Contents

05 — Database & Prisma: A Thinking Guide

> **Flagline** — Feature Flag SaaS Stack: React, Next.js (App Router), Node.js, TypeScript, PostgreSQL (Prisma), Redis Audience: Backend developer with 5+ years PHP/Laravel experience transitioning to this stack

This document is not a schema dump or a code reference. It is a guide for how to *think* about the database layer of a feature flag SaaS. The goal is to give you the mental models and decision rationale so that you can design, evolve, and debug the data layer yourself. Code snippets appear only when a concept is clearest in code, and they are kept to one to three lines.

Table of Contents

---

1. Why PostgreSQL Over MySQL

If you have been building with Laravel for five years, MySQL is probably what you know best. You know its quirks, its gotchas with character sets, its locking behavior under load. You know how to tune it. So the question is not “is PostgreSQL better in the abstract” – it is “why does PostgreSQL make more sense for *this specific project*?”

The answer comes down to three capabilities that Flagline uses daily, not just as nice-to-haves but as load-bearing parts of the architecture.

JSONB: The Reason That Matters Most

Here is the problem that drives the database choice. A feature flag has targeting rules. One flag might target users in the US on a Pro plan. Another might target users whose email ends with `@internal.company.dev`. A third might target 25% of all users by hashing their user ID. A fourth might combine all of the above with nested AND/OR logic.

The shape of these rules is different for every flag, and the shape will evolve over time as you add new targeting operators (semver comparisons, date ranges, regex matches). This is the kind of data where you need to ask yourself: should I normalize this into relational tables, or should I store it as a structured document in a single column?

Think about it from the query side first. When a flag evaluation request comes in, you need to read the rules for a given flag in a given environment, then evaluate them in application code. You do not need to query *across* rules in the hot path. You do not need "give me all flags that target users in France" on every SDK request. You need "give me the full rule tree for this flag so I can evaluate it." That is a single-row read of a single column.

Now think about the write side. When someone edits targeting rules in the dashboard, they are replacing the entire rule tree for one flag-environment combination. It is not "add one condition to a normalized conditions table." It is "here is the new rule tree, replace what was there." That maps naturally to a single UPDATE on a JSONB column.

MySQL has a JSON column type, but it stores JSON as a text blob that gets re-parsed on every read. PostgreSQL's JSONB stores it in a pre-parsed binary format. The difference matters less for the hot path (where you are caching in Redis anyway) and more for dashboard queries where you sometimes *do* need to search inside rules – "show me all flags that target the `country` attribute." PostgreSQL lets you index JSONB with a GIN index and query it with the `@>` containment operator. MySQL's `JSON_EXTRACT` cannot use an index for this.

> **Coming from Laravel/MySQL:** You have probably stored JSON in MySQL columns before – `$casts = ['settings' => 'json']` in Eloquent. The experience is fine for reading and writing. The difference shows up when you need to *query inside* the JSON, and when you need that query to use an index rather than scanning every row. If you have ever written `->whereJsonContains('settings->theme', 'dark')` in Laravel and noticed it does a full table scan, that is the gap PostgreSQL fills.

The key thinking framework is this: **JSONB is appropriate when the data is read as a whole document, written as a whole document, and only occasionally queried by its internal structure.** Targeting rules fit this pattern exactly. A user's profile settings would also fit this pattern. A table of orders where you regularly need to filter by line item attributes would *not* fit this pattern – that data should be normalized.

Partial Indexes: Indexing Only What You Query

A feature flag system accumulates dead data. Flags get archived. API keys get revoked. But you keep that data for audit trails and the ability to restore. Over months or years of operation, the majority of flags in the database might be archived.

Now consider the evaluation hot path. It only ever queries active, non-archived flags. Every query includes `WHERE is_archived = false`. In MySQL, the index covers all rows, including the

archived ones the query never wants. The database scans past those index entries and discards them. The index is larger than it needs to be, less likely to fit in memory, and slower than it should be.

PostgreSQL supports partial indexes: an index with a `WHERE` clause baked in. You create an index on `flags(project_id) WHERE is_archived = false`, and archived flags simply do not exist in that index. The index is smaller, fits in memory more easily, and every entry in it is a row the hot path actually wants.

The mental model is this: a regular index is a phonebook of everyone in town. A partial index is a phonebook of only the people who are currently reachable by phone. If 80% of the entries in the phonebook are disconnected numbers, why make the delivery driver carry the full book?

> **Coming from Laravel/MySQL:** MySQL has no equivalent feature. Laravel's migration builder has no method for it. When you need a partial index in Prisma, you create a custom migration with raw SQL. Prisma cannot express partial indexes in its schema language, but it can apply them through migrations. The index lives in the database; Prisma just does not "know" about it declaratively.

MVCC: Why Concurrent Reads and Writes Just Work

Flagline has a very specific concurrency pattern: thousands of SDK clients reading flag configurations simultaneously, with a handful of dashboard users occasionally writing changes. This is a 99/1 read/write ratio on the hot path.

Both PostgreSQL and MySQL use Multi-Version Concurrency Control (MVCC), but they implement it differently in ways that matter for this workload.

In PostgreSQL, readers never block writers, and writers never block readers. Period. A transaction reading flag configs will never wait for another transaction that is updating a flag rule. They each see their own consistent snapshot of the data.

MySQL's InnoDB MVCC has similar goals but uses a locking mechanism called next-key locking that can cause unexpected contention. If you have ever hit a deadlock in MySQL under concurrent writes to the same table, or seen `SELECT ... FOR UPDATE` hold up reads in ways you did not expect, that is the behavioral difference. It is not that MySQL is broken – it is that its concurrency model has more edge cases, and those edge cases tend to bite you under exactly the kind of workload Flagline has.

If you have been running Laravel with MySQL for five years and never hit these locking issues, that is probably because your application did not have thousands of concurrent read connections. For Flagline's evaluation API, the cleaner MVCC implementation matters.

> **Coming from Laravel/MySQL:** Think of it this way. In your Laravel app, you probably have a few dozen concurrent requests at most. Flagline's evaluation API can have thousands. At that scale, the difference between "readers never block writers" (PostgreSQL, always) and "readers usually do not block writers but sometimes gap locks cause surprises" (MySQL) becomes the difference between predictable latency and occasional tail latency spikes.

Ecosystem Alignment

The practical reason: the TypeScript/Next.js/Prisma ecosystem has converged on PostgreSQL. Prisma's PostgreSQL support is the deepest – native enums, JSONB, array columns, full-text search. Vercel Postgres is PostgreSQL-only. Supabase, Neon, and Railway are all PostgreSQL. Choosing MySQL in this ecosystem means swimming against the current of the toolchain and fighting for documentation parity.

