DEFENSIVE PUBLICATION — TECHNICAL DISCLOSURE (TDCOMMONS DEPOSIT COPY) · 2026-06-25

Keywords: defensive-publication, prior-art, ai-agents, rbac, access-control, authorization, multi-tenant

Resource-Provider RBAC for Multi-Domain AI Platforms

A boot-time resource-provider registry with path-scoped, wildcard-composable role resolution and a resolver-invariant registration contract

Field	Value
Document type	Technical Defensive Publication (public prior art)
Title	Resource-Provider RBAC for Multi-Domain AI Platforms
Author	Gustavo Assuncao, PhD
Publisher / Copyright	Gus IT LLC (Florida, USA)
Publication date	2026-06-25
Version	1.0
Classification	Public
License	AGPL-3.0-or-later (copyleft) + commercial license available (Gus IT LLC)
Deposit channel	To be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record.
Field	Access control in multi-tenant, multi-domain AI-agent platforms.

Abstract

Flat role-based access control — `admin` / `user` / `viewer` — does not scale to a platform hosting many independently-developed domains (CRM, communications, finance, documents, AI personas) that appear and disappear at runtime and share resources across domain boundaries. Hardcoding a capability map forces a code deploy for every new permission; scattering per-domain access-control lists duplicates safety-critical logic and drifts. This publication describes a **resource-provider RBAC** model for AI-agent platforms in which each domain module **registers its own resource catalogue at boot** — resource types, each with a set of actions and a schema — into a central registry. Every permission is a tuple `domain:resourceType:action`. Roles are sets of permission **patterns** with `*` wildcards (`crm:*:*`, `*:*:read`). Role **assignments** bind a principal — user, group, API token, **or AI persona** — to a role at a **scope path** in a resource hierarchy. Authorization is resolved by **walking the scope path from the queried node toward the root**, unioning the permission patterns of all assignments encountered, expanding role-template `{scope}` placeholders against the assignment scope, and testing the requested tuple against the union with three-axis wildcard matching. The defining property is the **resolver-invariant registration contract**: adding a domain, a resource type, or an action requires *no* change to the resolver, the schema, the roles, or any deploy. We present architecture, a state-machine and data-flow view, a full relational data model, a worked cross-domain sharing example with a sequence diagram, a STRIDE-style threat model, semantic alignment to NIST RBAC, XACML, Google Zanzibar and Azure Resource Manager, an evaluation

methodology, a clean-room Node.js reference implementation that reduces the method to practice, and one independent plus sixteen dependent claims. The publication is intentionally public to bar later patenting by others.

---

1. Executive Summary

1.1 Thesis

A multi-domain AI-agent platform should treat **access-control vocabulary as a plug-in supplied by each domain at boot**, not as a fixed map baked into the authorization core. When the permission vocabulary is owned by the domains and the resolver is invariant to it, the platform gains custom roles, cross-domain sharing, and AI personas as governed principals — **without a single line of resolver code changing** as the platform grows from three domains to thirty.

1.2 Contributions

#	Contribution	Where
C1	A **boot-time resource-provider registry**: each domain declares `{resourceType → {actions, schema, flags}}` and the platform assembles a global resource catalogue.	§6, §7
C2	A **three-axis permission tuple** `domain:resourceType:action` with * wildcards on each axis.	§7.2
C3	**Path-scoped role assignments with hierarchical inheritance** resolved by root-ward traversal of a scope tree.	§7.3
C4	The **resolver-invariant registration contract** — the characterizing property: new vocabulary never touches the resolver, schema, or roles.	§7.5
C5	**AI personas (and API tokens) as first-class typed principals** in the same assignment model as humans, so agent tool-calls pass the same resolver.	§7.6, §10
C6	**`{scope}`-templated roles** (e.g. `{scope}:*:read`) that bind to whatever domain the assignment names — one role definition, many domains.	§7.4
C7	A **cross-domain resource bridge** (time-boxed, audited shares) layered on the same tuple model.	§7.7
C8	A clean-room **reference implementation** reducing the method to practice.	§9, Appendix C

1.3 Headline claim

> An access-control method in which **domain modules register their own resource types and actions at boot into a registry consulted by, but never modified by, the permission resolver**, where permissions are `domain:resourceType:action` tuples matched with per-axis wildcards, and decisions are resolved by walking a scope hierarchy from the queried path toward the root and unioning the permission patterns of every role assignment encountered.

1.4 Scope — what this is and is NOT

It is: an authorization *model and resolution algorithm* for platforms that host many independently-developed domains and AI agents; a contract by which domains contribute permission vocabulary; a path-

scoped, wildcard-composable decision procedure.

It is NOT: an authentication / identity-provider system (it consumes an already -authenticated principal); a policy *language* competitor to XACML/Rego (patterns are deliberately minimal); a network policy or service-mesh authorization layer; a claim that the *individual* ingredients (tuples, wildcards, scope inheritance) are novel in isolation — the contribution is their **combination plus the boot-time, resolver-invariant, AI-persona-inclusive framing**.

---

2. Introduction & Motivation

2.1 The concrete problem

Consider a platform that began with three flat roles and a hand-written capability map:

```
admin    → CAN_MANAGE_USERS, CAN_MODIFY_SETTINGS, CAN_SEND_EMAIL, ...
user     → CAN_USE_LLM, CAN_SEND_EMAIL, CAN_START_CALLS, ...
viewer   → CAN_READ_EMAIL, CAN_READ_CALENDAR
```

It now hosts a dozen business domains and a fleet of AI personas that read CRM leads, draft emails, and file documents on a user's behalf. Three forces break the flat model:

1. **Granularity.** "User X may *read* CRM leads but not *delete* them" cannot be said with `user`/`viewer`. Every distinction means a new hardcoded capability.
2. **Domain proliferation.** A new `finance` domain ships next week. With a hardcoded map, its permissions (`CAN_READ_INVOICE`, `CAN_VOID_INVOICE`, …) must be added to the access core and re-deployed — coupling every domain's release to the authorization module.
3. **Cross-domain & non-human principals.** CRM wants to share a lead with Finance for invoicing. An AI persona must read CRM but never delete it. Flat roles model neither sharing nor typed non-human principals.

2.2 The "tax" of the status quo

Approach	Recurring tax
Hardcoded capability map	Every new permission ⇒ edit the auth core ⇒ a coupled deploy. The auth module becomes a merge bottleneck and a single point of regression for *every* domain.
Per-domain ACLs	The matching/wildcard/inheritance logic is **copy-pasted** into each domain. A copy-pasted authorization check is a defect waiting to drift: a bug fixed in CRM stays unfixed in Finance. No global audit, no global "who-can-do-what".
Off-the-shelf cloud RBAC (IAM-style)	Powerful but cloud-resource-shaped; it has no notion of *application domains registering vocabulary at boot*, nor of *AI personas* as principals whose tool-calls must pass the same resolver.

