https://doi.org/10.15407/intchsys.2026.03.102
UDC 004.056.53

A.V. MELNYCHENKO, PhD (Engineering), Assistant Lecturer,
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute",
37, Beresteiskyi ave., Kyiv, 03056, Ukraine
https://orcid.org/0009-0000-3588-4772
artemxl@gmail.com

O.V. SHALDENKO, PhD (Engineering), Associate Professor,
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute",
37, Beresteiskyi ave., Kyiv, 03056, Ukraine
https://orcid.org/0000-0001-6730-965X
o.shaldenko@gmail.com

TWO-POINT AUTHORIZATION ENGINE WITH REVOCABLE SHARING FOR MULTITENANT ENVIRONMENTS

Multi-tenant software-as-a-service (SaaS) platforms that contain tenant-partitioned data require authorization systems which cover scenarios where classic flat role-based access control (RBAC) fails. Specifically, these systems need to provide access to individual resources rather than whole sets of resources, controlled cross-tenant sharing, and account-less share links which can be revoked at any time. We present access-kit, an in-process authorization engine for the .NET platform built on an explicit model: principals, "area:verb" actions, composable deny-wins policy statements with inheritance, and a resource hierarchy. Enforcement is a co-designed pair of mechanisms which consist of an application-pipeline gate that rejects unauthorized actions outright and an object-relational mapper (ORM) row filter that filters the data to only include rows which are permitted to the current principal. The gate publishes per-request scope which is consumed by row filter, and this scope keeps the two mechanisms in agreement and therefore keeping error reporting consistent. As a result, a write on a readable-but-not-writable resource is refused as a forbidden action rather than disguised as a missing resource, preventing the existence of the resource from being leaked. In the developed framework, each request acts in a single active tenant, so access resolves to one flat statement set. Cross-tenant access is achieved by pulling the grants the active tenant owns, and a caller switches workspace (re-minting its token) to act in another permitted tenant. Account-less sharing is implemented by issuing a revocable, table-backed capability token. We give an

Cite: Melnychenko A.V., Shaldenko O.V. Two Point Authorization Engine with Revocable Sharing for Multitenant Environments. *Information Technologies and Systems*. 3 (9). 2026. 102—102. https://doi.org/10.15407/intechsys.2026.03.102

analytic cost model and an evaluation measured on PostgreSQL: a batch chain-walk resolves effective access in a number of database round trips that is independent of how many grants a principal holds, and we compare it with a naive baseline and a single-query recursive-CTE alternative. The per-request cost is dominated by this resolution, while the in-memory gate adds negligible overhead; the cost model and measurements are given in Section 5. The engine is implemented as a generalized reference implementation with adapters for Entity Framework Core, MediatR, and ASP.NET Core.

Keywords: *access control; authorization; multi-tenancy; resource scoping; cross-tenant sharing; access tokens; row-level security; .NET.*

Introduction

Any system that allows more than one part to act on shared data must have a mechanism to decide who may do what actions on which data — the problem of access control. It is getting increasingly important where data is partitioned among many independent parties — for example, a multi-tenant software-as-a-service (SaaS) platform, which is one of the most common ways of delivering software to customers. SaaS serves many customer organizations (tenants) from one deployment and must isolate each tenant's data from the others [1], yet the same structure arises in any application that confines users to their own slice of a shared store. The dominant authorization model for such systems is role-based access control (RBAC) [2, 3]: a user is assigned a role, and the role carries a fixed set of permissions. RBAC helps to resolve the problem of "what kind of user is this?" by assigning a role to a user, and the role carries a fixed set of permissions. However, RBAC is not sufficient to address the problem of access control in multi-tenant environments.

Collaborative, data-centric platforms such as research data collection, analytics workspaces, document management often have authorization requirements that classic RBAC cannot express. For example:

- Partial, resource-scoped access. "This user may edit given N projects and read everything else inside tenant." In this case, the unit of authorization is an individual resource, not a role.
- Cross-tenant sharing. "Tenant B may manage one project that belongs to tenant A," without users of B becoming members of A or A's isolation weakening for anything else.
- Account-less sharing. "Anyone with given link may view the dashboard". The link should be revocable at any moment, with no account provisioned for the viewer.

These requirements also share a structural subtlety. While executing a single API operation, the code generally performs two actions: it performs an **action** (it reads, updates, or deletes), and it operates over a **set of data rows** (the resources the request reads or targets). Authorizing the request therefore has two distinct facets that must stay consistent: whether the principal may perform the action at all, and which rows the request is permitted to see or modify. In practice these facets are enforced by different mechanisms: a permission check inside the handler and a data-scoping

filter over the query. The issue is that these two mechanisms can drift apart and when they do, the system either leaks data (a row is returned that the action was not authorized to touch) or behaves confusingly (a write reaches a row the read path cannot see and returns a misleading response). Bolting partial, cross-tenant, and account-less access onto role checks produces scattered ad-hoc conditionals that compare the caller's role and are impossible to audit and easy to get subtly wrong - the class of defect catalogued by OWASP as Broken Object Level Authorization [4] and, historically, as the confused deputy [5].

This article presents access-kit, an in-process authorization engine that addresses these requirements. The engine has been extracted and generalized from a production multi-tenant research platform into a reusable. NET library. We describe its model, its co-designed two-point enforcement, its cross-tenant sharing and revocable capability tokens, and we evaluate it analytically and experimentally.

Analysis of Recent Research and Publications

RBAC and ABAC. RBAC [2, 3] groups permissions under roles. It does not natively express per-resource grants or sharing across administrative boundaries. Attribute-based access control (ABAC) [6] decides access from attributes of the subject, object, and environment, and is expressive enough to encode resource scoping in principle, but a general ABAC policy engine passes the row scoping responsibility to the query author and provides no built-in tenant isolation.

