Contents

04 – Node.js / Backend API Thinking Guide

> **Flagline** – Feature Flag SaaS Stack: React, Next.js 14+ (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 code reference. It is a thinking guide – mental models, decision frameworks, and the reasoning behind every major architectural choice in the Flagline evaluation API. When a small inline snippet helps illustrate a concept, it appears, but the emphasis is always on the "why" and "how to think about it," not the "copy this block."

Table of Contents

---

1. Why a Separate Evaluation API

The Decision Framework

The first question a backend developer asks when looking at Flagline's architecture is: "Why not just put everything in Next.js API routes? You already have a server there." The answer requires

thinking about the two fundamentally different types of traffic this product serves.

Dashboard traffic is human-driven. A team of 5-10 people clicks through the UI, creates flags, edits targeting rules, reviews audit logs. This traffic is bursty (someone opens a page, fires a few requests, then reads for a while), low-volume (tens of requests per minute at peak), and tolerance for latency is generous – 200-500ms is perfectly fine for a dashboard page load.

Evaluation traffic is machine-driven. Every user session in every customer application that has integrated Flagline's SDK fires evaluation requests. A single customer with 100,000 daily active users generates hundreds of requests per second. This traffic is steady, high-volume, and extremely latency-sensitive – the SDK call sits in the critical path of the customer's application, so every millisecond matters.

> **Coming from Laravel:** Think about the difference between your admin panel and a public API endpoint that mobile apps hit on every screen load. You would never want a spike in admin traffic to slow down the mobile API, and vice versa. In Laravel, you might solve this with separate route groups and rate limits. In the Node.js/cloud world, the solution is separate services with independent deployment and scaling.

The Serverless Problem

Next.js on Vercel runs in a serverless model. Each request spins up a function instance (or reuses a warm one), does its work, and returns. This model has three properties that conflict with evaluation API requirements:

Cold starts. A serverless function that has not been invoked recently takes 100-500ms to spin up before it can even begin handling the request. For a dashboard page, this is invisible. For an evaluation call in a customer's hot path, it is unacceptable.

Execution time limits. Vercel imposes a maximum execution time (30 seconds for streaming, less for non-streaming). The evaluation API needs to hold SSE connections open indefinitely – minutes, hours, as long as the SDK is running. A serverless function cannot do this.

No shared in-process state. Each serverless invocation is isolated. You cannot keep a Redis subscription active across requests, maintain an in-memory connection pool, or hold a map of active SSE clients. Every invocation starts from scratch.

A traditional long-running Node.js process has none of these constraints. It boots once, keeps connections warm, holds state in memory between requests, and runs until you tell it to stop.

Where to Draw the Boundary

The mental model for the boundary is simple: anything a human interacts with through the dashboard stays in Next.js. Anything an SDK calls programmatically lives in the evaluation API.

Next.js handles: - Dashboard pages (server components, server actions) - Webhook receivers (Stripe, auth callbacks) - Internal revalidation endpoints - Marketing pages, docs, blog

Evaluation API handles: - `POST /v1/evaluate` – single and batch flag evaluation - `GET /v1/flags/stream` – SSE connection for real-time updates - `GET /v1/flags` – fetch all flag configs for an environment (SDK bootstrap)

The communication between the two is mediated by the shared database and Redis pub/sub. When a dashboard user changes a flag, the dashboard writes to PostgreSQL and publishes an event to Redis. The evaluation API, subscribed to that Redis channel, picks up the change and pushes it to connected SDKs. The two services never call each other directly in the hot path.

Express vs Fastify – The Decision Framework

Both Express and Fastify are mature Node.js HTTP frameworks. The question is not which is "better" in the abstract, but which fits the constraints of this specific service.

Express is the default choice for most Node.js developers. It has the largest ecosystem, the most middleware, the most Stack Overflow answers. If you are building a general-purpose API where developer familiarity and ecosystem breadth matter more than raw performance, Express is the safe pick.