2.3 Why existing approaches fall short

Cloud IAM systems (Azure RM, AWS IAM, Kubernetes RBAC) already have tuples, wildcards and scope inheritance — and we build *on* them (§4). What none of them provides is the **inversion of ownership** at the heart of this disclosure: the *application domains*, loaded as plug-in modules, **own and contribute** the permission vocabulary at **boot**, and the resolver is contractually forbidden from knowing any specific

vocabulary. The cloud systems assume a fixed, vendor-defined resource-provider namespace; here the namespace is open and grows with the app catalogue, including AI-agent resources.

2.4 Why now

AI-agent platforms host *more* distinct, fast-moving capability surfaces than traditional SaaS, and they introduce a new principal type — the **autonomous persona** — that acts on resources without a human in the loop for each action. An authorization model that lets capabilities self-register at boot and that treats a persona exactly like a scoped user is the natural fit, and it is the moment the flat model becomes untenable.

---

3. Problem Statement

3.1 Formal framing

Let **D** be a set of domain modules, each loaded at boot. Each domain `d ∈ D` publishes a resource catalogue

```
Rd = { (t, At, Schemat, Flagst) }
```

where `t` is a resource type, `At` the finite set of actions on `t`, `Schemat` a field map, and `Flagst` properties such as `searchable`/`shareable`. The global catalogue is `R = ∪_{d∈D} {d}×Rd`.

A **permission** is a tuple `p = (dom, type, act) ∈ (D∪{*}) × (T∪{*}) × (A∪{*})`. A **role** `ρ` is a finite set of permission patterns. A **role template** may contain the placeholder `{scope}` in its `dom` axis. An **assignment** is `a = (principalType, principalId, ρ, scopePath)` where `scopePath` is a node in a scope tree `S` whose root is `/`.

The decision function we require is

```
authorize(principal, dom, type, act) ∈ {ALLOW, DENY}
```

3.2 Requirements

ID	Requirement	Satisfied in
R1	A domain module shall register its resource types and actions **at boot**, declaratively, without imperative calls into the resolver.	§6, §7.1
R2	Adding a domain, resource type, or action shall require **no change** to the resolver, the database schema, or any existing role (resolver-invariant contract).	§7.5
R3	Permissions shall be `domain:resourceType:action` tuples with independent * wildcards on each axis.	§7.2
R4	Roles shall be authored as **pattern sets** that apply to resource types not yet registered when the role was written.	§7.4
R5	A role assignment shall bind a principal to a role at a **scope path**; permissions shall **inherit downward** along the scope hierarchy.	§7.3
R6	Authorization shall be resolved by traversing the scope hierarchy **from the queried path toward the root** and unioning all encountered patterns.	§7.3
R7	Principals shall be **typed** — user, group, API token, AI persona — and resolved uniformly.	§7.6
R8	Role templates shall support a `{scope}` placeholder bound to the assignment's named domain.	§7.4
R9	Cross-domain sharing shall be expressible as time-boxed, audited grants over the same tuple model.	§7.7
R10	Every decision and every grant shall be **auditable** (who, what, where, when, why).	§7.8, §11
R11	The default posture for an unmatched request shall be **fail-closed (DENY)**.	§11.2
R12	Assignments shall support **optional expiry**.	§8, §7.7

4. Related Work & Prior Art

We build deliberately on a mature lineage; none of these ingredients is claimed novel in isolation. The contribution is the combination and AI-platform framing (§5).

Source	What we adopt / acknowledge
NIST RBAC (ANSI INCITS 359-2004; Sandhu et al., 1996)	Roles as an indirection between users and permissions; role hierarchies; least privilege as a goal.
Azure Resource Manager RBAC	The `Provider/resourceType/action` action string, * wildcards, and **scope inheritance** (Management Group → Subscription → Resource Group → Resource). This is the closest structural ancestor; we adapt its *shape* to app domains.
AWS IAM	`service:Action` action namespacing, wildcard policies, and resource-ARN scoping.
Kubernetes RBAC	`apiGroups × resources × verbs` Roles, and namespaced vs cluster scope — a clear precedent for grouped resource verbs.
Google Zanzibar (Pondichik et al., 2019)	Relationship-/tuple-based authorization at scale; the idea of a uniform authorization tuple consulted by all services.
OASIS XACML 3.0	Attribute-based policy, combining algorithms, and the PDP/PEP/PIP separation we echo (resolver = PDP; route guard = PEP; registry = PIP).
OSGi Declarative Services / module systems	The pattern of **modules registering capabilities at activation**, which we apply to permission vocabulary specifically.
OAuth 2.0 scopes	Token-bound permission scoping, which our API-token principal type generalizes into the same tuple model.

The patentability screen for this subject matter (US ≈ 55/100, DE/EPO ≈ 42/100) concluded that the *isolated* ideas are anticipated by Azure RBAC, AWS IAM, and Kubernetes RBAC, and that defensive publication is the appropriate posture. This document is that publication.

---

5. Prior-Art Delta

Per novel feature, what the closest prior art **has**, what it **lacks**, and what **this** adds.

Prior source	What it has	What it LACKS	What THIS adds
Azure RM RBAC	`Provider/type/action`, `*` wildcards, scope inheritance over a fixed cloud-resource tree.	Open, **application-domain-owned** vocabulary that **self-registers at boot**; AI personas as principals.	Domains (loaded plug-ins) register `{type,actions,schema}` at boot; resolver is invariant to the vocabulary.
AWS IAM	`service:Action` strings, wildcard policies, resource scoping.	A hierarchical **scope-path** inheritance walk; a registry that *grows with the app catalogue*; persona principals.	Root-ward scope traversal unioning patterns; boot-registered catalogue; typed persona/token principals.
Kubernetes RBAC	`apiGroups×resources×verbs`, namespaced/cluster scope, CRDs adding resource types.	A single platform-wide **`{scope}`-templated** role applying across heterogeneous business domains; cross-domain *sharing*; AI-agent tool-call enforcement.	`{scope}` role templates; cross-domain time-boxed shares; same resolver guards agent tool-calls.
Google Zanzibar	Uniform authorization tuples; relationship graph; scale.	A **boot-time provider-registration contract** owned by app modules; per-axis `domain:type:action` wildcards; declarative resource schemas.	Boot self-registration of `{type,actions,schema}`; three-axis wildcard tuples; metadata/discovery surface.
OASIS XACML	PDP/PEP/PIP, ABAC, combining algorithms.	A **lightweight, domain-self-describing** model with no policy language; boot registration; persona principals.	Minimal pattern model + registry-as-PIP populated at boot by domains; AI-platform principal taxonomy.
Flat app RBAC (admin/user/viewer)	Simplicity.	Granularity, custom roles, scope, cross-domain, discovery, non-human principals.	All of the above via the registry + tuple + scope-walk combination.

The characterizing delta (R2): in every prior system the set of resource *providers* and their action vocabulary is **defined by the platform/vendor**; here it is **contributed by the application domains at boot**, and the resolver is **contractually invariant** to it. That inversion, combined with AI-persona principals passing the same resolver, is the disclosed point.