Relationship-based access control (ReBAC) and Zanzibar. Google's Zanzibar [7] models authorization as relationship tuples evaluated by a globally distributed service, and OpenFGA [8] is a widely used open-source implementation of the same idea. These systems are powerful and excel at deeply relational permission graphs, but they are out-of-process authorization services. Each decision is an API call, and row scoping is solved by reverse-expansion APIs that the application must wire into its queries separately from its database.

Policy libraries. Casbin [9] and Oso [10] are in-process policy engines (an enforcer over a model/DSL, and the Polar logic language, respectively). They decide whether an action is allowed but, like ABAC engines, leave row scoping and tenant isolation as application responsibilities. Auth0 FGA [11] is a managed Zanzibar-style service. Cloud IAM systems such as AWS IAM [12] combine RBAC and ABAC for control-plane resources but are not designed to be embedded as the data-plane authorization of a third-party SaaS application.

Policy languages: Cedar and OPA. The two closest in-process alternatives are AWS Cedar [13] and the Open Policy Agent (OPA) [14]. Cedar is an embeddable policy language with a formally verified evaluator and analysis tooling. OPA evaluates Rego policies and runs as a sidecar or embedded library. Both are decision engines: given a request and a policy

set they return permit/deny. Like Casbin and Oso they do not scope a database query, and they are typically deployed as a separate policy-decision point, so they carry the out-of-process trust boundary and per-check hop discussed in Section "System and threat model". Cedar's formal verification is an advantage our engine does not claim. The distinction is that access-kit co-designs the decision with an ORM row filter in the same process, trading that isolation for row scoping without an extra hop and automatic data scoping.

Capabilities and revocable tokens. Account-less sharing is a capability-security problem [15] where the link is the authority. Bearer capabilities are convenient but historically hard to revoke. Macaroons [16] add caveats and contextual confinement but remain self-contained bearer tokens that require an external check for revocation. Database row-level security [17] and ORM global query filters [18] provide the row scoping mechanism but not the policy model above them.

Against these systems, access-kit occupies a specific niche. It is in-process and co-designs the decision gate with an ORM row filter, so row scoping is implemented automatically and consistently with the gate rather than left to each query author; tenant isolation is the default; and account-less, instantly revocable cross-tenant sharing is built in. It does not aim to match the relationship-graph expressiveness of Zanzibar/OpenFGA, the formally verified analysis of Cedar, the policy-as-data central audit of OPA, or the hard out-of-process trust boundary those services provide — trade-offs revisited in Section "Discussion".

Problem Statement and Objective

Problem statement. The requirements of Section 1 and the gaps discussed in Section 2 define the problem addressed here: to design an in-process authorization engine for multi-tenant SaaS that makes resource-scoped, cross-tenant, and account-less access first-class, and that is designed to avoid data leaks. In particular, one whose action gate and data-scoping row filter do not disagree about the same request, neither returning a row the action was not authorized to touch nor disguising a forbidden action as a missing resource.

Objective. We design, implement, and evaluate such an engine. Its contributions are:

1. A co-designed enforcement architecture consisting of two mechanisms: an application-pipeline gate and an ORM row filter, which are implemented at three check points (L1/L2/L3) and kept in agreement by a published per-request scope.

2. A single "active tenant" design: every request resolves access for exactly one active tenant, which keeps the gate a plain allow-check, the row filter a single rule, and structure "access denied" and "not found" responses properly.

3. A cross-tenant sharing model of action-matched, server side grants (per-resource and tenant-wide) that take effect only in the owning tenant's

workspace, bounded by issuance rules, plus account-less revocable capability tokens whose validity is a single row lookup.

4. A supporting performance result: effective-access resolution by a batch chain-walk costing O(chain depth) database round trips independent of the number of grants, validated by an exact analytic model and measurement.

The Proposed Authorization Engine

System and threat model. A principal is the identity a request runs as: a user (a real account with a legacy role and a home tenant) or a share capability (no account). Each principal has a home tenant and, at any moment, acts in exactly one **active tenant**. Acting tenant is the home tenant by default, or a foreign tenant it holds a grant on. Every request is resolved for that single active tenant. The rows the principal may reach are the active tenant's rows it is scoped to, and to act in another tenant the principal switches workspace, never seeing two tenants at once.

Decision to make a currently active tenant was made after examining alternatives**.** An alternative design lets one token carry a merged view, which is the union of the principal's home tenant and every tenant it has been granted access to. A user then sees all of them at once. We deliberately reject that design for two reasons. First, user confusion (a design hypothesis we do not test empirically). When rows from several tenants appear together it is ambiguous which tenant a create or edit affects and where a shared item really lives, which invites mistakes on exactly the cross-tenant data that most needs care. Making the active tenant explicit removes that ambiguity, and the user picks one workspace. Second, backend simplification. Resolving for one tenant replaces three sources of complexity with simpler counterparts: the parallel home and granted channels collapse to a single flat statement set; the disjunction over those channels collapses to a single row-filter rule (a row is visible only when it belongs to the active tenant and lies within the principal's scope); and the tenant-aware gate becomes a plain allow-check, with the deny-vs-hide outcome following structurally. Fewer moving parts that must agree mean a smaller surface for the leak-prone bugs this engine exists to prevent.

We assume the application authenticates the principal (e.g., via an identity provider) and that the database is trusted. The adversary is a logged-in principal, possibly a guest or a share-link holder, attempting to exceed its authority: to read or write resources outside its scope, to discover whether a resource exists, to keep using a revoked share, or to assert an ownership or active-tenant value the backend would trust. Accordingly, two values are never taken from the caller: the resource's owning tenant and the active tenant. Both are derived server-side.