This is not a technical argument but an ergonomic one. When you search for "Prisma JSONB filtering" you get PostgreSQL answers. When you deploy to Vercel, the managed database is PostgreSQL. The path of least resistance and best documentation is PostgreSQL.

Additional PostgreSQL Features Worth Knowing About

A few more PostgreSQL capabilities that Flagline benefits from, briefly noted.

Native enums. PostgreSQL supports `CREATE TYPE role AS ENUM ('OWNER', 'ADMIN', 'MEMBER')`. Prisma maps its `enum` declarations directly to PostgreSQL enum types. In MySQL, Prisma emulates enums as `VARCHAR` with a `CHECK` constraint, which means the database does not actually enforce enum membership at the type level. For a system where roles and flag types are critical to correctness, real enum types are a meaningful safety net.

Array columns. PostgreSQL supports `TEXT[]`, `INT[]`, etc. Flagline uses this for API key permissions – a single column holds `['EVALUATE', 'SERVER_SIDE']` rather than requiring a join table for a simple list. MySQL has no array column type. You would use a join table or a JSON column, both of which add complexity for what should be a simple list of values.

Full-text search. PostgreSQL has built-in `tsvector`/`tsquery` for searching flag names and descriptions. This is not a replacement for Elasticsearch, but it handles the "search flags by name" use case in the dashboard without adding another service. MySQL has full-text search too, but PostgreSQL's is more flexible in ranking and weighting.

> **Coming from Laravel/MySQL:** These are the features that make PostgreSQL feel like it was designed for the TypeScript SaaS stack. In Laravel, you work around MySQL's limitations with Eloquent conventions (casting JSON, using pivot tables for lists). In this stack, PostgreSQL's native features let you express things directly.

---

2. Data Modeling Thinking for a Feature Flag System

Before writing a single model, you need to answer a series of questions. The schema should emerge from the answers, not from guessing at tables and then figuring out how they relate.

The Questions That Drive the Schema

What are the core entities in a feature flag system?

Start from the user's perspective, not the developer's. A customer signs up, creates an organization, creates a project (maybe "Web App" and "Mobile App"), and within each project creates environments (development, staging, production). Then they create flags and configure each flag

differently per environment. A flag might be enabled in staging but disabled in production. They generate API keys scoped to a specific project and environment, which their SDK uses to fetch flag configurations.

That narrative gives you the entity hierarchy: **Organization (Tenant) -> Projects -> Environments -> Flags**, with flags having per-environment state.

What queries will the evaluation API need to run thousands of times per second?

This is the single most important question. The answer shapes everything. The evaluation API receives a request like "give me all flag configurations for project X, environment Y." It needs to return every active, enabled flag with its targeting rules and rollout configuration. This query runs on every SDK poll, every SSE reconnect, every flag evaluation. It must be fast, and the data it returns must be in a shape that is easy to cache.

What queries does the dashboard need?

The dashboard needs to list flags, filter them, search them, show per-environment state side by side, display audit history, and manage API keys. These queries can tolerate more latency than the evaluation API. They also tend to join more tables (flags with their environments, with the user who created them, with audit logs).

With these questions answered, let us walk through the design decisions.

The Multi-Tenancy Hierarchy: Tenant, Project, Environment, Flag

The hierarchy is **Tenant -> Project -> Environment**, and flags belong to a project while having per-environment state. Here is the reasoning for each level.

Tenant is the top-level isolation boundary. In SaaS terms, this is the "organization" or "workspace." One billing account, one set of team members, one subscription. Every piece of data in the system ultimately belongs to a tenant, and no data should ever leak between tenants.

Why not just use "User" as the top level? Because feature flags are a team tool. Multiple people at the same company need to see and manage the same flags. The tenant is the shared container that users belong to.

Project is a grouping within a tenant. A company might have a web app, a mobile app, and an internal admin tool. Each has its own set of flags. The flag key `dark-mode` in the web app is a completely different flag from `dark-mode` in the mobile app. Projects provide this namespace separation.

Why not skip projects and have flags belong directly to a tenant? Because as soon as a company has more than one application, they need to separate flags by application. Without projects, you end up with naming conventions like `web-dark-mode` and `mobile-dark-mode`, which is fragile and does not scale.

Environment lives within a project. The standard set is development, staging, and production, but users can create custom ones (canary, QA, demo). Environments represent deployment contexts.

Why are environments per-project rather than per-tenant? Because different projects might have different deployment topologies. The web app might deploy to development, staging, and production. The internal admin tool might only have development and production. If environments were tenant-wide, every project would be forced into the same environment set.

The Key Design Decision: FlagEnvironment

Here is the central insight that shapes the entire schema. A flag's *definition* (its name, key, type) is project-scoped. A flag's *state* (enabled/disabled, targeting rules, rollout percentage) is per-environment. You want to be able to say "the `dark-mode` flag is enabled in staging with no rules, but disabled in production."

This means you need a join entity between Flag and Environment. In Eloquent terms, it is a pivot table with extra columns. In data modeling terms, it is an associative entity that holds the per-environment configuration.

This entity – call it `FlagEnvironment` – holds: whether the flag is enabled in that environment, the targeting rules (as JSONB), the rollout percentage, and optionally an environment-level default value override.

Why not store the enabled/disabled state directly on the Flag? Because a flag is not "enabled" in the abstract. It is enabled in staging and disabled in production. The state is a property of the flag-environment *combination*, not the flag alone.

Why not store rules on the Flag with an environment discriminator? Because that would mean one row holds rules for all environments, requiring you to parse and filter within the JSON. Separate rows per environment means you can query directly: "give me the FlagEnvironment row for flag X in environment Y." One row, one read.

> **Coming from Laravel/Eloquent:** This is like `$flag->environments()->withPivot('enabled', 'rules', 'rollout_percentage')` in Eloquent. The difference in Prisma is that there is no implicit pivot table. You model `FlagEnvironment` as a first-class model with its own ID and relationships. This feels heavier at first but is actually better because you can query `FlagEnvironment` directly without going through the parent flag, which is exactly what the evaluation API needs to do.

Targeting Rules: JSONB vs. Normalized Tables

This deserves its own deep analysis because it is the most impactful data modeling decision in the system.

The normalized approach would look like: a `rules` table with a foreign key to `flag_environment`, a `rule_groups` table for AND/OR grouping, and a `conditions` table with `attribute`, `operator`, and `value` columns. Evaluating rules means joining three tables, reconstructing the tree in application code, then evaluating it.

The JSONB approach stores the entire rule tree as a JSON document in a single column on `FlagEnvironment`. Evaluating rules means reading one column from one row and evaluating the JSON directly.

Here is the trade-off analysis, factor by factor.