Fastify is designed for performance-critical services. Its key advantages:

First, Fastify uses JSON Schema for request/response validation and serialization. The schema is compiled to optimized serialization functions at startup, which means response serialization is dramatically faster than `JSON.stringify()`. For an evaluation API returning JSON on every request, this matters.

Second, Fastify's plugin system enforces encapsulation. Each plugin gets its own scope, which prevents the middleware-ordering bugs that plague Express applications. In Express, the order in which you call `app.use()` determines behavior, and a misplaced middleware can silently break authentication for an entire route group. Fastify's encapsulated plugins eliminate this class of bug.

Third, Fastify provides built-in logging via Pino, which produces structured JSON logs. Express requires you to add Morgan or Winston and configure them yourself. For a production service where structured logging is non-negotiable, having it built in reduces setup friction.

Fourth, Fastify natively supports async/await in route handlers without the unhandled rejection issues that Express has. In Express, an unhandled promise rejection in a route handler silently hangs the request. Fastify catches it and returns a 500. This alone prevents an entire category of production incidents.

> **Coming from Laravel:** Fastify's plugin system is conceptually similar to Laravel's service providers – each plugin registers its own routes, decorators, and hooks in an encapsulated scope, and the framework composes them into the final application. Express is more like a raw middleware stack where order is everything.

For Flagline, the decision is Fastify. The evaluation API is a high-throughput, latency-sensitive service where the performance characteristics of the framework actually matter. This is not a CRUD app where the framework overhead is negligible compared to database queries.

---

2. API Design Thinking

The Principle: Minimize the Surface, Maximize Clarity

The evaluation API has a tiny surface area by design. It does one thing – evaluate feature flags – and it exposes the minimum number of endpoints needed to do that well. Every endpoint that

exists must justify its existence. Every endpoint that does not exist avoids a maintenance burden, a documentation page, and a potential security surface.

The complete endpoint list:

```
POST /v1/evaluate          Evaluate one or many flags
GET  /v1/flags             Fetch all flag configs for bootstrap
GET  /v1/flags/stream      SSE stream for real-time updates
GET  /health               Health check (liveness)
GET  /ready                Readiness check
```

Five endpoints. That is the entire API. An SDK can fully function with just the first and third.

Why POST for Evaluation

A developer's first instinct might be `GET /v1/flags/dark-mode?userId=abc&plan=pro`. After all, evaluation is a read operation – you are asking "what value should this flag have for this user?" GETs are cacheable, idempotent, and feel semantically correct for a read.

But consider what the evaluation context looks like in practice. A customer's user context might include a userId, an email, a plan tier, a country, an app version, custom attributes, and any number of fields relevant to their targeting rules. Encoding all of this in query parameters creates several problems.

URL length limits are real. Browsers enforce approximately 2,000 characters; some proxies and CDNs enforce even less. A rich evaluation context can easily exceed this.

Query parameters are logged by default. Access logs, CDN logs, load balancer logs – they all capture the full URL. If the evaluation context contains a user's email or any PII, it ends up in log files across the infrastructure. POST body content is not logged by default.

Cache poisoning risk is high. If you make evaluation results cacheable via GET, a CDN or intermediate proxy might cache a response for one user and serve it to another. The evaluation context is user-specific, so caching at the HTTP level is actively harmful.

POST with a JSON body solves all three. The context lives in the body, not the URL. It does not leak into logs. And POST responses are not cached by intermediate proxies unless you explicitly opt in.

> **Coming from Laravel:** This is the same reasoning behind why Laravel's form submissions use POST even for search forms with complex filters. When the input is rich and potentially sensitive, the body is the right place for it.

Why Batch Evaluation Matters

When an SDK initializes, it needs the values of every flag for the current environment. If the project has 50 flags, that is either 50 individual requests or one batch request. The math is obvious: one round-trip is better than fifty.

But the argument goes deeper than network efficiency. Batch evaluation lets the server load all flag configurations for the environment once, evaluate them all against the same context, and return a single response. The database query and Redis lookup happen once, not per flag. Server-side, the cost of evaluating 50 flags in memory after loading the config is negligible compared to the

cost of 50 separate request/response cycles, each with its own authentication, rate limiting, and connection overhead.