Because access-kit is an in-process engine, its trusted computing base is the entire application process: the gate and the row filter are advisory to

code that chooses to call them. The L3 row filter in particular is an ORM convenience and can be bypassed: by *IgnoreQueryFilters*, by raw SQL or a micro-ORM, by a second or misconfigured *DbContext*, or by a projection that reads columns off a type that does not carry a resource marker. This is the defining trade-off against out-of-process policy-decision points such as Zanzibar [7], OPA, or Cedar, which isolate the decision in a separate service or sandbox, gaining a hard trust boundary at the cost of a network or IPC hop on every check. access-kit deliberately takes the in-process side of that trade-off. The mitigations are to centralize data access on one configured context, to forbid *IgnoreQueryFilters* outside the authorization store (enforceable by a Roslyn analyzer or an architecture test), and, where a hostile in-process actor must be assumed, to carry the same tenant-scoped predicate at the database with row-level security [17] as defense in depth.

Authorization model. The model vocabulary displayed in Table 1.

An action is an "area:verb" string. Matching supports two wildcards: "*" matches everything and "area:*" matches any verb in an area. A resource selector is either global (every resource, bounded in practice by the tenant row filter) or an explicit set of root-aggregate ids. A statement is an "Allow" or "Deny" of an action over a selector. A policy is a named set of statements with an optional parent policy. A principal's statements along a policy chain are the union over the chain. A grant (*PrincipalPolicy*) binds a principal to a policy, optionally narrowed by a resource constraint, and records the owner tenant (*ResourceTenantId*) of the resources it reaches.

The resource hierarchy is expressed through three marker interfaces the host's entities implement, so the engine never references concrete domain types:

• *IRootResource*: a root aggregate (e.g., a Project) carrying its own *Id* and *TenantId*;

Table 1. **Core concepts**

Concept	Type	Meaning
Action	area:verb string	unit of authorization, e.g., document: update; supports "*" and "area:*"
Statement	*Allow* / *Deny* over a resource selector	a single authorization clause
Resource selector	*global* or an explicit id set	what a statement applies to
Policy	named statement set + optional ParentPolicyId	reusable, inheritable permission set
Grant	principal → policy (+ optional scope-down, owner tenant)	binds a principal to a policy over resources
Share token	revocable row carrying its own statements	account-less capability
Effective access	flattened statements at a point in time	the evaluated decision object; deny wins

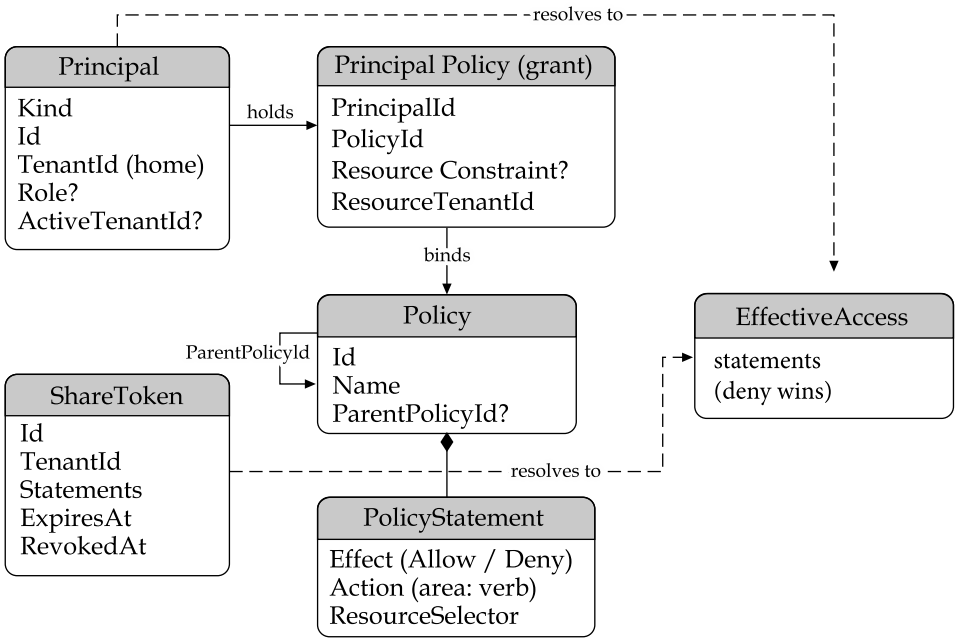

Fig. 1. Authorization data model

• *IChildResource*: homed to a root via *RootId*;
• *ITenantResource*: tenant-owned (carries a *TenantId*).

Effective access is the flattened set of statements for a principal at request time, with deny-wins precedence: an action is allowed if and only if some Allow matches and no Deny matches. An action, optionally on a specific resource, is allowed exactly when some Allow statement matches the action and either applies globally or covers that resource, and no matching Deny statement exists (deny wins). A request that names no resource asserts only the action, leaving the row filter (L3) to narrow the rows.

Figure 1 shows the data model.

A principal resolves to an EffectiveAccess either through its role and grants (walking policy inheritance) or, for a share principal, directly from a token's statements.

Enforcement checkpoints. A single authorized request passes three enforcement check points, realized by two mechanisms: a gate (check points L1 and L2) and a row filter (L3). These two mechanisms are the two points of the title, and the engine's contribution is to keep them in agreement (Fig. 2).

The gate (L1/L2) runs in the application pipeline before any data request is processed; the row filter (L3) runs inside the ORM on every query.

Throughout, we name two outcomes in transport-neutral terms: an access denied response and a not found response. In the HTTP reference adapter these correspond to 403 Forbidden and 404 Not Found, as the figures show.