Read performance on the hot path. JSONB wins decisively. One row read of one column vs. a multi-table JOIN that assembles a tree from rows. The evaluation API is the bottleneck, and every extra JOIN adds latency and complexity.

Write performance. JSONB wins for the common case. Updating rules means one UPDATE statement replacing the column value. The normalized approach means deleting all existing con-

ditions and rule groups, then inserting the new ones – a DELETE followed by multiple INSERTs, ideally in a transaction.

Schema flexibility. JSONB wins. When you add a new operator like `semver_gte`, the JSONB approach requires no schema change – just update the application validation. The normalized approach requires adding the new operator to an enum column or a check constraint, which means a migration.

Data integrity. The normalized approach wins here. Relational constraints enforce that every condition has a valid operator, that every rule group has at least one condition, that values are of the right type. With JSONB, the database only knows it is valid JSON. Your application must validate the shape, and if a bug writes malformed rules to the database, the database will accept them silently.

Querying inside rules. The normalized approach wins for analytical queries like "how many flags target users in France?" With normalized tables, that is a standard WHERE clause on the conditions table. With JSONB, you need the `@>` containment operator or `jsonb_array_elements` to unnest the JSON, which is less intuitive and harder to optimize.

When would you choose normalized? If the evaluation hot path were not the bottleneck – for instance, if evaluation happened entirely client-side and the server just served configs. Or if you needed the database to enforce complex integrity rules on the rule tree. Or if you frequently queried across rules for analytics.

When is JSONB the right call? When the hot path reads the whole document, writes replace the whole document, and the shape will evolve faster than you want to run migrations. That is Flagline's exact situation.

The mental model: **JSONB is a document store inside your relational database.** Use it for data that behaves like a document – read as a unit, written as a unit, with a schema that you enforce in application code (with Zod or similar) rather than in the database.

API Keys: Scoping, Hashing, and the Prefix

API keys are how SDKs authenticate with the evaluation API. The design decisions here are driven by security requirements and the query pattern.

An API key is scoped to a **project + environment** combination. Not just a project, and not just an environment. The SDK installed in your production web app gets a key that can only read production flags for the web app project. This is the principle of least privilege: a compromised key in one environment cannot read flags from another environment.

When a key is created, you generate a random string, show it to the user once, and then store only the SHA-256 hash. The raw key is never stored. This is the same pattern as password hashing, but simpler (SHA-256 is fine because API keys have high entropy, unlike user passwords which need bcrypt/argon2).

You also store the first eight characters of the key in plaintext. This is the "prefix" that appears in the dashboard – `fl_a3b7c9...` – so users can identify which key is which without being able to reconstruct the full key.

The query pattern on the hot path is: SDK sends the raw API key in a header. The evaluation API hashes it and looks up the hash. One indexed lookup returns the project ID, environment ID, and

permissions. No join needed.

> **Coming from Laravel:** Laravel Sanctum follows a similar pattern – hash the token, store the hash, look up by hash on each request. The difference is that Sanctum uses a `personal_access_tokens` table that is user-scoped, while Flagline's API keys are project+environment scoped. The hashing and lookup logic is the same.

Audit Logs: Append-Only With Snapshots

Audit logs are append-only. You never update or delete an audit log entry. This is a compliance requirement in many contexts and a debugging lifesaver in all contexts.

Each audit log entry records: who did what, to which entity, when, and what changed. The "what changed" is stored as two JSONB columns: `before` (snapshot of the entity before the change) and `after` (snapshot after the change). This allows you to reconstruct the full history of any entity by reading its audit log entries in order.

Why JSONB snapshots instead of just storing the changed fields? Because it gives you a complete picture without needing to chain diffs backward from the current state. If someone asks "what did the rules for this flag look like last Tuesday?" you find the audit log entry closest to that time and read the `after` snapshot. You do not need to replay a series of incremental changes.

The index on audit logs is `(tenant_id, created_at DESC)` because the most common query is "show me the most recent actions for this tenant." Adding `project_id` to a second composite index supports "show me actions for this project" without scanning all tenant-wide entries.

Audit logs grow unboundedly. You will eventually need to think about partitioning this table by time range or moving old entries to cold storage. But that is an optimization for when the table reaches millions of rows, not a day-one concern.

Subscriptions: Keeping Stripe State in Sync

The subscription model exists to cache Stripe subscription state locally. Stripe is the source of truth for billing. Your database is a local mirror that you can query without making API calls to Stripe.

The model stores: the Stripe customer ID, subscription ID, price ID, plan tier, status, current billing period, and whether the subscription cancels at period end. All of this comes from Stripe webhooks and is updated whenever Stripe sends you an event.

Why not just call the Stripe API every time you need to check a user's plan? Because the evaluation API needs to know the plan to enforce limits (free tier gets 1,000 evaluations per day), and you cannot add 50ms of Stripe API latency to every evaluation request. The local subscription record, further cached in Redis, gives you sub-millisecond plan lookups.

The one-to-one relationship between Tenant and Subscription is enforced at the database level with a unique constraint on `tenant_id`. A tenant either has a subscription or is on the implicit free tier.

Unique Constraints as Business Rules

Some of the most important design decisions in the schema are not about tables or columns but about unique constraints. A unique constraint is a business rule enforced by the database, and

it prevents an entire class of bugs that would otherwise require application-level checks with race conditions.

The flag key must be unique within a project. This means you cannot accidentally create two flags named `dark-mode` in the same project, even if two dashboard users click "Create Flag" at the same instant. The database rejects the second INSERT. But it allows `dark-mode` in the web app project and `dark-mode` in the mobile app project, because they are scoped differently.

The environment slug must be unique within a project. The project slug must be unique within a tenant. These are all compound unique constraints – unique across a combination of columns, not a single column. The thinking pattern is: **"what would break if this value were duplicated, and what is the scope within which duplication would be a problem?"**

A common mistake is making things globally unique when they should be scoped. If flag keys were globally unique across all projects and tenants, customers would collide with each other's naming choices. If project slugs were globally unique, two different companies could not both have a "web-app" project.

> **Coming from Laravel:** These are equivalent to `$table->unique(['project_id', 'key'])` in a Laravel migration. Prisma expresses them with `@@unique([projectId, key])` on the model, which makes the business rules visible right next to the fields they constrain rather than buried in a migration file from six months ago.

Soft Deletion vs. Archival

Flagline does not use soft deletes in the Laravel sense (a `deleted_at` timestamp that filters queries). Instead, flags have an `is_archived` boolean. The thinking behind this choice is worth understanding.

Soft deletes (a la Laravel's `SoftDeletes` trait) hide records from default queries but keep them in the database. The problem is that soft deletes affect every query on the table – every `findMany` needs to exclude soft-deleted rows, and forgetting the filter brings back "deleted" records like ghosts.