The API supports both single and batch evaluation through the same endpoint. If `flagKeys` is provided, evaluate those specific flags. If omitted, evaluate all flags for the environment. This eliminates the need for a separate "evaluate single flag" endpoint.

The Response Envelope

Every response from the evaluation API follows the same envelope structure: a `success` boolean, a `data` field with the payload, an optional `error` field, and optional `meta`. This consistency is not just aesthetic – it is a contract that SDK authors can rely on.

When the SDK parses a response, it should never need to inspect the HTTP status code to determine the shape of the response body. A 200 always has `{ success: true, data: ... }`. A 400 or 500 always has `{ success: false, error: { code, message } }`. The SDK's HTTP client can have a single response parser regardless of whether the request succeeded or failed.

This pattern also makes the API self-documenting in error cases. Instead of returning a bare string like `"Invalid API key"`, the error includes a machine-readable `code` (like `AUTH_INVALID_KEY`) that SDKs can switch on, and a human-readable `message` for debugging.

API Versioning

Flagline uses URL-based versioning: `/v1/evaluate`, `/v1/flags`. The alternative is header-based versioning (`Accept: application/vnd.flagline.v1+json`), but URL versioning wins for this use case because of how SDKs are distributed.

When a customer upgrades their SDK from version 1.x to 2.x, the SDK internally switches from `/v1/evaluate` to `/v2/evaluate`. The URL makes the version visible in logs, monitoring, and debugging. You can see at a glance which version a request is targeting without inspecting headers. You can route different versions to different backend deployments at the load balancer level.

The trade-off is that URL versioning conflates the resource identifier with the API contract. In theory, `/flags/dark-mode` is the same resource whether you are using v1 or v2 of the API. In practice, for a machine-to-machine API where the only consumers are SDKs that you control, this theoretical purity matters less than operational clarity.

The rule of thumb for when to create a v2: only when you make a breaking change to the response shape that existing SDKs cannot handle. Adding new fields to a response is not breaking. Removing fields or changing their types is. Changing the semantics of an existing field is the most dangerous because it is not detectable by type checking.

---

3. Authentication Mental Model

API Keys, Not Sessions

The evaluation API uses API key authentication, not session-based authentication. This is a fundamental design choice that follows from the nature of the consumer.

Session-based auth (cookies, JWTs tied to a login flow) is designed for humans interacting through a browser. The user logs in, gets a session token, and the browser sends it automatically on every request. Sessions are short-lived, require a refresh mechanism, and are tied to a specific user's identity.

API key auth is designed for machine-to-machine communication. The SDK is configured with an API key at initialization and sends it on every request. There is no login flow, no refresh, no session expiry. The key is long-lived, scoped to a project and environment, and identifies the tenant, not a user.

> **Coming from Laravel:** This is the difference between `auth()->user()` (session-based, from a logged-in browser) and Laravel Sanctum/Passport API tokens (sent in a header, identifying an application). The evaluation API exclusively uses the latter pattern. There is no session middleware, no CSRF protection, no cookie handling.

The Key Format as Documentation

Flagline API keys follow a self-documenting format: `fl_live_` for production keys and `fl_test_` for development/staging keys. The prefix serves three purposes.

First, it makes keys visually distinguishable. When a developer accidentally uses a test key in production, the prefix makes this obvious in logs and debugging output. You do not need to look up the key in a database to know what environment it targets.

Second, it enables secret scanning. GitHub, GitLab, and other platforms can detect leaked secrets by scanning for known prefixes. A key that starts with `fl_live_` is identifiable as a Flagline production key without any ambiguity.

Third, it enables quick validation before any database or cache lookup. If a key does not start with `fl_`, it is not a Flagline key – reject it immediately without touching Redis or PostgreSQL. This is a cheap guard that protects the hot path from garbage input.

The Lookup Chain

When a request arrives with an API key, the authentication flow resolves it through a chain:

```
key -> hash(key) -> Redis lookup -> (on miss) PostgreSQL lookup -> project + environment + tenant
```

The key itself is never stored. On creation, the full key is shown to the user once and then only the SHA-256 hash is persisted. This means that if the database is compromised, the attacker has hashes, not usable keys.