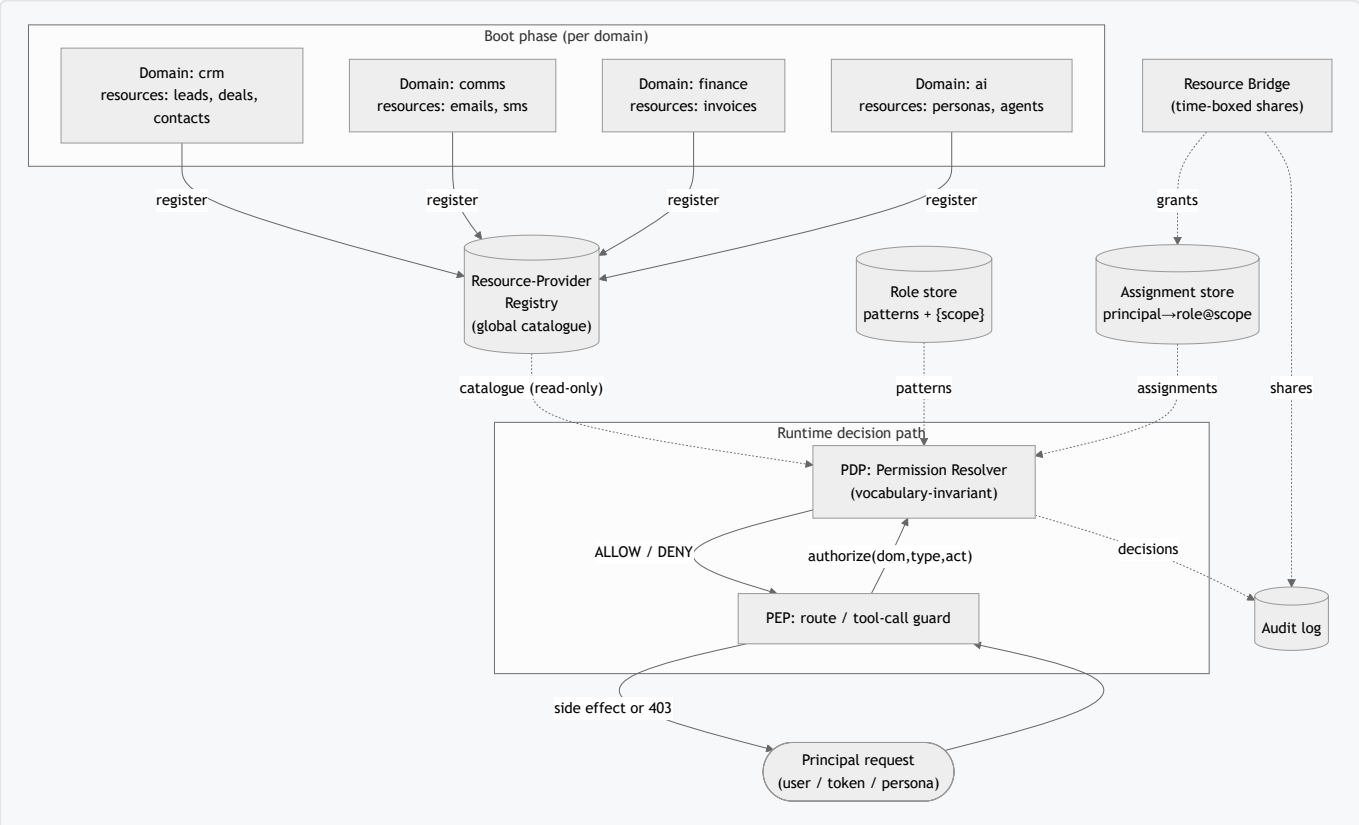
---

6. System Architecture

6.1 Components

Component	Role (XACML analogue)	Responsibility
Domain modules	Resource providers	At boot, declare `resources: {type → {actions, schema, flags}}`.
Resource-Provider Registry	PIP (info point)	Aggregates all domain declarations into a global catalogue; serves discovery (`$metadata`).
Role store	Policy store	Built-in + custom roles, each a set of permission patterns (possibly `{scope}`-templated).
Assignment store	Policy store	`(principalType, principalId, roleId, scope, expiresAt)` rows.
Permission Resolver	PDP (decision point)	Pure function: `authorize(principal, dom, type, act)` via scope-walk + wildcard match. **Invariant to vocabulary.**
Route/Tool guard	PEP (enforcement point)	Wraps HTTP routes and **AI-agent tool-calls**; calls the resolver before side effects.
Resource Bridge	Delegated grants	Time-boxed, audited cross-domain shares expressed in the same tuple model.
Audit log	—	Append-only record of grants, shares, and (optionally) decisions.

6.2 Architecture diagram



6.3 Cross-cutting properties

- **Vocabulary invariance (R2).** The PDP imports no domain-specific symbol. It reads the catalogue as data. New vocabulary is new *rows*, not new *code*.

- **Pure, side-effect-free resolution.** `authorize()` is a pure function of (assignments, roles, request); easy to test exhaustively and to cache.
- **Uniform principal handling (R7).** Users, groups, tokens, and personas differ only by `principalType`; the PDP treats them identically.
- **Fail-closed default (R11).** Absence of a matching ALLOW is DENY.
- **Auditability (R10).** Grants and shares are recorded with who/what/where/when/why.

---

7. Detailed Mechanics

7.1 Boot-time provider registration

Each domain module declares its resources **declaratively** in its module contract:

```
// domains/crm/mod.js  (illustrative)
export default {
  id: 'crm',
  resources: {
    leads:    { actions: ['read','write','delete','export','assign'],
                schema: { id:'bigint', name:'string', email:'string', status:'string' },
                searchable: true,  shareable: true },
    deals:    { actions: ['read','write','delete','close','forecast'],
                searchable: true,  shareable: true },
    contacts: { actions: ['read','write','delete','merge'],
                searchable: true,  shareable: true },
    tickets:  { actions: ['read','write','delete','close','escalate'],
                searchable: true,  shareable: false }, // internal only
  },
};
```

At boot the platform iterates loaded domains and **upserts** each declaration into the registry. The OS itself registers a fixed set of platform resources (`platform.users`, `platform.roles`, `platform.config`, …) through the **same** path — the OS is just another provider.

```
registerProviders(domains):
  for d in domains:
    for (type, decl) in d.resources:
      registry.upsert(domain=d.id, type=type,
                      actions=decl.actions, schema=decl.schema,
                      searchable=decl.searchable, shareable=decl.shareable)
```

Invariant (R2): `registerProviders` writes only registry rows. It does not import or branch on any specific domain or action name.

7.2 Permission tuples and three-axis wildcard matching

A permission pattern is `dom:type:act`, each axis either a literal or `*`:

```
crm:leads:read     crm:leads:*     crm:*:read     crm:*:*     *:*:read     *:*:*
```

A requested tuple `q = (qd, qt, qa)` is **covered** by a pattern `P = (pd, pt, pa)` iff each axis matches:

```
axisMatch(p, q) := (p == '*') or (p == q)
covers(P, q)    := axisMatch(pd,qd) and axisMatch(pt,qt) and axisMatch(pa,qa)
```

This is **per-axis** wildcarding: `crm:*:read` covers `crm:leads:read` and `crm:deals:read` but not `crm:leads:write`. `*:*:*` is god-mode.