Archival is a business concept, not a technical one. An archived flag is not "deleted." It is retired. Users can see it in the "Archived" section of the dashboard, review its history, and even restore it. The `is_archived` boolean communicates intent: this flag is no longer active, but it is not gone.

The practical difference: with soft deletes, you tend to add a global scope that hides deleted records everywhere. With an `is_archived` field, you explicitly choose where to filter and where not to. The evaluation hot path filters it out. The dashboard shows both, with a tab or toggle. There is no global scope hiding data – every query is explicit about whether it wants archived flags.

Prisma does not have built-in soft delete support (no `SoftDeletes` trait equivalent). This is intentional – Prisma prefers explicit queries over implicit behavior. If you want soft delete semantics, you implement them yourself with a Prisma Client Extension that adds the filter. For Flagline, the explicit `is_archived` approach is a better fit.

---

3. Indexing Strategy Thinking

The most common indexing mistake is to add indexes after performance problems appear. The second most common mistake is to add indexes on every column "just in case." Both are wrong. The right approach is to start from the queries you know you will run and work backward to the indexes you need.

Start From the Queries

Write down the queries before you write down the indexes. For Flagline, the critical queries are:

1. **Evaluation hot path:** Get all enabled flag configs for a given environment where the flag is not archived. This runs thousands of times per second.
2. **Dashboard flag list:** Get all flags for a project, optionally filtered by environment and archived status. This runs on every dashboard page load.
3. **API key lookup:** Find an API key by its hash. This runs on every evaluation request.
4. **Audit log pagination:** Get recent audit entries for a tenant, optionally filtered by project, newest first.
5. **Unique constraint checks:** Is this flag key already taken in this project? Is this slug already taken in this tenant?

Each of these queries tells you what index it needs.

Composite Index Column Ordering

The mental model for a composite index is a sorted phonebook. If the index is on `(last_name, first_name)`, the phonebook is sorted by last name first, then by first name within each last name. You can look up everyone named "Smith" efficiently (using just the first column). You can look up "Smith, Alice" efficiently (using both columns). You *cannot* look up everyone named "Alice" efficiently (using just the second column), because Alices are scattered throughout the phonebook next to their various last names.

Applied to Flagline: the index on `flag_environments(environment_id, enabled)` supports three query patterns: filtering by `environment_id` alone, filtering by `environment_id AND enabled`, and (in principle) range scans on `environment_id`. It does *not* efficiently support filtering by `enabled` alone, because `enabled=true` rows are scattered across all environment IDs in the index.

The rule of thumb: **put the most selective column first** (the one that narrows down the most rows), then add columns in order of decreasing selectivity. For the evaluation hot path, `environment_id` narrows to a few dozen rows at most (all flags in one environment), and `enabled` further narrows to the ones that are turned on. The index reads only the rows the query wants.

> **Coming from Laravel/MySQL:** The same rule applies in MySQL. If you have ever added a composite index in Laravel with `$table->index(['project_id', 'is_archived'])` and wondered why queries filtering only by `is_archived` did not use the index, this is why. Column order in composite indexes is not just a style choice; it determines which queries the index can serve.

Partial Indexes: When They Are Worth It

A partial index includes a `WHERE` clause, and only rows matching that clause go into the index. The classic use case is boolean flags where one value is much more common than the other, or where you only ever query one value.

In Flagline, you only query active (non-archived) flags on the hot path. You only query enabled flag environments. You only query non-revoked API keys. In all three cases, the "active" subset is what you care about at query time, and the "inactive" subset accumulates over time as dead weight in a regular index.

The thinking framework: **if your WHERE clause always includes a fixed predicate (like `is_archived = false`), and the rows matching that predicate are a minority of the table, a partial index gives you a smaller, faster, more memory-efficient index.** "Minority" does not need to be dramatic – even a 50/50 split means the index is half the size. But the benefit is biggest when 90% of rows are in the excluded state.

When are partial indexes *not* worth it? When you query both states equally often (the dashboard might list both active and archived flags). In that case, a regular index serves both queries, and a partial index only serves one. The answer is usually to have the regular index for the dashboard and the partial index for the hot path, because the hot path is where milliseconds matter.

GIN Indexes on JSONB: When You Need Them

A GIN (Generalized Inverted Index) index on a JSONB column indexes every key and value in the JSON document. It supports the `@>` containment operator ("does this document contain this sub-document?") and key existence checks.

You need a GIN index on the `rules` JSONB column if – and only if – you query *inside* the rules at the database level. The evaluation hot path does not do this; it reads the rules column and evaluates them in application code. But the dashboard might have an admin feature like "show me all flags that target the `country` attribute," which queries inside the JSON.

If you never query inside the JSON, do not add the GIN index. It slows down writes (every INSERT/UPDATE on that column must update the index) and uses disk space. Add it when you have a concrete query that needs it.

There are two flavors: `GIN (column)` indexes all operators (`@>`, `?`, `?|`, `?&`), while `GIN (column jsonb_path_ops)` only supports `@>` but is smaller and faster. If your only JSONB query is containment checks, use `jsonb_path_ops`. If you need key existence checks, use the default.

The Cost of Indexes

Every index speeds up reads but slows down writes. Each INSERT must update every index on the table. Each UPDATE that touches an indexed column must update the corresponding indexes. For a table that receives thousands of writes per second, adding five indexes means five extra write operations per row.

Flagline's `flag_environments` table is read-heavy on the hot path but written to only when someone changes flag configuration in the dashboard. This is a write-infrequent table. Indexes are essentially free here.

The `audit_logs` table is write-heavy (every action in the system creates an entry) and read-moderate (viewed in the dashboard). Be judicious with indexes here. The composite index for pagination is necessary. A GIN index on the `metadata` column is probably not worth it unless you have a specific query that needs it.

The `usage` table receives writes on every API call (aggregated via Redis counters and flushed periodically, but still). Keep indexes minimal.

Think about it as a budget: **each index costs write performance and storage. Spend that budget on the queries that matter most.**

---

4. Multi-Tenancy Thinking

Multi-tenancy is the hardest thing to get right in a SaaS database, not because it is technically complex, but because a single bug – one missed WHERE clause – can expose one customer's data to another. The architecture decision here is not just about performance; it is about how many layers of defense you put between tenants.

Row-Level Filtering vs. Schema-Per-Tenant

There are three common approaches to multi-tenancy in PostgreSQL.

Database-per-tenant: Each tenant gets their own PostgreSQL database. Complete isolation. But you now have N databases to manage, N connection pools, N migration runs. This makes sense for enterprise customers with strict data residency requirements. It does not make sense for a SaaS with hundreds of tenants on shared plans.