The resolved metadata tells you everything you need for the request: which project this key belongs to, which environment (production, staging, development), which tenant (for billing and rate limiting), and what permissions the key has (evaluate, server-side, admin).

Why Redis Sits in This Path

The API key lookup happens on every single evaluation request. If you go to PostgreSQL every time, that is a database round-trip on the hottest path in the system. With a 5-minute TTL cache in Redis, the vast majority of requests resolve the key from memory in under a millisecond.

The mental model: the first request with a given key pays the PostgreSQL cost. Every subsequent request within the 5-minute window pays only the Redis cost. Given that SDKs typically make requests every few seconds, the cache hit rate for active keys approaches 100%.

The risk of caching is staleness. If a key is revoked in the dashboard, it could remain valid in the Redis cache for up to 5 minutes. This is why revocation explicitly deletes the Redis cache entry – `invalidateApiKey(keyHash)` is called as part of the revocation flow, not left to TTL expiry.

> **Coming from Laravel:** This is exactly the same as using `Cache::remember("api_key:{$hash}", 300, fn() => ApiKey::where('key_hash', $hash)->first())`. The pattern is identical; only the syntax differs.

Key Rotation Thinking

A well-designed API key system supports having multiple active keys for the same project/environment. The reason: key rotation. When a customer needs to rotate a key (because it was accidentally committed to a public repo, or as part of a security policy), they should be able to create a new key and then revoke the old one. If only one key can exist at a time, rotation requires a coordinated, zero-downtime switch across all deployed instances of the customer's application. That is unreasonable to expect.

With multiple active keys, the rotation workflow becomes: create new key, deploy application with new key, verify it works, revoke old key. Each step can happen independently, on its own timeline.

The Trust Boundary

API key authentication establishes a trust boundary: the key proves that the request is authorized to evaluate flags for a specific project and environment. Everything beyond that boundary is untrusted input.

The evaluation context (userId, attributes) comes from the customer's application. It is user- supplied data. You do not trust it for anything beyond what it claims – you validate its shape (is it a valid object with string/number/boolean attribute values?), but you do not verify that the userId actually exists or that the attributes are accurate. The customer's application is responsible for passing correct context; the evaluation API is responsible for evaluating rules against whatever context it receives.

This is a deliberate design choice. The alternative – the evaluation API verifying user identity – would require access to the customer's user database, which is neither feasible nor desirable.

---

4. The Flag Evaluation Engine – Algorithmic Thinking

The Core Mental Model

The evaluation engine is the heart of the product. Everything else – the API layer, the caching, the authentication – exists to support this one function: given a flag configuration and a user context, determine what value to return.

Think of it as a function with a clean signature: `evaluate(flagConfig, userContext) -> value`. No side effects, no network calls, no database queries. The config was already loaded

(from Redis or PostgreSQL) and the context was already provided in the request. The evaluation itself is pure computation.

This purity is the most important design decision in the engine. It means evaluation is fast (no I/O), testable (no mocks needed for the core logic), and deterministic (same inputs always produce the same output).

The Priority System: First Match Wins

A flag's targeting rules are an ordered list. The engine walks the list from top to bottom and returns the value of the first rule that matches the user context. If no rule matches, it falls through to the default value.

This is the same mental model as CSS specificity, firewall rules, or Nginx location blocks. Order matters. The first match wins.

> **Coming from Laravel:** Think of this like middleware priority in the HTTP kernel, or like a chain of `if/elseif/else` blocks. The first condition that evaluates to true determines the outcome. Everything after it is ignored.

Why ordered rules instead of weighted rules, priority scores, or a most-specific-match algorithm? Simplicity and predictability. A user looking at the rules list in the dashboard can read them top to bottom and understand exactly what will happen. "Internal users get the experimental version. Enterprise users get the stable version. 10% of everyone else gets the beta. Everyone else gets the default." The mental model is immediately obvious.