7.3 Path-scoped assignments and root-ward inheritance

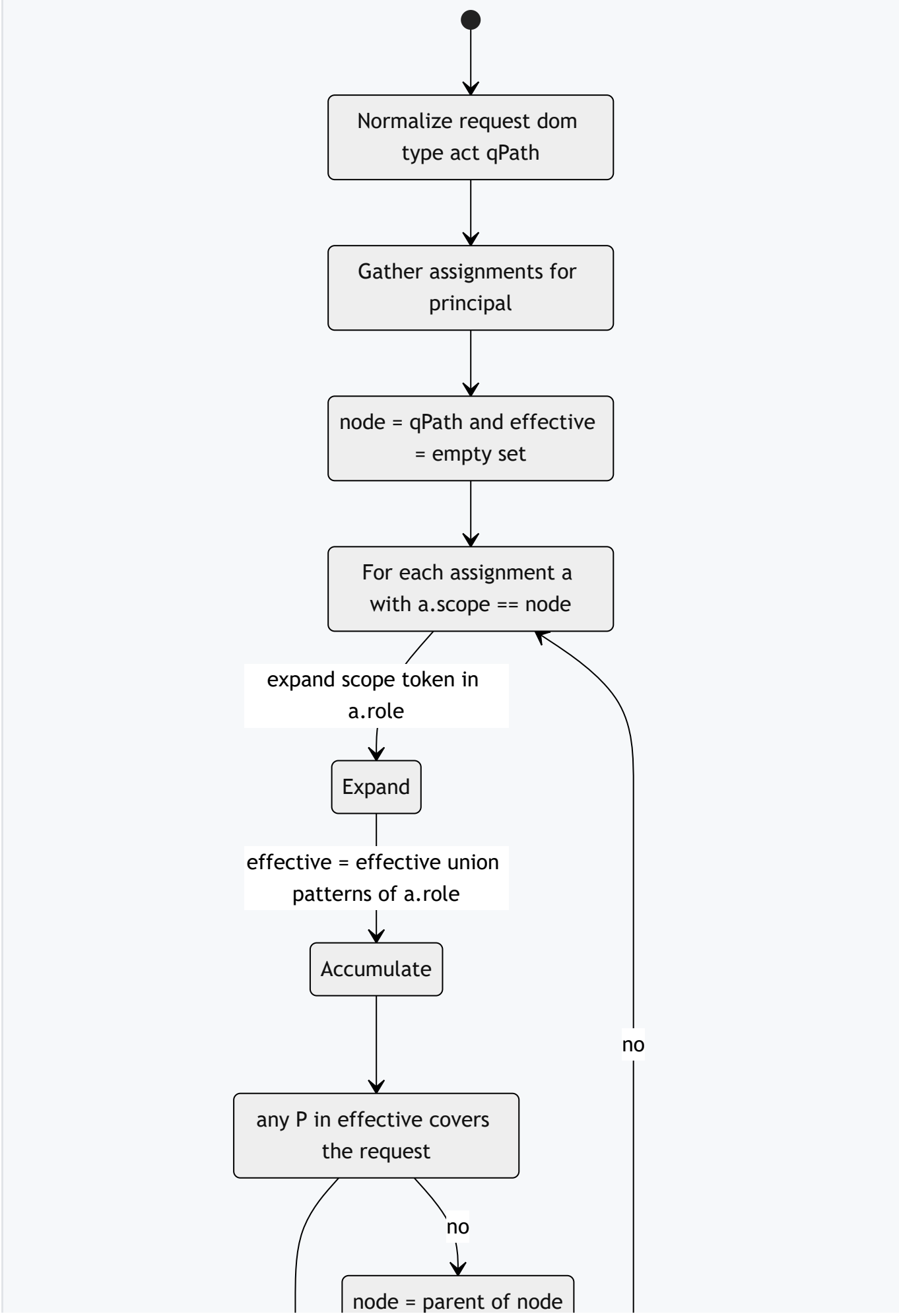
Scopes form a path tree rooted at `/`:

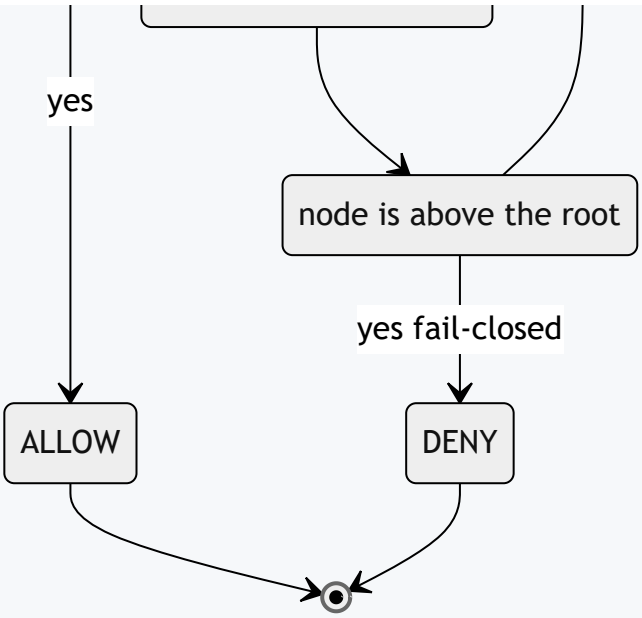
```
/  →  /crm  →  /crm/leads
/  →  /finance
/  →  /platform  →  /platform/users
```

An assignment at `/crm` **inherits downward** to `/crm/leads`, `/crm/deals`, etc. To decide a request at scope path `qPath`, the resolver walks **from `qPath` toward the root**, collecting the pattern sets of every assignment whose scope is an ancestor-or-self of `qPath`:

```
ancestorsInclusive('/crm/leads') = ['/crm/leads', '/crm', '/']
```

The union of patterns from all assignments at those scopes is the effective pattern set for the request.


Normalize request dom type act qPath
Gather assignments for principal
node = qPath and effective = empty set
For each assignment a with a.scope == node
expand scope token in a.role
Expand
effective = effective union patterns of a.role
Accumulate
any P in effective covers the request
no
node = parent of node
no



(The walk may terminate **early** at the first covering pattern — an optimization — or accumulate the full set for explainability; both yield the same decision.)

7.4 `{scope}`-templated roles

A built-in role can be authored once and bound to whichever domain an assignment names, via a `{scope}` placeholder on the `dom` axis:

```
role 'contributor' := { '{scope}:*:read', '{scope}:*:write' }
role 'reader'      := { '{scope}:*:read' }
role 'domain-admin':= { '{scope}:*:*' }
```

When resolving an assignment (`…, 'contributor', '/crm'`), the resolver derives the **scope token** from the scope path (`/crm → crm`) and substitutes it:

```
expand('{scope}:*:read', scope='/crm') = 'crm:*:read'
```

Thus **one** `contributor` role definition governs CRM, finance, comms, and any future domain — satisfying R4 and R8 with zero per-domain role authoring. Custom roles (e.g. `sales-manager := {crm:*:*, finance:invoices:read, comms:emails:write}`) use literal axes and are stored as data.