Schema-per-tenant: Each tenant gets their own PostgreSQL schema (namespace) within a single database. Tables are identical across schemas. Better isolation than row-level, but you still have N migration runs (one per schema), and connection pooling becomes complicated because you need to set the schema search path per connection. Some ORMs handle this well; Prisma does not handle it natively.

Row-level filtering: All tenants share all tables. Every table with tenant-specific data has a `tenant_id` column, and every query includes `WHERE tenant_id = ?`. Simplest operationally – one database, one migration run, one connection pool. The risk is in the "every query" part: forget the filter once, and you have a data leak.

For Flagline, row-level filtering is the right choice. The operational simplicity of a single database with a single set of migrations, running on a single connection pool, is worth the discipline of ensuring every query is tenant-scoped. The number of tenants will be in the hundreds or low thousands, not millions. The data per tenant is modest. There is no regulatory requirement for physical isolation.

The Danger: Forgetting the Filter

The fundamental problem with row-level tenancy is that it relies on application code to always include the tenant filter. A single `prisma.project.findMany()` without a `tenantId` in the WHERE clause returns *every tenant's projects*.

In Laravel, you might have used global scopes (`addGlobalScope('tenant', ...)`) or a package like `stancl/tenancy` that sets the scope automatically. The same pattern applies here, but in Prisma's idiom.

Safety Net 1: Prisma Client Extensions

The first line of defense is a Prisma Client Extension that automatically injects the `tenantId` filter into every query on tenant-scoped models. You create a function that takes a `tenantId` and returns a wrapped Prisma client where every `findMany`, `findFirst`, `create`, `update`, and `delete` on models like `Project`, `User`, and `AuditLog` automatically includes the tenant filter.

The mental model: this is like a Laravel global scope, but applied at the client level rather than the model level. You create a tenant-scoped client once per request (when you know who the logged-in user is and which tenant they belong to), and then every query through that client is automatically scoped.

The pattern is: `const db = withTenant(prisma, session.tenantId)` and then use `db` for all queries in that request. If you pass `db` to your service functions instead of the raw `prisma` client, you cannot accidentally skip the filter because the filter is built into the client itself.

> **Coming from Laravel:** Think of this as creating a dedicated Eloquent connection that has `->where('tenant_id', $tenantId)` baked into every query builder. In Laravel, global scopes serve this purpose. In Prisma, client extensions serve the same purpose with a different mechanism.

Safety Net 2: PostgreSQL Row Level Security

Row Level Security (RLS) is a database-level feature where PostgreSQL itself enforces row visibility based on a policy. Even if your application code has a bug and issues a query without a tenant filter, the database will filter the rows.

The setup is: enable RLS on each tenant-scoped table, create a policy that compares the row's `tenant_id` to a session variable, and then set that session variable at the start of each request using `SET LOCAL app.current_tenant_id = '...'`.

RLS adds a small per-query overhead because PostgreSQL evaluates the policy predicate on every row access. For Flagline, the Prisma extension is the primary defense (fast, application-level), and RLS is the secondary defense (slower, database-level) that you enable for compliance-sensitive deployments or as a paranoia-driven safety net.

The mental model: **the Prisma extension is the lock on your front door. RLS is the alarm system. You want both, but you rely on the lock for daily life.**

The Pattern: Every Query is Scoped by Default

The guiding principle is that tenant scoping should be opt-in to *remove*, not opt-in to *add*. If a developer writes a query and forgets to think about tenancy, it should be safe by default (scoped to the current tenant), not unsafe by default (returning all tenants' data).

This means: raw `prisma` should only be used in system-level operations like data migrations, background jobs that process across tenants, or seed scripts. All request-handling code should

use the tenant-scoped client.

If you find yourself writing `prisma.project.findMany({ where: { tenantId } })` frequently, with the tenant filter added manually each time, that is a smell. It means the scoping is not automatic and someone will eventually forget it.

---

5. The Prisma Mental Model

If you are coming from Eloquent, Prisma will feel alien for the first few days and then refreshingly direct once the mental model clicks. The key is that Prisma is *not* an Active Record ORM.

Not Active Record, Not Data Mapper – Just a Query Builder

In Eloquent, a model is a class that represents a row. `$user = User::find(1)` gives you a `User` object with methods like `$user->save()`, `$user->delete()`, `$user->projects`. The model knows how to persist itself.

In Prisma, there is no `User` class. There is no `Flag.find(1)`. Instead, you have a client object (`prisma`) with methods on it for each model: `prisma.user.findUnique(...)`, `prisma.flag.create(...)`. The return value is a plain JavaScript object, not a class instance. It has no methods. It does not know how to save itself.

This feels like a step backward until you realize what you gain: the return type of every query is *exactly* what you asked for, and TypeScript knows its shape at compile time.

The Return Type Depends on What You Selected

This is the most important mental model shift. In Eloquent, `User::find(1)` always returns a `User` with all columns. If you want only specific columns, you use `->select(['name', 'email'])`, but the return type is still `User` – your IDE does not know that `created_at` is missing.

In Prisma, the return type changes based on your `select` and `include` clauses. If you write `prisma.user.findUnique({ where: { id }, select: { name: true, email: true } })`, the return type is `{ name: string; email: string }`, not `User`. TypeScript will error if you try to access `createdAt` on the result. This is enforced at compile time.

The `include` clause eagerly loads relations. `prisma.flag.findUnique({ where: { id }, include: { flagEnvironments: true } })` returns a `Flag & { flagEnvironments: FlagEnvironment[] }`. Without the include, the `flagEnvironments` property does not exist on the return type. Not just at runtime – at *compile time*.

> **Coming from Laravel:** In Eloquent, you access `$user->projects` and it lazy-loads the relation if not eager-loaded, returning an empty collection or the actual data. This is convenient but hides N+1 queries. In Prisma, there is no lazy loading. If you did not `include` the relation, it is not on the object, and your code will not compile. This forces you to think about your data access pattern at write time rather than discovering N+1 issues in production logs.

The Migration Workflow Difference

In Laravel, you write migrations by hand: `Schema::create('flags', function (Blueprint $table) { ... })`. You decide the SQL. The migration *is* the source of truth.

In Prisma, the `schema.prisma` file is the source of truth. You describe the desired state of your database – the models, their fields, their relationships, their indexes. Then you run `prisma migrate dev`, and Prisma generates the SQL migration by diffing your schema file against the current database state.

The mental model is declarative vs. imperative. In Laravel, you say "add this column, then add this index, then modify that constraint." In Prisma, you say "the Flag model should have these fields and these indexes." Prisma figures out what SQL is needed to get from the current state to the desired state.

You can (and should) review the generated SQL before applying it. Sometimes Prisma generates a DROP + ADD when you meant a RENAME. In those cases, you use `--create-only` to generate the migration file without applying it, edit the SQL by hand, and then apply it.