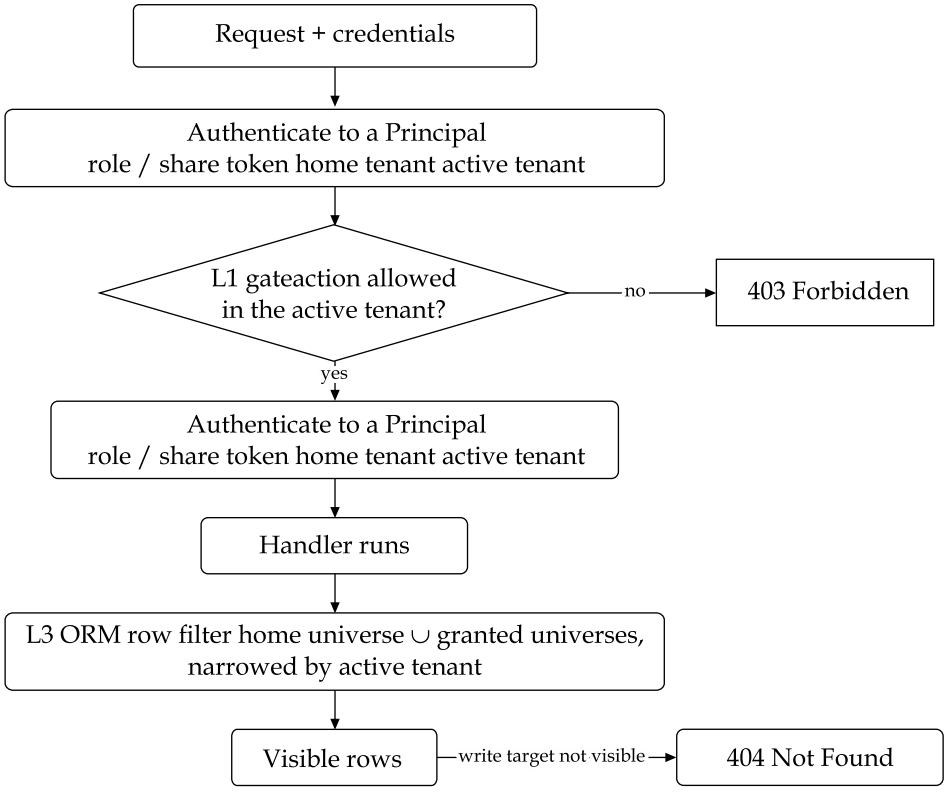

Fig. 2. Request flow

The **gate** lives in the request pipeline (in our case, a MediatR pipeline behavior). It is explicit, cheap, and the natural place to deny an action outright. But it sees only the action and (optionally) a resource id, not the set of rows a query will return. The **row filter** implemented as a global query filter in the ORM [18]. It is implicit so every query against a marked entity is scoped automatically, and a handler cannot accidentally return unscoped data. But it cannot produce an access denied response: an unauthorized row is simply absent.

Using either one alone is unsafe. A gate without a row filter relies on every query author to scope correctly (the BOLA defect [4]). A row filter without a gate cannot distinguish "forbidden" from "does not exist," and cannot deny an action before its side effects. Access-kit uses both, and the gate publishes the per-request scope that the row filter consumes. Both mechanisms then read the same EffectiveAccess, resolved for the active tenant, so the gate consults it to allow the action, and the filter to bound the rows. The actions the gate authorizes and the rows the filter exposes are therefore derived from one structure rather than two independently maintained code paths, the usual source of gate/filter divergence. We argue from this shared construction that the two stay consistent. Verifying that agreement in a given deployment is the integrating host's responsibility.

The row filter is a single rule per resource family, applied over the active tenant's rows, so a row is kept only when it belongs to the active tenant and lies within the principal's scope. The matching predicate covers a root aggregate, a tenant-aware child, a scope-only child, and an off-spine tenant row.

Denying an action versus hiding a resource. Placing an action gate (L1) in front of a row filter (L3) raises one case the filter alone cannot report correctly: a principal that may see a resource but not act on it. A guest with read-only access to a foreign tenant who issues a write must receive an "access denied" response, not a "not found" one. The latter would misreport a forbidden action as a missing resource and leak which ids exist. Because access is resolved for a single active tenant, this falls out structurally. The principal's effective access in that tenant simply contains no matching write statement, so the plain gate denies the action before any row is loaded, while the read still passes. No special tenant-aware check is needed. An access denied response does itself disclose that something exists, so where a resource must be hidden even from principals who lack read access the application returns a uniform not found response (deny read and write alike), which the row filter already produces by making the row invisible.

Multi-tenant isolation and cross-tenant sharing. Every tenant-owned row is filtered to the active tenant, so the isolation is implemented by default. Because a request acts in one tenant, a result set never contains a second tenant's data. Sharing is what lets a principal act in a tenant other than its home: a grant owned by tenant A (its *ResourceTenantId* set server-side from the granting context, never trusted from the caller) gives the grantee statements that take effect only while it is acting in A's workspace. Switching to that workspace re-mints the token with A as the active tenant. The grantee's home role does not follow it across the boundary, so in A it can do exactly what A granted and no more.

Grants come at two grains, **per-resource** (scoped to specific root-aggregate ids) and **tenant-wide** (the whole tenant), and are action-matched: a read-only grant contributes only its read statements, so a document: view grant can never satisfy a document:update request. Resolved for the active tenant, the row filter therefore needs only two inputs (Table 2).

Two issuance safeguards bound sharing. Grantable templates model the common shapes: for example, *ResourceReader/ResourceEditor* (attached

Table 2. **Inputs to the row filter (resolved for the active tenant)**

Input	Source	Meaning
active tenant	activeTenantId (server-derived)	the one tenant every row is filtered to
in-scope roots	ResourceScopeFor(area)	*global*, or the root-aggregate ids the principal is scoped to in that tenant

with a resource constraint, i.e. per-resource) and *TenantGuestReader/TenantGuestManager* (attached without one, i.e. tenant-wide). Guest grant rules bound what a tenant may grant by area, for example, read verbs in any non-protected area, write verbs only in whitelisted areas, and the policy, tenant, admin, and share areas never grantable. Wildcards are never grantable so it caps the blast radius of any share regardless of how the grant is constructed.

