

MULTI-TENANT SAAS

Building a multi-tenant authorization system **that survives its own customers**

Tenant isolation is the part everyone designs for. What breaks a model is everything that comes after it — the customer who belongs to two tenants, the partner who belongs to none, the admin who needs to see a customer's data to answer a ticket, and the access that was only ever meant to last a fortnight.

The shape of the problem

Every B2B SaaS platform arrives at the same list. Keep tenants apart. Let each one organise itself into teams and projects. Let customers administer their own users rather than filing tickets. Let two customers collaborate without merging. Let your own staff help without holding standing access to everything.

Individually, each is tractable. Together they defeat a role table, because a role names a *level* of access but has nowhere to record *which object* it applies to — and in a multi-tenant system, which object is the entire question.

What this paper covers

A concrete model, expressed in SpiceDB's schema language, that handles the full list without a separate mechanism for each requirement. Every pattern here is one of four things: a definition, a relation, a permission expression, or a relationship written at runtime.

It assumes you have read nothing about Zanzibar. Where a design has a real cost, that cost is stated rather than skipped.

The useful question is not “can this system isolate tenants?” — everything can. It is “**what does the fifth exception cost, once four are already in production?**”

CONTENTS

1. Tenant isolation
2. Organizations, teams and projects
3. Roles, and where they go wrong
4. Delegated administration
5. Cross-tenant sharing
6. Support and break-glass access
7. Temporary permissions
8. What to verify before you commit

AUDIENCE

Platform and security engineers designing authorization for a multi-tenant product

ASSUMES

Familiarity with RBAC. No prior knowledge of Zanzibar or SpiceDB

READING TIME

About twelve minutes

1 · TENANT ISOLATION

Make isolation structural, not a filter

Most breaches of tenant isolation are not failures of the rule but failures to *apply* it — a query missing its `WHERE tenant_id = ?`, a new endpoint nobody scoped, a background job running as a service account. The defence is a model in which the unscoped query is not expressible.

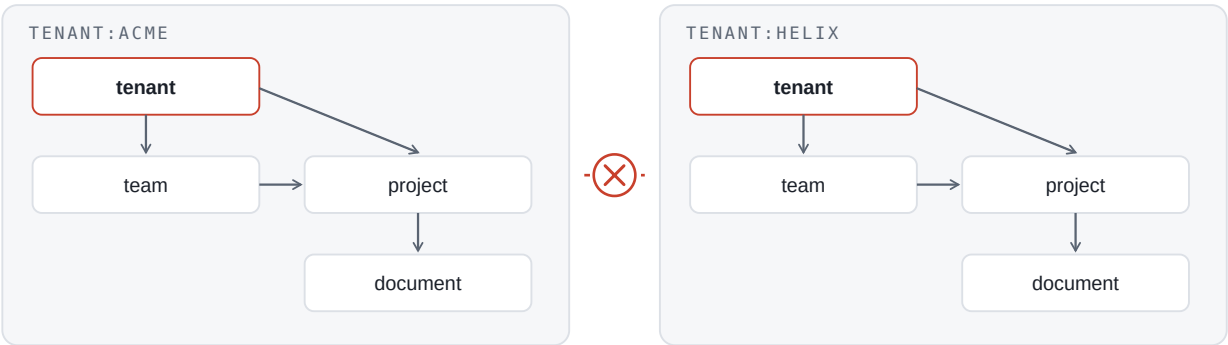


Figure 1 – A document reaches its tenant only through its project. No edge crosses the boundary, so there is no traversal that could return another tenant's object.

2 · ORGANIZATIONS, TEAMS, PROJECTS

One hierarchy, inherited downward

A grant at any level reaches everything beneath it. A tenant administrator reaches every project without being named on one; a team lead reaches their team's projects and no others.

The hierarchy is written once. Each new resource type declares its level and asks that level the question, rather than implementing the walk itself.

```
// each level knows only its parent
definition tenant {
  relation admin: user
  relation member: user
  permission administer = admin
}

definition team {
  relation tenant: tenant
  relation lead: user
  relation member: user
  permission view = lead + member
                  + tenant->administer
}
```

3 · ROLES

A role is a relationship, not a column

The failure mode is predictable. Because a role carries no object, the object ends up encoded into the role's *name* — `project_42_editor` — or into a conditional no audit will find. A relationship puts the object back in the record.

REQUIREMENT	AS A ROLE TABLE	AS RELATIONSHIPS
Editor on everything	One role. Well suited	One relationship per object, or one on the parent
Editor on one project	A role per project, or a conditional in code	One relationship. Direct
“What can this user see?”	Not answerable — the store never knew the objects	One call. Direct
Revoke everything, now	Delete rows across every table that encoded it	Delete the relationships; derived access ends with them

The requirements that break role tables

Isolation and hierarchy are the half most designs get right. These four are the half that arrives later, usually one enterprise deal at a time. Each is a single relationship or a single expression — not a new subsystem.