How `@@map` and `@map` Work

Prisma model names follow TypeScript convention (PascalCase): `FlagEnvironment`. Database table names follow SQL convention (snake_case): `flag_environments`. The `@@map("flag_environments")` directive on the model tells Prisma to use that table name. Similarly, `@map("tenant_id")` on a field maps `tenantId` in TypeScript to `tenant_id` in the database.

This mapping means your TypeScript code always uses camelCase (`fe.environmentId`, `fe.rolloutPercentage`) while your database columns are snake_case (`environment_id`, `rollout_percentage`). The translation is invisible in your application code.

> **Coming from Laravel:** Eloquent does this implicitly – model properties are camelCase when accessed as attributes, and columns are snake_case by convention. Prisma makes it explicit with `@map`/`@@map`, which is more verbose but leaves nothing to convention or magic.

CUIDs vs. Auto-Increment IDs

Flagline uses CUIDs (Collision-resistant Unique Identifiers) as primary keys instead of auto-incrementing integers. A CUID looks like `clx7abcde0001...` – a string that is roughly sortable by creation time, globally unique, and does not reveal how many records exist (unlike integer IDs where `id=42` tells you there are at most 42 records).

The trade-off: CUIDs are larger (25+ characters vs. 8 bytes for a bigint), which makes indexes larger and joins slightly slower. But they enable one critical pattern: generating the ID on the client side before the INSERT. The application creates the ID, uses it in the INSERT statement, and can return it immediately without a database round-trip to read the generated ID. This matters for transactions and for building related objects before persisting them.

In Prisma, `@default(cuid())` generates the CUID in the Prisma client, not in the database. The ID is set before the query is sent to PostgreSQL.

The Json Type and Its Quirks

Prisma's `Json` type maps to PostgreSQL's `jsonb`, but the TypeScript type it returns is `JsonValue` – which is essentially `unknown`. Prisma cannot know at compile time what shape the JSON has, because the database stores arbitrary JSON.

This means every time you read a `Json` field, you must validate and cast it before using it. In Flagline, the `rules` field is validated with a Zod schema. You read the raw `JsonValue`, parse it through `FlagRulesSchema.safeParse(raw)`, and get back either a typed `FlagRules` object or a validation error.

There is also a subtle distinction between `null` and JSON null. When a `Json?` field is SQL NULL (the column has no value), Prisma represents it as `Prisma.DbNull`. When the column contains the JSON literal `null` (the string "null" stored as JSONB), Prisma represents it as `Prisma.JsonNull`. JavaScript `null` is ambiguous and Prisma rejects it for `Json` fields to avoid confusion.

This trips up every developer coming from Eloquent, where `$model->rules` being `null` means "the column is NULL" and there is no distinction. In Prisma, you need to use `Prisma.DbNull` explicitly when checking for SQL NULL on a JSON field.

The Singleton Pattern for PrismaClient

In Next.js development mode, hot module reloading creates a new PrismaClient instance on every file save. Each instance opens its own connection pool. After a dozen saves, you have exhausted your database's connection limit.

The fix is a singleton pattern: store the PrismaClient on `globalThis` so that it survives hot reloads. The pattern is universally used in the Next.js + Prisma ecosystem. It is the equivalent of Laravel's service container binding – the container ensures only one database connection manager exists. In Next.js, you manage this yourself.

> **Coming from Laravel:** Laravel's service container handles connection lifecycle automatically. `DB::connection()` always returns the same connection. In Next.js, you are the service container. The singleton pattern is your way of ensuring connection reuse. This is a one-time setup – write it once, put it in `lib/db.ts`, and import from there everywhere.

---

6. The Evaluation Hot Path

The evaluation hot path is the query that runs on every flag evaluation – every time an SDK asks "what value should this flag have for this user?" Everything in the database and cache layers should be designed to make this path fast, because it is the one path that directly affects your customers' application performance.

What Data Does the Evaluation Need?

When an SDK client sends an evaluation request, it provides: its API key (which maps to a project + environment), and optionally a user context (attributes like `userId`, `email`, `country`, `plan`).

The evaluation API needs to return: every active, enabled flag for that environment, with its flag key, flag type, targeting rules, rollout percentage, and default value.

Notice what it does *not* need: flag descriptions, who created the flag, when it was last updated, audit history, or anything about other environments. The evaluation response is a narrow slice of the data model, optimized for the SDK to evaluate flags locally.

The Database Query

The database query for the hot path joins two tables: `flag_environments` (for the per-environment state) and `flags` (for the flag key and type). It filters by: `environment_id = ?`, `enabled = true`, `is_archived = false`. It selects only the columns the SDK needs.

This query is served by the composite index on `flag_environments(environment_id, enabled)` and the index on `flags(project_id, is_archived)`. With proper indexing, it completes in under 1ms for a project with a hundred flags.

But under 1ms is still too slow to run on every evaluation request if you have thousands of requests per second. A hundred concurrent evaluation requests each taking 1ms of database time means a hundred connections occupied simultaneously. The connection pool becomes the bottleneck before CPU or memory do.

The Cache-Aside Pattern With Redis

The solution is to cache the evaluation response in Redis. The pattern is called "cache-aside" (sometimes "lazy loading"):

1. SDK request comes in.
2. Check Redis for the cached response.
3. Cache hit: return the cached data. Done. No database involved.
4. Cache miss: query PostgreSQL, store the result in Redis with a TTL, return the result.

This means the database is only hit once per TTL interval per environment, regardless of how many evaluation requests come in. If the TTL is 60 seconds and you get 10,000 requests per second for one environment, the database sees one query per minute instead of 600,000.

What Triggers Cache Invalidation?

There are two invalidation strategies, and you want both.

TTL-based expiry: Every cached entry has a time-to-live (60 seconds for flag configs). After 60 seconds, the entry disappears from Redis, and the next request triggers a fresh database query. This provides a guaranteed upper bound on staleness: flag changes are visible within 60 seconds even if active invalidation fails.

Active invalidation on flag changes: When someone changes a flag in the dashboard, the API explicitly deletes the relevant Redis keys. This means changes propagate immediately (on the next request) rather than waiting for TTL expiry.

Why both? Because neither alone is sufficient. Active invalidation alone means that if the invalidation message is lost (Redis network blip, application crash between database write and cache delete), the cache serves stale data forever. TTL alone means that flag changes take up to 60

seconds to propagate, which is fine for most cases but annoying when you are testing changes in staging and want to see them immediately.

The flow after a dashboard flag change is: write to PostgreSQL, delete the Redis cache keys for the affected environment, publish a Pub/Sub event for connected SSE clients. The next evaluation request finds no cache, queries the database, caches the fresh result.