Account-less sharing: revocable capability tokens. A *ShareToken* is a database row carrying its own statements, an *ExpiresAt*, and a *RevokedAt*. The share link carries only a compact, URL-safe encoding of the row id (a version byte plus the 16-byte id in base64url, ~23 characters), and therefore no authority of its own. Because the row is the single source of truth for scope, tenant, expiry, and revocation, revoking a share is a single column write. There is no signed bearer token to invalidate and no cache to clear, in contrast to self-contained capabilities such as Macaroons [16]. Validity is decided by reading the row: the share is active when it has not been revoked and has not yet expired, so revocation and expiry take effect on the next request. The version byte lets the link format be changed while old links are rejected cleanly. The web adapter turns a valid link into a Principal (subject share|<id>), so the rest of the request pipeline treats a share like any other principal.

Token entropy and link hygiene. Since the link is the authority until the row is revoked, the row id must be unguessable. Access-kit mints it from a cryptographically secure RNG as a 128-bit value, leaving an attacker no better strategy than brute force over the full 128-bit space. It must not be a sequential or time-ordered identifier (for example a version-7 GUID), which would let one live link expose the range of others, nor *Guid.NewGuid()*, whose output is not contractually a CSPRNG. Because the capability transmitted in a URL, it therefore inherits URL-leakage hazards, such as browser history, the Referer header, server and proxy access logs, and shared-screen exposure. The mitigations are a short expiration timespan, instant revoke, HTTPS-only transport, treating the link as a secret, and optionally exchanging it once for a cookie-bound session so the capability does not linger in the address bar.

Active-tenant token exchange. When a user can act in several tenants, the application exchanges its identity token, once and server-side, for a short-lived application token carrying an *activeTenantId*, re-minted on every workspace switch (in the spirit of OAuth 2.0 [19] and its token-exchange extension [20]). The requested tenant is validated against the caller's home tenant together with the tenants it holds a grant on. A request outside that set is rejected. Because every re-mint re-validates against current grants, a revoked grant stops working within one token lifetime, and the active tenant is never a client-supplied value the backend trusts. Performing this exchange once on the server is the fix for the identity/access *double-exchange* flaw, in which a client is trusted to assert which tenant it is acting in (Fig. 3).

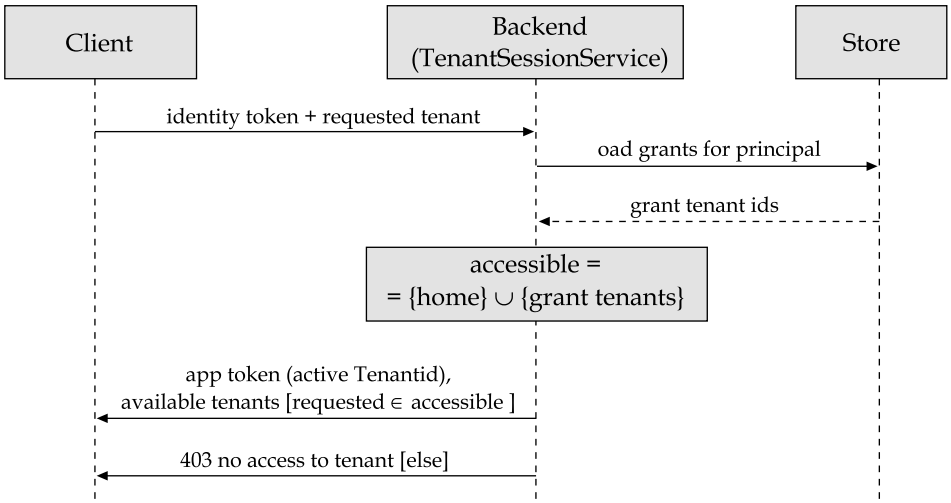

Fig. 3. Active-tenant exchange

To act in another tenant the caller exchanges its token, server-side, for one carrying that activeTenantId; the requested tenant is validated against the caller's own grants and never trusted from the client.

Effective-access resolution. Resolving a user's effective access for the active tenant starts from the role's baked-in statements only when the principal is acting in its home tenant, and adds, for each grant owned by the active tenant, the union of statements along the grant's policy-inheritance chain, intersected with the grant's optional scope-down constraint. The result is one flat statement set for that tenant. A share principal resolves directly from its token's statements.

The naive way to walk policy inheritance is one database query per policy per grant, which results in an N+1 pattern. Access-kit instead resolves inheritance by a batch chain-walk, which collects the frontier of policy ids at each hierarchy level and loads that whole level in one query, deduplicating already-loaded policies (which also guards against cycles). A principal with any number of grants therefore costs O(chain depth) queries (typically one to three), regardless of how many grants share those policies. The store reads current state on every call, so grant and revoke are immediate with no cache to invalidate. A caching decorator can wrap the store if a hot path needs it.

A database that supports recursive common table expressions (WITH RECURSIVE, e.g., PostgreSQL) can resolve the entire policy closure for a principal's grants in a single query: O(1) round trips, one fewer order than the batch walk's O(chain depth). We implement and benchmark this alternative in Section "Experimental results" and it is, as expected, the fastest. The batch walk in this case is not presented as round-trip-optimal, but as a solution which is located above the storage port. In this case, there is no need to use database-specific recursive SQL, runs unchanged on any backing store (relational, document, or in-memory), and keeps policy-

resolution logic in application code rather than pushing it into the database. Its contribution is removing the N+1, turning O(grants × depth) round trips into O(depth), while a recursive CTE trades that portability for one fewer order of round trips. Both sit far below the naive baseline. The choice is portability versus a database-native single round trip.

Results

Analytic cost model. Let G be the number of grants a principal holds, D the depth of the policy-inheritance chain they resolve through, and Q the number of database round trips spent loading policies. The naive resolver issues one policy query per level per grant, so $Q_{naive} = G \times D$ (plus one grants-load). The batch chain-walk loads one query per level and deduplicates shared policies, so $Q_{batch} = D$, independent of G. Table 3 (and the exact counts in Table 4) follow directly.

