document and validates the presented JWT signature. Proof-of-key-possession confirms the presenter controls the key that signed the manifest.
Time-of-use token issuance	Service agent	The service is the sovereign issuer of its own usage tokens, bound per §4.7 to (actor, intent, intent instance) with dual TTLs. No central token authority. Each service applies its own access policy at token issuance.
Token caching	searchAndInvoke helper	Client-owned caching per (service endpoint, intent instance), within dual-TTL bounds. Proactive refresh when the effective exp is within a configurable threshold (default 30 seconds).
Replay protection	searchAndInvoke + service agent	DPoP (RFC 9449): per-request proof JWT bound to HTTP method, URI, jti, and iat. Service maintains jti seen-set within expiry window. Time-of-use binding scopes the token to one flow instance. Captured tokens and proofs cannot be replayed.
Token expiry and refresh	searchAndInvoke helper	Proactive refresh via silent re-authentication to the service's OIDC endpoint; most stringent of client/server TTLs governs. Reactive refresh on 401 response. Fresh acquisition per intent instance.

Authority establishment	Deployment-defined process (§4.13)	Producer-signed authority grant, verified at genesis and conveyed in the SCT. SADAR is the conveyor, never the author.
A2A interoperability	searchAndInvoke helper	searchAndInvoke is the out-of-band credential acquisition process that A2A explicitly leaves undefined. The service's <code>oidc_issuer</code> in the SADAR manifest corresponds to the <code>openIdConnectUrl</code> in an A2A Agent Card security scheme.

4.13 Authority Establishment and Organizational Boundaries

Authority establishment. The scope of authority governing a flow — an RFC 9396 `authorization_details` structure carrying largely *functional* boundaries (spending limits, data scopes, process constraints) — is produced by a **deployment-defined authority-establishment process** invoked by `searchAndInvoke` at flow genesis. That process is outside this specification: it may consult the IAM as one input, but functional authority is not an identity-layer artifact, and SADAR neither defines the producer nor constrains its logic. SADAR's obligations are conveyance with integrity: the producing authority **signs the grant** (a nested JWS under its own key — the producer attests content; the custodian attests carriage; the same testimony idiom as the actor attestation); the genesis trust boundary **verifies the grant before binding it** into the signed root (9. SCT Operations B.0); and the chain carries it, readable at every hop, unaltered for the life of the flow. Verification thereafter is elective — the expected posture, mirroring registry ingestion, is that consumers trust the genesis-point verification plus the custodian's carriage signature, re-verifying where risk warrants. Conveyance of the verified originator identity into the genesis trust boundary may take any form the deployment can attest — an authenticated session token `searchAndInvoke` validates against the IdP, an RFC 8693 token exchange presenting the platform's token as `actor_token` and the originator's as `subject_token`, a credential-broker sidecar, or workload identity — provided the identity stamped into the chain is one the trust boundary verified, never one application code asserted.

Two-tier verification. Authorization is evaluated **as established at genesis, bounded by the chain's expiry** — the signed-check semantics. Within the originating trust boundary, zero-trust enforcement MAY additionally revalidate live, at the enforcement point's own standing and never with the originator's credentials: the IAM layer in the usual way (a decision query keyed by the chain's originator reference), and the authority grant via the deployment's validation routing (the grant's `granted_by` provenance identifies the establishing authority; the deployment's configuration maps it to a validation interface). **Only the genesis trust boundary ever handles originator credentials.** No interoperable binding for cross-organizational zero-trust enforcement is assumed to exist; live validation is trust-boundary-internal.

The organizational boundary. When a flow crosses into another organization, the receiving side is a black box whose published manifest is the interface contract, and enforcement is the receiving domain's alone — designed by the party that built the systems, in its own vocabulary, operating on what is directly passed. The chain's cross-boundary role is provenance, integrity, completeness, and audit; its readable context is available to the receiver, never binding on it. Within the originating organization the authority grant is an active enforcement mechanism across loosely federated services; the egress hop is its final

enforcement point and MAY map grant content into the external interface's shape, exactly as roles are mapped into SSO assertions today. The complete treatment — conceptual, architectural, and implementation — is the **Cross-Organizational Flows** companion.

Appendix A: Requirements — Agent Credential Binding and Invocation Authentication

This appendix states the requirements that flow from the architectural design described in Sections 4.4, 4.5, and 4.6. These requirements, with their identifiers (A.1.x–A.4.x), are defined normatively in the SADAR Invocation Authentication companion; they are restated here for the reader's convenience. Requirements are expressed as SHALL (mandatory), SHOULD (strongly recommended), and SHALL NOT (prohibited). Conformance is assessed against these statements, not against specific implementation mechanisms. Non-normative implementation guidance is provided in Section A.4.

Note: While every effort is taken to ensure the currency and consistency of this document versus the normative specifications on the project's [GitHub](#), that remains the canonical source that takes precedence over any contradictions, additions, or omissions.