7.5 The resolver-invariant registration contract (characterizing feature)

The PDP's `authorize()`:

1. reads **assignments** for the principal (data),
2. reads **role pattern sets** (data),
3. reads, for discovery/validation only, the **catalogue** (data),
4. performs scope-walk + `{scope}` expansion + per-axis matching.

It contains **no** reference to any specific domain, resource type, or action. The proof obligation for R2 is mechanical: *a new domain's `mod.js` changes only registry rows; `authorize()` and the schema are byte-*

identical before and after. This is the contract that distinguishes the disclosure from a hardcoded capability map (where new vocabulary edits the resolver) and from per-domain ACLs (where the matching logic is duplicated).

7.6 Typed principals (users, groups, tokens, AI personas)

Principals are `(principalType, principalId)`:

Type	Example id	Notes
`user`	`user:alice`	Human; assignments may also come via group.
`group`	`group:sales-team`	Membership expands to a user's effective assignments.
`token`	`token:tok_xxxx`	API key; may carry explicit `customPermissions` in addition to its role.
`persona`	`persona:assistant-sales`	**AI agent**; assigned scoped roles like a user. Its tool-calls pass the same PEP→PDP path.

The persona type is the AI-platform-specific element: an autonomous agent that reads CRM and drafts comms is governed by a scoped `reader`/`contributor` assignment, and **every tool it invokes is enforced by the same resolver** — no separate agent ACL.

7.7 Cross-domain Resource Bridge (time-boxed shares)

A domain may grant another domain (or principal) a **time-boxed** permission over a specific resource, expressed in the same tuple model and recorded for audit:

```
share(resourceAddress='crm.leads/123', sharedWith='finance',
      permissions=['read'], reason='Invoice generation',
      expiresAt='2026-07-01T00:00:00Z')
```

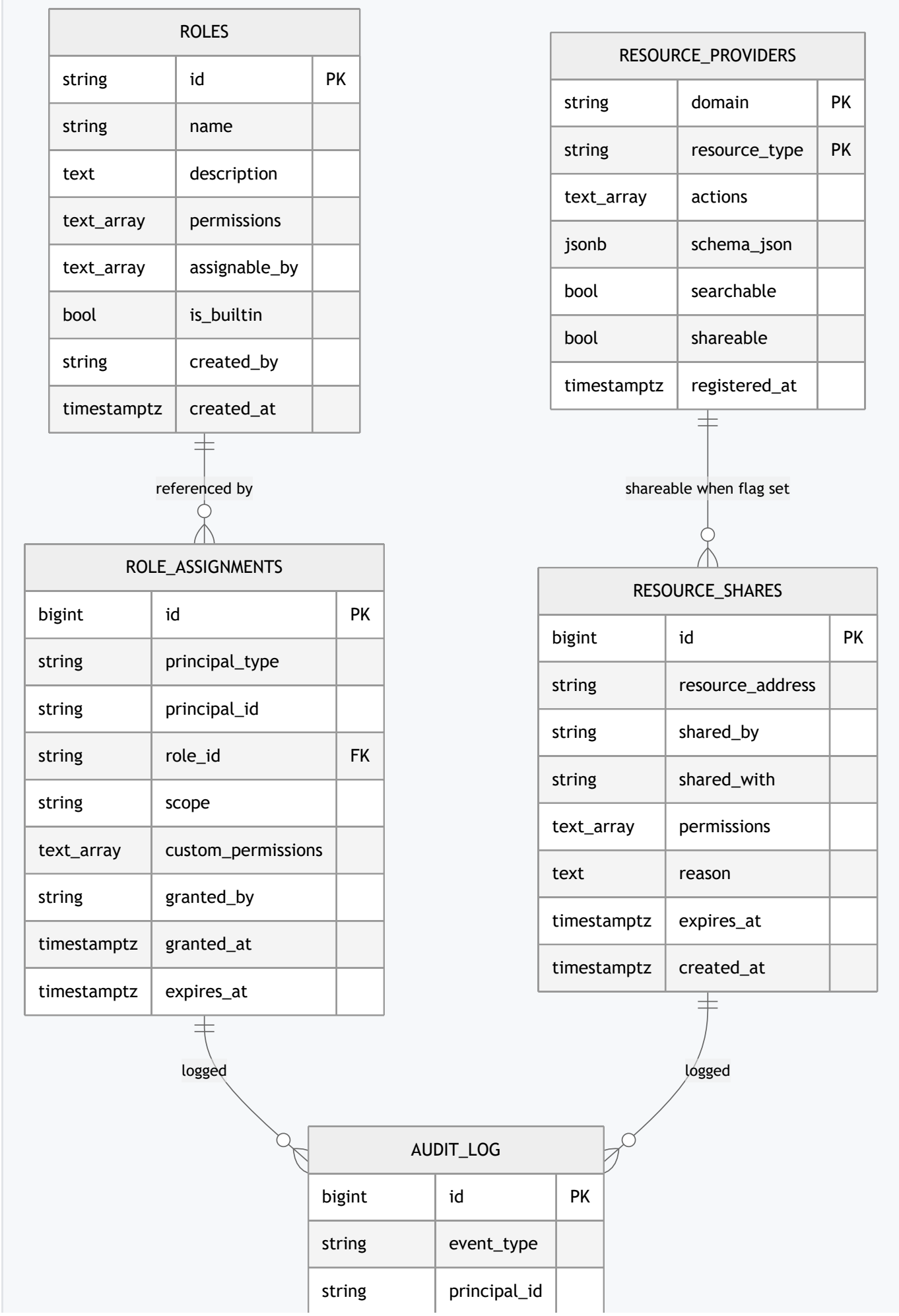
During resolution, an unexpired share that covers the request contributes its permission just like an assignment pattern. Shares satisfy R9/R12 and are strictly *additive* and *expiring* — they cannot broaden beyond the `shareable` flag the source domain declared at boot (a `shareable:false` resource type, e.g. `crm.tickets`, can never be bridged).

7.8 Error handling, configuration, and audit

- **Unknown vocabulary.** A request naming a `(dom,type,act)` not in the catalogue is *still* resolved (it simply will not be covered unless a wildcard pattern matches), but the guard may additionally **reject** unknown tuples as a hardening option (configurable; default: resolve normally, deny if uncovered).
- **Malformed pattern / scope.** Rejected at write time (role/assignment creation), not at decision time.
- **Expiry.** Expired assignments and shares are excluded from the union.
- **Configuration points.** Default-deny vs. reject-unknown; whether decisions (not just grants) are audited; cache TTL for assignment/role reads; the OS resource set.
- **Audit.** Grants, shares, role edits, and (optionally) every decision are written to an append-only log with `who/what/where/when/why`.

---

8. Data Model


ROLES
string id PK
string name
text description
text_array permissions
text_array assignable_by
bool is_builtin
string created_by
timestamptz created_at
RESOURCE_PROVIDERS
string domain PK
string resource_type PK
text_array actions
jsonb schema_json
bool searchable
bool shareable
timestamptz registered_at
referenced by
shareable when flag set
ROLE_ASSIGNMENTS
bigint id PK
string principal_type
string principal_id
string role_id FK
string scope
text_array custom_permissions
string granted_by
timestamptz granted_at
timestamptz expires_at
RESOURCE_SHARES
bigint id PK
string resource_address
string shared_by
string shared_with
text_array permissions
text reason
timestamptz expires_at
timestamptz created_at
logged
logged
AUDIT_LOG
bigint id PK
string event_type
string principal_id

string	target	
jsonb	detail	
timestamptz	at	

Field notes.