If a single round trip costs τ (network + execution), end-to-end resolution latency is $\approx Q \times \tau$. The exact counts (Table 4) give this projection directly for $\tau \in \{0.2, 1.0\}$ ms.

Experimental results. We implemented the harness as a BenchmarkDotNet project. The query-count benchmark counts logical round trips through an instrumented store. The timed benchmarks measure in-process cost, and the database-backed benchmarks run against PostgreSQL 16 in a Testcontainers container on the same host. The environment was an Apple M3 Pro, .NET 10.0.3 (Arm64 RyuJIT), under the ShortRun job.

The round-trip counts confirm the analytic model exactly (Table 4): the batch walk's count equals the chain depth and does not grow with the number of grants, while the naive count grows as $G \times D$, so the reduction factor is exactly the grant count G: linear in G, not a fixed multiplier (500× at $G = 500$, and 5000× at $G = 5000$). The point is the shape: naive scales with grants, the batch walk does not.

Because the difference is round trips, wall-clock impact scales with database latency, and on PostgreSQL the projection holds (Fig. 4). Naive resolution rises with the grant count (2.8 ms at 10 grants, 27.7 ms at 100 at depth 1, and 83.7 ms at 100 grants over a depth-3 chain), because every policy load is a separate round trip. The batch walk stays flat in the grant

Table 3. **Resolution cost (database round trips for policy loading)**

Quantity	Naive	Batch
policy round trips	$Q(G \times D)$	$Q(D)$
sensitivity to grants G	Linear	None
sensitivity to depth D	Linear	Linear

Table 4. **Measured (counted) policy round trips**

Grants	Depth	Naive	Batch	Reduction
10	3	30	3	10×
100	3	300	3	100×
500	3	1500	3	500×
500	5	2500	5	500×

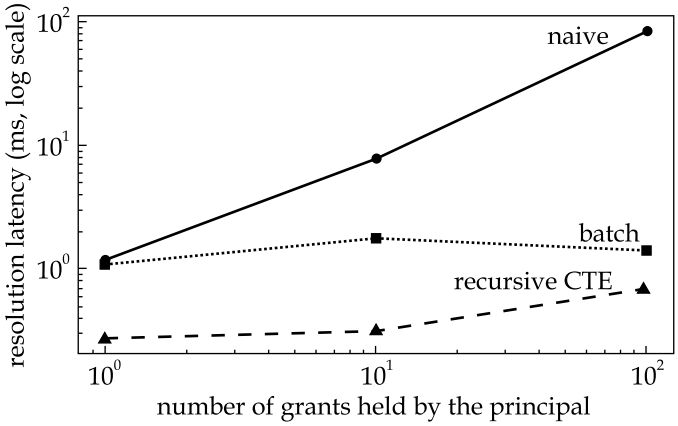

Fig. 4. Measured resolution latency on PostgreSQL (chain depth 3, log–log) naive grows with the grant count, the batch walk is flat in grants, and the recursive CTE is fastest at one round trip

count at 0.6-1.8 ms, and the recursive CTE is fastest at 0.27-0.69 ms in a single round trip. At 100 grants and depth 3 the three approaches differ by orders of magnitude (83.7 vs 1.4 vs 0.69 ms), confirming that the batch walk removes the N+1 and that, where the database offers recursive CTEs, a single database-native query is round-trip-optimal.

Resolution therefore dominates the per-request budget (one to a few round trips, ≈0.3-1.8 ms) and is the natural cache point: the caching decorator over the access store resolver serves effective access from memory, collapsing those round trips to zero on a hit. Every other cost is negligible beside it. Against a zero-latency in-memory store (the worst case for the batch walk) the batch walk is ~1.8× slower than a straight walk at a single grant (941 ns vs 532 ns) but overtakes it by ~10 grants, running ~2-2.4× faster and allocating ~40% as much at 100-500 grants, so removing the N+1 is a net win even on CPU. The in-memory gate runs in tens of nanoseconds to ~1 μs (41 ns at 5 statements, 1.14 μs at 500), and the share-link codec encodes and decodes in ~35 ns. Both are three to five orders of magnitude cheaper than one database round trip.

The L3 row filter's cost is driven by the size of the in-scope id set it must match (the IN/ANY list it emits), not the table's row count. Over a fixed pool of 5000 home rows, reading the visible set takes ≈1.4-1.8 ms for 0-10 ids, 3.0 ms at 100, 3.5 ms at 1000, and 16.2 ms at 10000 ids. Modest sharing (≤1000 shared roots) costs at most ~2× the base scan, while a 10000-id set is expensive and argues for tenant-grain sharing (a single tenant id) over enumerating thousands of roots. A by-row-count sweep on the same PostgreSQL instance confirms the predicate adds no measurable cost beyond the base scan: 2.22 ms filtered vs 2.11 ms unfiltered at 10000 rows. In this case, the id-set size, not the row count, is the cost variable.

Security: leakage scenarios addressed. Table 5 lists the information-leakage scenarios access-kit addresses against a naive role-check baseline. How it differs qualitatively from the other engines is discussed in Section 2.

Discussion: advantages and limitations. Advantages:

(i) In-process: every decision is an in-memory check with no extra network hop, unlike Zanzibar-style services [7].

(ii) Co-designed enforcement: the gate and row filter read one published scope, eliminating the most common object-level authorization defects [4] and reporting access denied versus not found consistently, an agreement we argue from the shared construction rather than prove (see Limitations):

(iii) Instant revocation: both grants and share tokens are read from current state, so revocation needs no propagation window.

(iv) Storage-agnostic: the engine depends on ports, with an Entity Framework Core adapter provided.

(v) Migration-friendly: existing roles become baked-in statements with no data migration.

(vi) Single active tenant: resolving for one tenant keeps the gate a plain allow-check and the row filter a single per-family rule and makes the deny-vs-hide outcome structural, at the cost that acting in another tenant needs an explicit workspace switch.