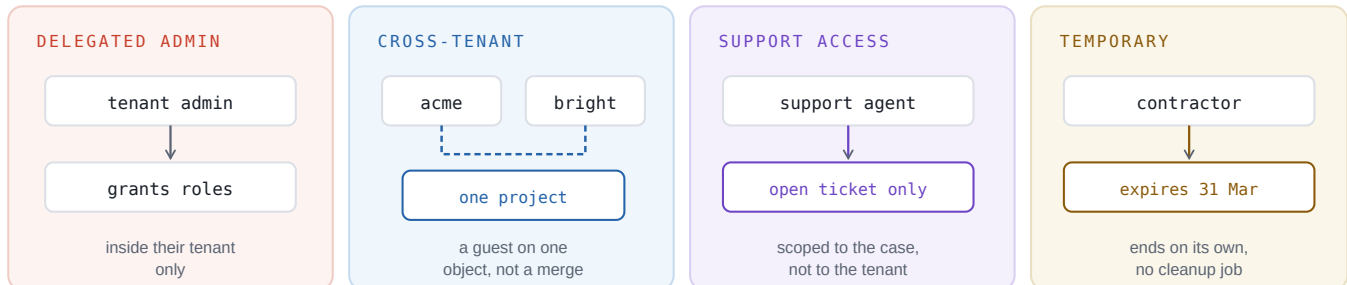


Figure 2 – Each exception grants access to one object or one case, rather than widening a role that applies everywhere.

4 · DELEGATED ADMINISTRATION

Customers administer themselves

The requirement is that a customer's own administrator can add users, create teams and grant project access — **without that power reaching outside their tenant**, and without your support team becoming a provisioning queue.

This falls out of the hierarchy rather than needing a feature. `tenant->administer` reaches everything beneath that tenant and, by construction, nothing beside it. The same expression that grants the power bounds it.

6 · SUPPORT AND BREAK-GLASS

Scope to the ticket, not the tenant

Your staff need to see customer data to resolve a case. Standing access to every tenant is the usual shortcut and the one that appears in every breach write-up.

Model the **case** as the object. A support agent assigned to an open ticket reaches the records that ticket concerns; when the ticket closes, the relationship goes and so does the access. It is auditable because the grant is a row, not a role nobody can enumerate.

5 · CROSS-TENANT SHARING

A guest, not a merge

Two customers need to work on one thing — a deal, a case, a joint project. The instinct is to create a shared workspace or to copy data. Both are expensive and both are hard to unwind.

The relationship form is narrower: name the outside user on the one object they need. They are a subject in your graph already; nothing about their own tenant changes, and ending it is one delete.

```
relation external_collaborator: user
permission view = owner + collaborator
+ external_collaborator
+ team->member + tenant->administer
```

7 · TEMPORARY PERMISSIONS

Access that ends on its own

Contractor engagements, trial periods, incident response, a quarter-end audit. Each is access that should stop on a date — and in most systems stops when somebody remembers.

Attaching an expiry to the relationship itself moves that from a process to a property. Nothing is deleted and no job runs; the check simply stops finding a live path.

```
use expiration

// the end date is mandatory here
relation contractor: user with expiration
```

What the model does not solve

A schema answers “who may do what.” Three problems sit outside it, and a design that ignores them will meet them in production instead.

STALENESS

A revoked permission and a cached allow. If a check is answered from a replica that has not seen the revocation, it returns the right answer for stale data. Pin checks that must reflect a specific write to that write's token; let the rest take the fast path.

TWO SYSTEMS, ONE REQUEST

Your database and the permission store. No transaction spans both, so a crash between the two writes leaves them disagreeing. A transactional outbox — the intent written inside your own transaction, drained separately — is the common answer.

IDENTITY

A stable subject per person and per service. This has to exist before any of the above means anything, and an authorization layer cannot supply it. In most migrations it is the long pole, not the schema.

SAID PLAINLY

None of this makes a decision about *who should* have access. That is a governance question answered by your security and compliance teams. What a relationship model provides is a single place where the answer is written down, enforced identically by every service that asks, and auditable afterwards. It makes the policy enforceable, not correct.

What to verify before you commit

Questions worth putting to any authorization system, including this one. They are the ones whose answers diverge.

ASK	WHY IT SEPARATES SYSTEMS
Can one user hold different roles in two tenants, with neither leaking?	Tests whether identity is global while access is scoped. Role tables usually answer by duplicating the user.
How do I grant access to one object without inventing a role?	The per-object case is where role models encode the object into a name or a conditional.
Can I ask “which objects can this user see?” in one call?	Needed for list endpoints, search filtering and AI retrieval. Many systems can only answer one object at a time.
What happens to a cached allow the moment access is revoked?	Distinguishes a consistency model from a hope.
Can a grant expire without a cleanup job?	Determines whether temporary access is a property or a process.
Can I prove, to an auditor, who could reach this record last quarter?	Requires the object in the record and a history of changes.

Where to start

Model one service, not the estate. Pick the surface where the exceptions already hurt — usually sharing or external collaboration — and express its real rules, including those living in application code.

Then run both systems in parallel and diff every answer under production traffic before cutting over. Disagreements are almost always a rule nobody wrote down — worth finding whatever you adopt.

About SpiceDB

SpiceDB is an open-source authorization database inspired by Google's Zanzibar paper, maintained by AuthZed. The schema language, consistency model and APIs described here are all in the open-source project.

Try the model: the Playground runs in a browser with nothing to install — play.authzed.com. The source is at github.com/authzed/spicedb.