- `resource_providers` PK is `(domain, resource_type)`; `upsert` on boot makes registration idempotent across restarts (R1/R2).
- `roles.permissions` holds pattern strings, possibly `{scope}`-templated.
- `role_assignments` UNIQUE on `(principal_type, principal_id, role_id, scope)` prevents duplicate grants; `expires_at` gives R12.
- `resource_shares.permissions` are action literals; resolution gates them by the source type's `shareable` flag.
- Indices: `role_assignments(principal_type, principal_id)` and `role_assignments(scope)` make both the principal-gather and scope-walk cheap.

The DDL is reproduced in `src/schema.sql`.

---

9. Reference Implementation & Enablement

9.1 What `src/` demonstrates

`src/rbac.js` is a **clean-room, dependency-free Node.js** implementation of the full model: a registry, a role store with `{scope}` templates, an assignment store with expiry, the per-axis wildcard matcher, and the **root-ward scope-walk resolver**. `src/example.js` boots three domains, registers their providers, defines built-in and custom roles, makes scoped assignments (including an **AI persona** and an **API token**), and asserts a battery of ALLOW/DENY outcomes — including a cross-domain **share**. `src/schema.sql` gives the relational form.

9.2 How it reduces the invention to practice

The reference code exhibits, end to end and runnably:

- **R1/R2** — `registerProvider()` only writes catalogue entries; the resolver imports no domain symbol. A self-check asserts that registering a brand-new domain (`projects`) and immediately authorizing against it works **without any resolver edit**.
- **R3** — `covers()` implements three-axis wildcard matching.
- **R4/R8** — `expandScope()` substitutes `{scope}`; one `contributor` role serves multiple domains.
- **R5/R6** — `authorize()` walks `ancestorsInclusive(path)` root-ward, unioning patterns.
- **R7** — principals are `type:id`; a persona and a token are asserted alongside users.
- **R9/R12** — `share()` adds a time-boxed grant; an expired share is excluded.
- **R11** — uncovered ⇒ `DENY`.

9.3 Configuration points exposed

`createEngine({ rejectUnknownTuples, auditDecisions, now })` — the `now` injector makes expiry deterministically testable; `rejectUnknownTuples` toggles the hardening posture of §7.8; `auditDecisions` toggles per-decision logging.

This is **illustrative, not production** code: no persistence, no auth, no concurrency control — the persistence shape is in `schema.sql`.

---

10. Worked Example / Scenario

Setup. Domains `crm`, `comms`, `finance` register at boot. Roles: `contributor := {{scope}:*:read, {scope}:*:write}`, `reader := {{scope}:*:read}`, custom `sales-manager := {crm:*:*, finance:invoices:read, comms:emails:write}`.

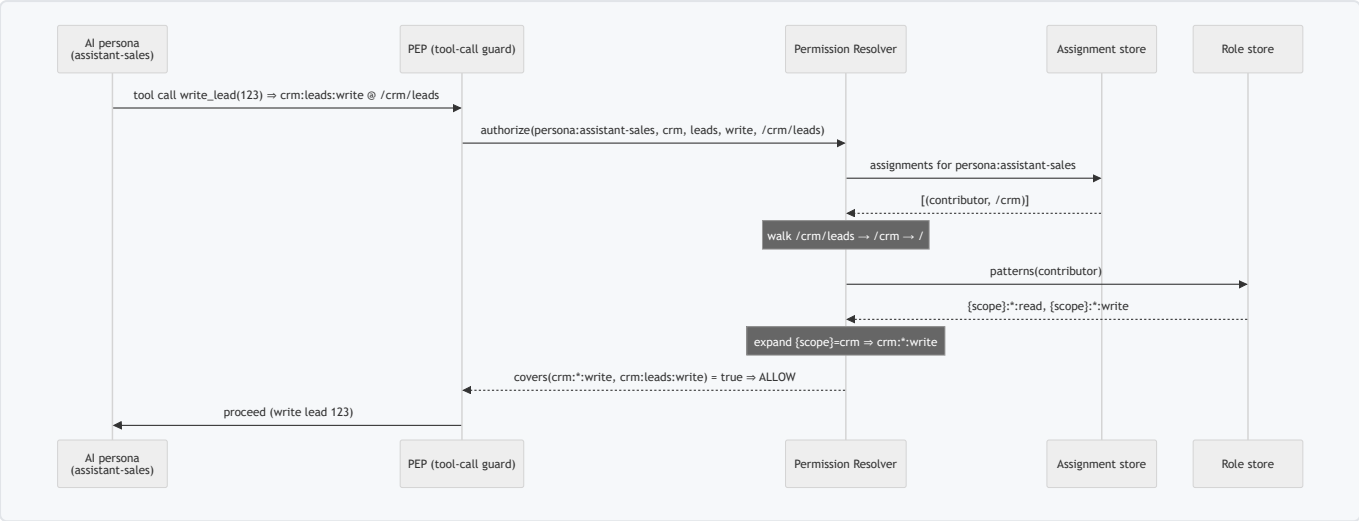
Assignments.

Principal	Role	Scope
`user:alice`	`sales-manager`	`/`
`user:bob`	`reader`	`/finance`
`persona:assistant-sales`	`contributor`	`/crm`
`token:tok_ro`	`reader`	`/crm/leads`

Questions resolved.

#	Request	Decision	Why
1	alice: `crm:deals:delete` @ `/crm/deals`	**ALLOW**	`sales-manager` at `/` covers via `crm:*:*`.
2	bob: `finance:invoices:read` @ `/finance`	**ALLOW**	`reader@/finance` expands to `finance:*:read`.
3	bob: `finance:invoices:write` @ `/finance`	**DENY**	reader has no write; uncovered ⇒ fail-closed.
4	persona: `crm:leads:write` @ `/crm/leads`	**ALLOW**	`contributor@/crm` → `crm:*:write`, inherited downward.
5	persona: `crm:leads:delete` @ `/crm/leads`	**DENY**	contributor lacks delete.
6	token: `crm:deals:read` @ `/crm/deals`	**DENY**	token scoped to `/crm/leads`, not an ancestor of `/crm/deals`.
7	finance (via **share**): `crm:leads:read` on `crm.leads/123`	**ALLOW**	time-boxed share grants `read`; `leads` is `shareable`.
8	finance (via share): `crm:tickets:read`	**DENY**	`tickets` declared `shareable:false`; bridge refuses.