Limitations: (i) The gate and row filter derive from one shared EffectiveAccess, so they are designed to agree; but because access-kit is an in-process library, upholding that agreement in a given deployment rests with the host that integrates it, and a host can confirm it for its own domain entities with conformance tests on its side. (ii) Access-kit is in-process: its trusted computing base is the whole application process, and the L3 filter can be bypassed by in-process code (IgnoreQueryFilters, raw SQL, a second context), so for a hostile-code threat model it must be paired with database row-level security. (iii) The engine is .NET-specific; cross-language or centrally audited/analyzable deployments are better served by OPA [14], Cedar [13], or OpenFGA [8]. (iv) The row filter captures a per-request scope; with ORM context pooling the compiled model must be keyed per scope to avoid cross-tenant reuse. (v) "Instant" revocation is bounded by token lifetime for active-tenant claims.

Conclusions

We presented access-kit, an embeddable authorization engine for multitenant SaaS that treats partial, cross-tenant, and account-less access as first-class concerns. Its central idea is a co-designed pair of enforcement mechanisms, an application-pipeline gate and an ORM row filter that read one published per-request scope, which makes the two report errors consistently. On top of this, every request acts in a single active tenant, chosen over a merged multi-tenant view to remove user confusion and simplify the backend, so the gate stays a plain allow-check, the row filter a single per-family rule, and the deny-vs-hide outcome structural. Cross-tenant access comes from action-matched, server-derived grants that take effect only in the owning tenant's workspace, alongside account-less, instantly revocable capability tokens. Measured on PostgreSQL, effective-access

resolution by the batch chain-walk costs O(chain depth) database round trips independent of the number of grants, a reduction over the naive N+1 that is linear in the grant count, and we compare it against a single-round-trip recursive CTE. The per-request cost is dominated by this resolution, with the in-memory gate adding nanoseconds.

Future work: a machine-checked formalization of the gate and row-filter agreement invariant; a distributed caching decorator with explicit revocation propagation bounds for very hot paths; packaging and a broader empirical study against production workloads and alternative engines; and extending the resource model beyond a two-level root/child hierarchy.

DECLARATIONS

AI and automated tools usage. AI tools were used to support drafting, language editing and spellcheck. All technical claims, experimental design, measurements, and conclusions were verified by the author, who bear full responsibility for the content; no material was included without author verification.

Conflict of interest. The author(s) declare no conflict of interest.

Grant or financial support. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

1. Chong F., Carraro G., Wolter R. Multi-Tenant Data Architecture. Microsoft *Developer Network*, 2006.
2. Ferraiolo D.F., Sandhu R., Gavrila S., Kuhn D.R., Chandramouli R. Proposed NIST standard for role-based access control. ACM *Transactions on Information and System Security*, 2001, Vol. 4 (3), 224–274. https://doi.org/10.1145/501978.501980
3. Sandhu R.S., Coyne E.J., Feinstein H.L., Youman C.E. Role-based access control models. *Computer*, 1996, Vol. 29 (2), 38–47. https://doi.org/10.1109/2.485845
4. OWASP Foundation. OWASP API Security Top 10 API1:2023 Broken Object Level Authorization. 2023. URL: https://owasp.org/API-Security/ [Accessed Jun. 2026]
5. Hardy N. The Confused Deputy: or why capabilities might have been invented. ACM SIGOPS *Operating Systems Review*, 1988, Vol. 22 (4), 36–38. https://doi.org/10.1145/54289.871709
6. Hu V.C., Ferraiolo D., Kuhn R., et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication 800-162*, 2014. https://doi.org/10.6028/NIST.SP.800-162
7. Pang R., et al. Zanzibar: Google's Consistent, Global Authorization System. *USENIX Annual Technical Conference (ATC)*, 2019, 33–46.
8. OpenFGA Authors. OpenFGA: a high-performance and flexible authorization/permission engine. Cloud Native Computing Foundation, documentation, 2024. URL: https://openfga.dev [Accessed Jun. 2026]
9. Luo Y., et al. Casbin: an authorization library that supports access control models like ACL, RBAC, ABAC. Documentation, 2024. URL: https://casbin.org [Accessed Jun. 2026]
10. Oso Security. Polar: a declarative logic language for authorization. Documentation, 2024. URL: https://www.osohq.com [Accessed Jun. 2026]
11. Okta.Auth0. Auth0 Fine-Grained Authorization (FGA). Documentation, 2024. URL: https://auth0.com/fine-grained-authorization [Accessed June 2026]
12. Amazon Web Services. AWS Identity and Access Management User Guide policies and ABAC. 2024. URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/ [Accessed Jun. 2026]

13. Cutler J.W., et al. Cedar: a New Language for Expressive, Fast, Safe, and Analyzable Authorization. *ACM on Programming Languages*, 2024, Vol. 8 (OOPSLA1), 670–697. https://doi.org/10.1145/3649835
14. Open Policy Agent Authors. Open Policy Agent (OPA) and the Rego policy language. Cloud Native Computing Foundation, documentation, 2024. URL: https://www.openpolicyagent.org [Accessed Jun. 2026]
15. Dennis J.B., Van Horn E.C. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 1966, Vol. 9 (3), 143–155. https://doi.org/10.1145/365230.365252
16. Birgisson A., Politz J.G., Erlingsson Ú., Taly A., Vrable M., Lentczner M. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. *Network and Distributed System Security Symposium (NDSS)*, 2014. https://doi.org/10.14722/ndss.2014.23212
17. PostgreSQL Global Development Group. PostgreSQL Documentation Row Security Policies. 2024. URL: https://www.postgresql.org/docs/current/ddl-rowsecurity.html [Accessed Jun. 2026]
18. Microsoft. Entity Framework Core Global Query Filters. Documentation, 2024. URL: https://learn.microsoft.com/ef/core/querying/filters [Accessed Jun. 2026]
19. Hardt D. (Ed.). RFC 6749: *The OAuth 2.0 Authorization Framework*. IETF, 2012, 75 p. https://doi.org/10.17487/RFC6749
20. Jones M., Nadalin A., Campbell B., Bradley J., Mortimore C. *RFC 8693: OAuth 2.0 Token Exchange*. IETF, 2020. https://doi.org/10.17487/RFC8693
21. Akinshin A. Pro. NET Benchmarking: The Art of Performance Measurement. Apress, Berkeley, CA, 2019.