A.1 Agent Credential Binding

- **A.1.1** — Each registered agent entity SHALL have an associated authentication credential bound to its SADAR registry identity at provisioning time. This credential SHALL be the basis for all authentication assertions made on behalf of that agent.
- **A.1.2** — The credential binding SHALL be established prior to any agent invocation. Credential binding during an active invocation transaction is not permitted.
- **A.1.3** — The credential SHALL NOT be directly accessible to agent application logic. Conformant implementations MUST ensure that the credential is not retrievable by code executing within the agent's own logic boundary.
- **Note:** The mechanism by which credential isolation is achieved is an implementation concern. Conformant approaches include but are not limited to: platform-attested workload identity (e.g. cloud provider managed identity services), credential broker sidecar services exposing a local loopback signing endpoint, and framework-level injection prior to agent execution. See Section A.4 for non-normative guidance on each pattern.
- **A.1.4** — The credential SHALL be scoped to a single registered agent identity. Credentials shared across multiple distinct agent registry entries do not satisfy this requirement.
- **Note:** Where the registry credential_scope field resolves an agent identity to a broader scope (entity, project, division), the credential itself remains agent-scoped. The broader scope is a property of the identity mapping at the OIDC server, not of credential sharing between agents. Multiple agents may present the same service-facing identity through the mapping without sharing private credentials.

A.2 Invocation Service Authentication

- **A.2.1** — The SADAR Invocation Service acting on behalf of a registered agent SHALL authenticate to the target service's authorization server using a signed assertion that demonstrates binding to that agent's registered credential.
- **A.2.2** — The signed assertion SHALL conform to RFC 7523 (JSON Web Token Profile for OAuth 2.0 Client Authentication and Authorization Grants) or an equivalent mechanism that provides equivalent non-repudiation and replay protection properties.
- **A.2.3** — The Invocation Service SHALL NOT transmit the agent's private credential material over any network boundary to obtain an authentication token. The credential SHALL be used only to produce a signed assertion within the agent's trust boundary.
- **A.2.4** — The authentication token returned by the authorization server SHALL reflect the identity granularity defined by the agent's registry credential_scope mapping. The mapping SHALL resolve to one of the following configured scopes: agent, entity, project, or division. The resolved scope SHALL be established in the registry and SHALL NOT be determined at invocation time by the calling application.
- **A.2.5** — The Invocation Service SHALL NOT accept caller-supplied credential material as a substitute for the registered agent credential. The credential used for authentication SHALL be obtained exclusively from the agent's provisioned trust boundary.

A.3 Credential Lifecycle

- **A.3.1** — Conformant implementations SHALL support credential rotation without requiring re-registration of the agent's SADAR registry entry.
- **A.3.2** — Revocation of an agent credential SHALL be propagatable to all authorization servers with which that credential's mapped identity is registered within a time window defined by the deployment's security policy.
- **A.3.3** — Credential expiry and rotation SHALL be managed outside the agent invocation path. An agent invocation SHALL NOT trigger credential provisioning.

A.4 Non-Normative Implementation Guidance

The following patterns represent conformant approaches to satisfying the requirements in Sections A.1 through A.3. Implementors are not required to use any specific pattern; any approach that satisfies the normative requirements is conformant. Patterns are listed in order of preferred security posture.

A.4.1 Platform Workload Identity (Preferred)

The deployment platform attests the workload's identity at launch and makes a signed token available via a local metadata endpoint. `searchAndInvoke` calls the local endpoint to obtain a token representing the agent's identity; no private key is ever retrieved by application or infrastructure code. Examples include cloud provider managed identity services and SPIFFE/SPIRE workload identity frameworks. This pattern

satisfies A.1.3 architecturally: the private key does not exist as a retrievable artifact in the application environment.

A.4.2 Credential Broker Sidecar

A minimal sidecar process co-deployed with the agent holds the agent's private credential and exposes only a local loopback signing endpoint. `searchAndInvoke` submits a claims payload to the sidecar and receives a signed assertion in return; it never receives or holds the private key. The sidecar's secrets manager access is scoped to the single agent credential by deployment configuration. This pattern is appropriate for deployment targets that do not support platform workload identity and provides a comparable blast-radius boundary: a compromise of the sidecar exposes one agent's credential, not all agents' credentials.

A.4.3 Framework-Level Injection

In framework environments that support pre-invocation hooks (decorators, middleware, or explicit setup calls prior to agent execution), the deployment developer retrieves the agent's private credential from a secrets manager and injects it into a session-scoped context accessible to `searchAndInvoke`. The credential is held in a controlled in-memory area for the duration of the session. `searchAndInvoke` reads from this context rather than from the secrets manager directly. The developer is accountable for secrets manager access; `searchAndInvoke` receives the credential via the injected context rather than fetching it independently. This pattern satisfies A.1.3 when the injection occurs prior to agent logic execution and the context is not accessible to agent application code. It is appropriate only where the framework provides a reliable pre-execution injection point that the agent cannot subvert.

A.4.4 Conformance Priority

Implementations **SHOULD** prefer platform workload identity where the deployment target supports it. Where platform workload identity is unavailable, a credential broker sidecar is the recommended alternative. Framework-level injection is permitted where neither of the preceding patterns is applicable. Implementations that give `searchAndInvoke` direct, unrestricted access to a secrets manager containing credentials for multiple agents are permitted but not recommended; such implementations **SHOULD** document the scope restriction applied to limit `searchAndInvoke`'s secrets manager access to the agents it is authorized to represent.

*Appendix B — SADAR Context Token Operations — is maintained as the separate companion file **9. SCT Operations** and is cross-referenced from §4.6.*