Sequence (Question 4 — the AI persona path):



11. Security, Safety & Failure Modes

11.1 STRIDE-style threat table

Threat	Vector	Mitigation in the model
Spoofing	A persona/token impersonating a user.	Principals are **typed**; assignments bind to `(type,id)`; identity is established upstream (out of scope) and the type cannot be forged within the resolver.
Tampering	Editing roles/assignments to escalate.	Role/assignment edits require `platform:roles:write` / `:assign`, themselves resolved by the model; all edits audited.
Repudiation	"I never granted that."	Append-only audit of grants, shares, role edits with `granted_by/reason`.
Information disclosure	Over-broad `*:*:read`.	Wildcards are explicit and discoverable; `$metadata` lets auditors enumerate exposure; `shareable:false` blocks bridging of sensitive types.
Denial of service	Pathologically deep scope trees.	Scope depth is bounded by the (small) domain hierarchy; resolution is O(depth × assignments).
Elevation of privilege	Stale/expired grant still honored.	`expires_at` excludes expired assignments and shares at decision time.

11.2 Fail-open vs. fail-closed posture

The resolver is **fail-closed**: any request not covered by a live ALLOW pattern is DENY (R11). There is no implicit allow, no "owner bypass" inside the resolver, and unknown vocabulary defaults to DENY (and, optionally, to outright rejection). The *only* deliberately permissive surface is god-mode `*:*:*`, which is a literal pattern that must be explicitly assigned and is itself audited.

11.3 Additional failure modes

Mode	Effect	Handling
Registry empty at first request (boot race)	No catalogue ⇒ discovery degraded, but resolution still fail-closed.	Guard resolution gates on boot-complete; pre-boot requests DENY.
Conflicting patterns (allow vs. a hypothetical deny)	This model is **allow-only** (no explicit deny rules) ⇒ no conflict-resolution ambiguity.	By design; simplicity is a safety property.
Group cycle (group A in group B in A)	Infinite membership expansion.	Membership expansion is cycle-detected at write time.

12. Standards & Framework Mapping

We claim **semantic alignment**, not certification, with the following.

Framework	Alignment
NIST RBAC (ANSI INCITS 359-2004)	Core RBAC (users–roles–permissions) and hierarchical RBAC (scope inheritance approximates role hierarchy); least-privilege via narrow patterns.
OASIS XACML 3.0	Registry = PIP; resolver = PDP; route/tool guard = PEP; allow-only combining ≈ "permit-overrides" with no deny rules.
NIST SP 800-162 (ABAC)	Scope path and principal type are attributes feeding the decision; not full ABAC, but attribute-aware.
NIST SP 800-53 rev.5	AC-2 (account mgmt via assignments), AC-3 (access enforcement at the PEP), AC-6 (least privilege), AU-2 (auditable events).
Azure RM / AWS IAM / K8s RBAC	Structural ancestors for `provider/type/action` + wildcards + scope; this is an app-domain, boot-registered adaptation.
OAuth 2.0 (RFC 6749) scopes	Token principal carries `customPermissions`, mapping OAuth scopes into the tuple model.

These mappings are **descriptive**; this disclosure does not assert compliance certification with any of them.

13. Evaluation Methodology

Numbers below are **illustrative** placeholders to define *how* one would measure, not measured results.

Dimension	Metric	Method	Illustrative target
Resolver invariance (R2)	# resolver/schema lines changed when adding a domain	Diff resolver+schema before/after adding `projects` domain	**0** (hard requirement)
Onboarding cost	Time/LOC to expose a new resource type	Count LOC in domain `mod.js` only	~5–10 LOC, 0 deploys to auth core
Decision latency	p50/p99 µs per `authorize()`	Microbench over N assignments × depth D	p99 < 50 µs (in-memory)
Decision complexity	Big-O	Analysis	O(D × A) per request (D = scope depth, A = principal assignments)
Wildcard correctness	% of a truth-table covered	Exhaustive enumeration of axis combinations	100%
Audit completeness	% of grants/shares with full who/what/where/when/why	Schema NOT NULL + log scan	100%
Fail-closed coverage	% of uncovered requests denied	Property test: random uncovered tuples ⇒ DENY	100%

Interpretation. R2 invariance and fail-closed coverage are *correctness* properties (must be 100% / 0). Latency and onboarding cost are *efficiency* properties where the model's value over hardcoded maps is the **elimination of an auth-core deploy per permission**.

---

14. Novelty & Inventive Claims

> These claims are written in patent-claim style to delineate the disclosed subject matter for prior-art purposes. **No patent is sought**; this is a defensive publication placing the subject matter in the public domain.

Claim 1 (independent)

A method for access control in a multi-domain platform comprising: receiving, from each of a plurality of domain modules during a boot phase, a registration of a set of resource types each having an associated set of actions and stored in a resource catalogue; representing each permission as a tuple of a domain identifier, a resource type, and an action; receiving role definitions each consisting of a set of permission patterns in which one or more of the domain identifier, resource type, and action may be a wildcard; receiving role assignments each binding a principal to a role at a scope path within a hierarchy of resources; and resolving an authorization decision for a requested tuple by traversing said hierarchy from the queried scope path toward a root, unioning the permission patterns of all role assignments encountered along the traversal, and determining whether any unioned pattern covers the requested tuple by per-axis matching; characterized in that registration of a resource type and its actions occurs without modification to the resolver that performs said traversal and matching, to the role definitions, or to the storage schema.

Dependent claims

2. The method of claim 1, wherein each said registration is **declarative** and stored idempotently keyed by (domain identifier, resource type), such that re-registration on a subsequent boot does not alter resolver behavior.

3. The method of claim 1, wherein a permission pattern supports a wildcard **independently on each of** the domain, resource type, and action axes, and covering requires every axis of the pattern to either equal the corresponding axis of the requested tuple or be the wildcard.

4. The method of claim 1, wherein at least one role definition contains a **scope placeholder** on its domain axis, and resolving substitutes, for said placeholder, a domain token derived from the scope path of the role assignment under evaluation, whereby a single role definition applies to multiple domains.

5. The method of claim 4, wherein the same scope-placeholder role definition applies to a domain whose resource types were registered **after** the role definition was authored.

6. The method of claim 1, wherein principals are **typed**, the types including at least a human user, a group, an API token, and an **autonomous AI persona**, and the resolver applies the identical traversal and matching to every type.

7. The method of claim 6, wherein a tool invocation by said AI persona is gated by the **same resolver** used for human-originated requests, prior to any side effect of the tool invocation.

8. The method of claim 1, wherein a role assignment carries an **expiry** time and the resolver excludes assignments whose expiry has passed from the union.

9. The method of claim 1, further comprising a **resource-sharing grant** that binds a target principal or domain to one or more actions over a specific resource address for a bounded time interval, and wherein the resolver treats an unexpired such grant as an additional source of permission for the resolution.

10. The method of claim 9, wherein each registered resource type carries a **shareability flag**, and the sharing grant is refused for any resource type whose flag denies sharing.

11. The method of claim 1, wherein an authorization request whose requested tuple is not covered by any unioned pattern is **denied by default (fail-closed)**.

12. The method of claim 1, further comprising a **discovery interface** that returns the resource catalogue, including for each registered resource type its actions and schema, assembled from the boot-phase registrations of the domain modules.

13. The method of claim 1, wherein the resolver, the resource catalogue, and the role and assignment stores together implement a separation in which the catalogue serves as an information point, the resolver as a decision point, and a route or tool guard as an enforcement point.

14. The method of claim 1, wherein an operating-system layer registers a set of platform resource types through the **same boot-phase registration path** used by the domain modules, such that platform and domain resources share one catalogue and one resolver.