Received 09.07.2026
Accepted 20.07.2026
Published XX.XX.2026

А.В. МЕЛЬНИЧЕНКО, канд. техн. наук, асистент,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»,
просп. Берестейський 37, Київ, 03056,Україна
https://orcid.org/0009-0000-3588-4772
artemxl@gmail.com

О.В. ШАЛДЕНКО, канд. техн. наук, доцент,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»,
просп. Берестейський 37, Київ, 03056,Україна
https://orcid.org/0000-0001-6730-965X
o.shaldenko@gmail.com

ДВОТОЧКОВИЙ МЕХАНІЗМ АВТОРИЗАЦІЇ З ВІДКЛИЧНИМ СПІЛЬНИМ ДОСТУПОМ ДЛЯ СЕРЕДОВИЩ З РОЗДІЛЕННЯМ ДАНИХ ПО ОРГАНІЗАЦІЯМ

Вступ. Хмарні платформи з розділенням організацій *(multi-tenant*), що розповсюджуються за моделлю «програмне забезпечення як послуга» (*SaaS*) і зберігають спільні дані окремих організацій, дедалі частіше потребують керування доступом, яке не може забезпечити рольова модель (*RBAC*): доступу до окремих ресурсів, а не ролей; контрольованого спільного доступу в декількох організацій; а також посилань для спільного доступу, які можна відкликати в будь-який момент. Сервер під час обробки запиту водночас виконує дію та оперує набором даних, тож авторизація має два аспекти, які мусять залишатися узгодженими: чи дозволена дія в цілому та які записи запит має право бачити або змінювати. Коли ці механізми неузгоджені, система або розкриває дані,

або повертає оманливі відповіді, що належить до класу вад «порушення авторизації на рівні об'єкта».

Мета роботи. Спроєктувати, реалізувати та оцінити вбудований механізм авторизації для *SaaS*-застосунків для реалізації часткового доступу (на рівні окремого ресурсу) всередині організації, надання спільного доступу між організаціями і спільного доступу за посиланням. Застосування механізму має не допустити непомітного витоку даних, зокрема, щоб фільтри при авторизації дії та фільтри записів з бази даних не суперечили одне одному щодо одного й того самого запиту.

Методи. Запропоновано компактну явну модель (суб'єкти, дії у форматі «область:операція», твердження політик із пріоритетом заборони та успадкуванням, ієрархія ресурсів) і узгоджений механізм застосування: шлюз на рівні обробки вхідного запиту в застосунку, який відповідає за перевірку дозволу на дію, та фільтр рядків на рівні *ORM*, який відповідає за фільтрацію даних, які дозволені для поточного суб'єкта. Їхню узгодженість забезпечує контекст області видимості запиту, який забезпечує шлюз і постачається у фільтр записів, тож обидва механізми узгоджені. Кожен запит діє в межах однієї активної організації: доступ обчислюється саме для активної організації, тож на рівні шлюзу здійснюється проста перевірка дозволу, а на рівні *ORM* застосовується фільтр записів. Ефективний доступ обчислюється пакетним обходом ланцюга політик, що усуває проблему *N*+1 запитів; розглянуто також альтернативу на основі рекурсивного *CTE*. Продуктивність оцінено аналітичною моделлю вартості та вимірюваннями на *PostgreSQL* за допомогою *BenchmarkDotNet*.

Результати. Реалізований механізм дозволяє надавати частковий доступ до ресусів, доступ поза межами організації та спільний доступ за посиланням, при цьому не потребує сторонніх сервісів та працює в рамках самого додатку. Для визначення ефективного доступу використовується пакетний обхід політик доступу з наданими дозволами. Кількість звернень до бази даних при цьому дорівнює глибині ланцюга політик і не залежить від кількості надань суб'єкта; рекурсивний *CTE* розв'язує всю задачу за одне звернення. Під час проведення експериментів з використанням СКБД *PostgreSQL* було отримано наступні показники: за умови 100 наданих дозволів і глибини ланцюга що дорівнює 3, класичний підхід без пакетної обробки потребує близько 83,7 мс, пакетний обхід — близько 1,4 мс, а *CTE* — 0,69 мс. Операції на рівні шлюзу виконуються за десятки наносекунд, тобто на 3—5 порядків швидше за одне звернення до бази; вартість фільтра рядків визначається радше розміром множини наданих ідентифікаторів ресурсів, ніж кількістю рядків таблиці. Спільний доступ між організаціями виражено через дозволи (*grants*), узгоджені за дією, з визначенням власника на боці сервера, що діють лише у робочому просторі організації-власника, а безобліковий доступ — через токени з можливістю відкликання, чинність яких визначається одним зчитуванням запису з бази даних. Узгоджене повідомлення про помилки відрізняє результати «заборонено» від «не знайдено», не розкриваючи існування ресурсу.

Висновки. *access-kit* поєднує механізм ухвалення рішення про надання доступу та фільтр рядків *ORM*, які зчитують один контекст області видимості, з міжорганізаційним і безобліковим спільним доступом. Кожен запит діє в межах однієї активної організації, таким чином зменшується імовірність здійснити дію в сторонній організації, і суттєво спрощується імплементацію, а для дій в іншій організації, користувач перемикає робочий простір. Механізм імплементовано як узагальнену еталонну реалізацію з адаптерами для *Entity Framework Core*, *MediatR* та *ASP.NET Core*.

Ключові слова: *керування доступом; авторизація; обмеження за ресурсами; спільний доступ; токени доступу; захист на рівні рядків; .NET.*