Thinking About TTL Duration

The TTL is a dial between freshness and database load. A 5-second TTL means changes propagate within 5 seconds but the database gets a query every 5 seconds per environment. A 5-minute TTL means the database is barely touched but changes take up to 5 minutes to propagate without active invalidation.

For flag configs (60 seconds): this is the sweet spot. SDK polling intervals are typically 30-60 seconds, so even without active invalidation, the SDK picks up changes within two polling cycles. The database sees at most one query per minute per environment.

For API key metadata (5 minutes): API keys change very rarely (created once, maybe revoked once). A 5-minute cache is fine because revocation also triggers active invalidation.

For tenant plan information (10 minutes): Plan changes are extremely rare (maybe once a month per tenant). The 10-minute cache minimizes database queries for limit checks.

The mental model: **the TTL should match the rate of change of the data.** Frequently changing data gets short TTLs. Rarely changing data gets long TTLs. All data gets active invalidation as the fast path.

Connection Pooling and the Hot Path

One detail that is easy to overlook: database connections are a finite resource. A typical PostgreSQL instance allows 100 to 300 concurrent connections. Prisma manages a connection pool, and each application instance has its own pool.

If the evaluation API has 4 instances each with a pool size of 20, that is 80 connections. If the dashboard has 2 instances each with a pool size of 10, that is 20 more. You are at 100 connections, which is approaching the default limit.

The cache-aside pattern with Redis is not just about latency. It is about connection pressure. Without caching, every evaluation request occupies a database connection for the duration of the query (~1ms). At 10,000 requests per second, that is 10 connections occupied at any given instant just for evaluations. With caching, evaluations almost never touch the database, and the connection pool is free for dashboard queries, webhook processing, and background jobs.

This is why the cache layer is not optional for the evaluation path. It is not a “nice to have for performance.” It is a requirement for the system to function at scale without exhausting database connections.

> **Coming from Laravel:** In Laravel, the default MySQL connection pool is managed per-process by PHP-FPM. Each worker gets one connection, and the connection is released when the request ends. This is simpler but also more wasteful (no connection

sharing between requests). In the Node.js world, the event loop handles many concurrent requests on a single process, and the connection pool is shared across all of them. This makes pool management more important – and caching more impactful.

Thinking About What the SDK Receives

One more consideration for the hot path: the shape of the evaluation response should be optimized for what the SDK needs to do with it.

The SDK needs to evaluate targeting rules locally. It receives the full rule tree as JSON and evaluates it against the user context on the client side. This means the server does not need to evaluate rules – it just needs to serve them. The server's job is data serving, not data processing.

This architectural decision (server serves configs, client evaluates them) is why the evaluation query can be so simple. The server does not need to know the user context. It does not need to JOIN against a users table or compute hash-based rollout bucketing. It just reads flag configs from cache (or database on cache miss) and returns them.

If you were doing server-side evaluation (the server evaluates rules against the user context and returns just the flag values), the hot path would be more complex: you would need the user context in the query, and the response would be per-user rather than per-environment. The cache would be less effective because it would need to be keyed per user, not per environment.

Both architectures are valid. LaunchDarkly supports both. For Flagline, client-side evaluation is the default because it makes the server stateless with respect to users, makes caching trivial (cache per environment), and pushes evaluation compute to the client.

---

7. Migration and Evolution Thinking

The schema you deploy on day one will not be the schema you have on day 365. Features will be added, fields renamed, tables restructured. The question is not whether the schema will change but how you will change it safely in production.

The "Always Additive" Principle

The safest schema change is an additive one: adding a new column, a new table, or a new index. These changes do not affect existing queries or existing application code. The old code does not know about the new column and does not care. The new code can start using it.

The dangerous changes are subtractive (removing a column, removing a table) and transformative (renaming a column, changing a column's type, adding a NOT NULL constraint to a column that has NULL values).

The principle: **make additive changes, then backfill, then make the constraint.** Want to add a required column? Add it as optional first, deploy the code that writes to it, backfill existing rows, then make it required. Never try to do all three in one migration on a table that has data.

The Three-Step Column Addition

This pattern comes up repeatedly and is worth internalizing.

Step one: add the column as optional (nullable, no default). This migration is safe because no existing code reads or writes this column, and no existing rows need a value for it. Deploy the migration. Deploy application code that writes to the new column.

Step two: run a data migration that backfills existing rows. This is a standalone script, not a Prisma migration. It updates rows in batches to avoid locking the table. Once complete, every row has a value for the new column.

Step three: alter the column to be required (NOT NULL with a default). This migration is safe because every row already has a value (from the backfill) and new rows will get the default.

If you try to skip steps and add a NOT NULL column with no default in one migration, it will fail on a table with existing rows. PostgreSQL will reject the ALTER TABLE because it cannot fill in the missing values.

> **Coming from Laravel:** The same principle applies in Laravel migrations, but you might be used to `$table->string('description')->nullable()` followed by a backfill and then `$table->string('description')->nullable(false)->change()`. The Prisma workflow is the same three steps, just expressed differently: edit the schema to add the optional field, run `prisma migrate dev`, run the backfill script, edit the schema to make the field required, run `prisma migrate dev` again.

Development vs. Production Migration Mindset

In development, you can freely destroy and recreate the database. `prisma migrate reset` drops everything and rebuilds from scratch. `prisma db push` applies schema changes directly without creating migration files. This is fast and convenient for iterating on schema design.

In production, every migration must be safe. You cannot DROP a column that the running application is reading. You cannot ALTER a column type while active queries reference it. You cannot add an index on a table with 10 million rows without considering the lock time (`CREATE INDEX CONCURRENTLY` is your friend, but Prisma does not generate it by default – edit the migration SQL).

The mental shift: **development is a blank slate, production is a live surgery.** In development, move fast. In production, be paranoid. Review every generated migration SQL. Ask yourself: "will this lock the table? Will this break running queries? What happens if the deploy rolls back halfway through?"

Prisma's Approach to Rollbacks

Prisma does not have a `migrate:rollback` command. This is a deliberate design choice. Rollbacks in production are dangerous because they reverse a schema change without reversing the data that was written under the new schema. If you added a column, wrote data to it, and then rolled back (dropping the column), that data is lost.

The recommended approach is forward migration: if a migration was wrong, write a new migration that corrects it. If you added a column you should not have, write a migration that drops it. This is explicit, reviewable, and does not silently lose data.

> **Coming from Laravel:** `php artisan migrate:rollback` is comfortable because each migration has a `down()` method. But have you ever actually rolled back a production migration? In practice, rollbacks are a development convenience, not a production safety mechanism. Prisma makes the production reality the default behavior.