15. The method of claim 1, wherein the traversal **terminates early** upon encountering the first unioned pattern that covers the requested tuple, and wherein early termination yields a decision identical to full accumulation.

16. The method of claim 1, further comprising recording, in an **append-only audit log**, each role assignment, each sharing grant, and optionally each authorization decision, with an actor, a target, a scope, a timestamp, and a reason.

17. The method of claim 1, wherein an API-token principal additionally carries an explicit set of permission patterns that are **unioned** with those of its assigned role during resolution, narrowing or extending its effective permissions independently of the role.

---

15. Limitations & Threats to Validity

- **The ingredients are individually known.** Tuples, wildcards, and scope inheritance exist in Azure RM, AWS IAM, and Kubernetes RBAC. The disclosed point is the *combination*: boot-time, domain-owned, resolver-invariant registration with AI-persona principals. We do **not** claim novelty of any ingredient alone.
- **Allow-only model.** There are no explicit deny rules; deny is the default. This is a deliberate simplicity/safety choice, but it cannot express "allow everywhere except here" without restructuring scopes. ABAC/XACML are more expressive at the cost of complexity.
- **Identity is out of scope.** Authentication, token issuance, and group membership sourcing are assumed; only authorization is disclosed.
- **Scale of patterns.** Very large custom-role pattern sets degrade decision latency linearly; caching the union per (principal, scope) is the mitigation but is not itself novel.
- **Prior art is genuinely close.** Azure RM RBAC is the nearest ancestor; a skeptical reader should weigh §5's delta table carefully. We publish defensively *because* the §103-style obviousness risk over these ancestors is real.
- **Intentionally withheld.** Production concerns — persistence, concurrency, cache invalidation, the identity layer — are out of scope and not enabling-critical to the disclosed method.

---

16. Future Work & Open-Source Reference App

Planned: a self-contained **Resource-Provider RBAC Sandbox** open-source app — registry, resolver, admin UI, `$metadata` discovery, audit viewer — packaged as a container and deployable to a generic Kubernetes/AKS cluster via plain `kubectl`/Helm-style manifests. It will let a reader register fictitious domains, author roles, make scoped assignments (including a mock AI persona), and watch ALLOW/DENY decisions with a full explanation trace. Roadmap and deployment sketch: docs/OPEN-SOURCE-APP.md. Further work: a decision-cache with correctness-preserving invalidation; an optional explicit-deny layer with a combining algorithm; a formal model-check of fail-closed coverage.

---

17. Conclusion

Flat RBAC fails the multi-domain AI platform on granularity, growth, and non-human principals. The resource-provider model resolves all three by **inverting ownership of the permission vocabulary**: domains register their own resource types and actions at boot into a catalogue that the permission resolver *reads but never embeds*. Permissions become `domain:resourceType:action` tuples matched with per-axis wildcards; assignments bind typed principals — including AI personas — to roles at scope paths; decisions resolve by a root-ward scope walk. The characterizing property — that new vocabulary never touches the resolver, schema, or roles — yields custom roles and cross-domain sharing with zero auth-core deploys. This document, with its enabling reference implementation, is published as dated prior art so the technique remains free for all to practice.

Appendix A — Prior-Art Landscape (well-trodden vs. candidate-novel)

Well-trodden (treat as known):

- `provider/type/action` action strings with `*` wildcards (Azure RM, AWS IAM).
- Scope inheritance over a resource tree (Azure management-group→resource).
- Grouped resource verbs and namespaced scope (Kubernetes RBAC).
- Uniform authorization tuples consulted by all services (Google Zanzibar).
- PDP/PEP/PIP separation and ABAC (OASIS XACML, NIST SP 800-162).
- Token-scoped permissions (OAuth 2.0).
- Modules registering capabilities at activation (OSGi Declarative Services).

Candidate-novel (the disclosed combination):

- **Boot-time, application-domain-owned** registration of `{resourceType, actions, schema}` into a catalogue the resolver is **contractually invariant** to.
- The same model treating **AI personas** as scoped principals whose **tool-calls** pass the identical resolver.
- `{scope}`-templated roles spanning heterogeneous business domains, applying to resource types registered after the role was authored.
- A cross-domain, time-boxed, shareability-flag-gated **resource bridge** in the same tuple model.

Honesty attestation. The prior-art review here is **directional, not exhaustive**. No formal freedom-to-operate or novelty search was commissioned for this publication. The patentability screen rated this subject matter US ≈ 55/100 and DE/EPO ≈ 42/100, explicitly citing heavy prior art (Azure RBAC, AWS IAM, Kubernetes RBAC) and recommending defensive publication — which is the posture of this document. Readers should not treat any statement here as a legal opinion.

Appendix B — Glossary

Term	Meaning
Domain module	An independently-developed application unit (CRM, comms, finance) loaded at boot; acts as a resource provider.
Resource provider	An entity that registers resource types and their actions into the catalogue.
Resource type	A class of governed object (e.g. `leads`) with a set of actions and a schema.
Permission tuple	`domain:resourceType:action`.
Pattern	A permission tuple in which axes may be *.
Role	A named set of patterns, possibly `{scope}`-templated.
Assignment	A binding of a typed principal to a role at a scope path, optionally expiring.
Scope path	A node in the resource hierarchy (`/`, `/crm`, `/crm/leads`).
Principal	`(type, id)`; type ∈ {user, group, token, persona}.
Resolver (PDP)	The vocabulary-invariant decision function.
Guard (PEP)	The enforcement point wrapping routes and tool-calls.
Registry (PIP)	The resource catalogue.
Resource Bridge	Time-boxed, audited cross-domain shares.
Resolver-invariant contract	The property that new vocabulary never modifies the resolver/schema/roles.

Appendix C — Reference-Implementation Index

File	Purpose
`src/rbac.js`	Clean-room engine: registry, role store, assignment store, `{scope}` expansion, per-axis matcher, root-ward scope-walk resolver, shares.
`src/example.js`	Boots 3 domains, defines roles, makes scoped assignments (user, persona, token), asserts ALLOW/DENY incl. a share and a post-hoc new domain.
`src/schema.sql`	Relational form of the data model (§8).
`src/README.md`	How to run; what it demonstrates; clean-room/illustrative disclaimer.

Appendix D — Defensive-Publication Deposit & Timestamp

- **Publication date:** 2026-06-25.
- **Publisher:** Gus IT LLC (Florida, USA). **Author:** Gustavo Assuncao, PhD.
- **Version:** 1.0. **License:** AGPL-3.0-or-later (copyleft) plus a commercial license available from Gus IT LLC (see `COMMERCIAL-LICENSE.md`). AGPL-3.0, via GPLv3 section 11, includes an express patent license.
- **Deposit channel:** to be assigned (IP.com / Zenodo / arXiv) — establishes a public, dated, citable prior-art record. No DOI is asserted at time of writing.
- **Statement of intent:** This disclosure is **intentionally public to bar later patenting by others**. By publishing a complete, enabling description with a working reference implementation,

the publisher places the described method in the public domain as prior art so that it remains freely practiceable.