When You Need Custom SQL in Migrations

Prisma's schema language cannot express everything PostgreSQL supports. Partial indexes, GIN indexes, Row Level Security policies, custom functions, and triggers all require raw SQL. The workflow is:

1. Run `prisma migrate dev --create-only --name add_custom_indexes` to create an empty migration file.
2. Write the SQL by hand in the generated `migration.sql` file.
3. Run `prisma migrate dev` to apply it.

The custom SQL lives alongside the auto-generated migrations in the `prisma/migrations/` directory. Prisma tracks it in its migration history and will not re-apply it.

---

8. Redis as a Complement to PostgreSQL

Redis is not a replacement for PostgreSQL. It is a complement. The mental model is simple: **PostgreSQL is the source of truth. Redis is the fast-access layer.**

If Redis goes down, your system is slower but not broken. The evaluation API falls back to database queries. Rate limiting becomes permissive (or you fail open). SSE connections reconnect and refetch state. Nothing is lost, because Redis never held the authoritative copy of any data.

If PostgreSQL goes down, your system is broken. No flag evaluations (once the Redis cache expires), no dashboard, no audit logs, no API key validation. PostgreSQL is the foundation. Redis is the performance layer on top.

What Goes in Redis vs. What Goes in Postgres

The decision framework is straightforward.

Postgres holds: anything that must survive a restart, anything that has complex relational integrity, anything that is the source of truth. Flag definitions, targeting rules, API keys, user accounts, subscriptions, audit logs.

Redis holds: derived data that can be recomputed from Postgres (caches), ephemeral counters (rate limits), real-time coordination (pub/sub), and anything where microsecond access matters more than durability.

A concrete test: if you `FLUSHALL` Redis (delete everything), does the system recover on its own? If yes, the data belongs in Redis. If no, it belongs in Postgres.

The Three Uses of Redis in Flagline

Caching flag configs. The evaluation API caches the full set of flag configs per environment. This turns a database query into a Redis GET, reducing latency from ~1ms to ~0.1ms and reducing database connections from thousands to single digits. The cache key includes the project ID and environment ID, making invalidation precise.

Pub/Sub for real-time updates. When a flag changes in the dashboard, the API publishes a message to a Redis channel. All API server instances subscribed to that channel receive the message and push it to connected SSE clients. Redis Pub/Sub is fire-and-forget – if no one is listening, the message is lost. That is fine because clients that miss a message will pick up changes on their next poll or cache refresh.

Rate limiting. Each API key has a rate limit. Redis's `INCR` command is atomic and O(1), making it perfect for counting requests in a time window. The key includes the time window boundary, so it self-expires when the window passes. No cleanup job needed.

Key Naming Conventions

Redis keys are strings with no inherent structure, so structure must be imposed by convention. The convention in Flagline is colon-delimited namespaces starting with `flagline:`.

The pattern: `flagline:{scope}:{identifier}:{data-type}`.

For example, `flagline:{projectId}:{envId}:flags` caches all flag configs for an environment. `flagline:apikey:{hash}` caches API key metadata. `flagline:ratelimit:{apiKeyId}:{window}` tracks request counts.

Why does this matter? Because Redis has no tables or collections. Every key lives in one flat namespace. Without a naming convention, you end up with keys like `flags_prod_123` and `user:42:settings` and `rate-limit-abc` – inconsistent, hard to discover, impossible to batch-delete by pattern.

With the namespace convention, you can use `SCAN` with a pattern like `flagline:*:flags` to find all flag config caches, or `flagline:ratelimit:*` to inspect all rate limit counters. The convention makes Redis inspectable.

> **Coming from Laravel:** If you have used Laravel's cache with the `prefix` config, the concept is the same. Laravel prefixes all cache keys with `laravel_cache:` by default to avoid collisions. Flagline's convention is more explicit and hierarchical, but the purpose is identical.

TTL Strategy Thinking

Every piece of data in Redis should have a TTL (time-to-live) unless there is a good reason for it to persist indefinitely. The risk of TTL-less data is that it accumulates silently until Redis runs out of memory.

The two exceptions in Flagline are SSE connection counters (managed by explicit increment/decrement, cleaned up by a periodic reconciliation job) and any data you are using as a distributed lock (which has a short TTL as a safety mechanism, but needs to persist for the lock duration).

For cache data, the TTL is the maximum acceptable staleness. Ask yourself: if this data were 60 seconds stale, would anyone notice? For flag configs, probably not – most flag changes are not urgent to the millisecond. For API key revocation, yes – a revoked key should stop working within seconds, which is why revocation triggers active invalidation rather than waiting for TTL expiry.

For rate limit counters, the TTL matches the window. A 60-second rate limit window has a 60-second TTL on the counter key. When the window passes, the key expires, and the count resets. No cron job needed.

Redis Data Structure Choices

Redis offers several data structures, and choosing the right one for each use case matters.

Strings (the simplest: key-value) are used for cached flag configs (a JSON string) and rate limit counters (an integer that you `INCR`). Strings are right when you always read and write the whole value.

Hashes (a key with named fields) are used for cached API key metadata. The advantage over a JSON string is that you can read individual fields without parsing the whole document: `HGET flagline:apikey:{hash} permissions` returns just the permissions, not the entire key metadata. This matters when different code paths need different fields.

Pub/Sub (fire-and-forget messaging) is used for flag change notifications. It is not a data structure in the storage sense – messages are not persisted. If no subscriber is listening, the message is gone. This is fine for SSE push because clients that miss a message will reconcile on their next poll.

The mental model: **use the simplest structure that serves the access pattern.** If you read and write the whole blob, use a String. If you need field-level access, use a Hash. If you need ordered data or scoring, use a Sorted Set. If you need broadcast messaging, use Pub/Sub.

Final Thought: The Schema Is a Conversation

The schema you design today is a hypothesis about how the system will be used. The evaluation hot path hypothesis ("we need to serve all flags for an environment, cached, sub-millisecond") drives the JSONB decision, the FlagEnvironment join entity, the composite indexes, and the Redis caching layer. The multi-tenancy hypothesis ("hundreds of tenants sharing one database, row-level filtering") drives the tenant ID on every table, the Prisma extension, and the RLS policies.

As the system evolves – as you add new flag types, new targeting operators, percentage-based rollouts with sticky bucketing, A/B test analytics, scheduled flag changes – the schema will evolve with it. The thinking in this document does not give you the final schema. It gives you the framework for making schema decisions as the product changes.

Every schema change should start with the question: "What query does this change serve, and how often does that query run?" If the answer is "the evaluation hot path, thousands of times per second," you optimize for read performance above all else. If the answer is "an admin report that runs once a day," you optimize for correctness and simplicity over speed.

The database is not a dumping ground for data. It is the backbone of the system's behavior. Every table, every column, every index is a decision about what the system values. Make those decisions consciously.