The alternative – a specificity-based system like CSS – creates a debugging nightmare. When two rules both match and the user cannot predict which one wins without understanding the specificity algorithm, you have turned a simple product into a puzzle. Feature flags are operational tooling. They should be boring and predictable.

Consistent Hashing for Percentage Rollouts

A 25% rollout does not mean "randomly select 25% of requests." It means "deterministically assign 25% of users to the enabled group, and keep them there forever (or until the rollout percentage changes)."

The "deterministically" part is critical. If user Alice is in the 25% on her first page load, she must be in the 25% on every subsequent page load. If the evaluation were random, Alice would see the feature appear and disappear unpredictably, which is worse than not having the feature at all.

The mechanism is consistent hashing. Take the flag key and the user ID, concatenate them, hash the result with MurmurHash3 (a fast, well-distributed non-cryptographic hash), take the result modulo 100, and compare to the rollout percentage.

```
bucket = murmurhash3(flagKey + userId) % 100
isEnabled = bucket < rolloutPercentage
```

Why MurmurHash3 and not SHA-256? Speed. The evaluation engine runs this hash for every percentage rollout rule on every request. MurmurHash3 is designed for hash tables and bucketing – it is fast (nanoseconds) and has excellent distribution properties. SHA-256 is a cryptographic

hash designed for security properties we do not need here. Using SHA-256 would be like using a sledgehammer to hang a picture frame.

Why concatenate the flag key with the user ID? So that the same user gets different assignments for different flags. If you hashed only the userId, user Alice would either be in the rollout for every flag or out of the rollout for every flag. Adding the flag key as a salt ensures that Alice's bucket for "dark-mode" is independent of her bucket for "new-checkout-flow."

Why modulo 100? It maps the hash output to a number between 0 and 99, which directly corresponds to a percentage. A rollout of 25% enables the flag for users whose bucket is 0-24.

> **Coming from Laravel:** If you have implemented A/B testing or canary deployments, this is the same idea. The key insight is that "random" and "consistent" are different things. A percentage rollout must be consistent – the same user gets the same result every time – which means you need a deterministic function (a hash), not a random number generator.

When Percentage Changes

If you increase a rollout from 25% to 50%, all users who were in the 25% remain in the 50%. The new 25% is additive – users with buckets 25-49 are now included, but users with buckets 0-24 were already included and remain so. No one loses access to a feature when you widen the rollout.

This property falls out naturally from the "bucket < percentage" comparison. If your bucket is 17, you were in at 25% and you are still in at 50%. This is not something you need to implement specially – it is an inherent property of the algorithm.

The one caveat: if you decrease the rollout (50% back to 25%), users with buckets 25-49 lose the feature. This is expected and correct – you are narrowing the rollout – but it is worth surfacing in the dashboard UI so that the person making the change understands the impact.

Attribute Matching and Compound Conditions

Each targeting rule contains conditions that compare user attributes to expected values. A condition has three parts: the attribute name (what to look at), the operator (how to compare), and the value (what to compare against).

The operator set needs to cover the common cases without becoming a query language. Flagline supports: `eq`, `neq`, `in`, `nin`, `gt`, `gte`, `lt`, `lte`, `contains`, `startsWith`, `endsWith`, `exists`, and semver comparisons (`semver_gt`, `semver_gte`, `semver_lt`, `semver_lte`). Each operator is a simple function that takes two values and returns a boolean.

Conditions within a rule group are combined with AND or OR logic. An AND group requires all conditions to be true. An OR group requires at least one. Groups can be nested – an AND group can contain an OR group, which can contain another AND group – enabling arbitrarily complex targeting logic.

In practice, most rules are flat AND groups: "user is in the US AND on the Pro plan AND created after January 1st." Deeply nested boolean trees are rare, but the engine supports them because the implementation cost is trivial (recursive evaluation of a tree structure) and the flexibility is occasionally needed.

The critical thinking about attribute matching is type safety. When a condition says `attribute: "age", op: "gt", value: 30`, the user context might provide age as a string (`"25"`) or a number (25). The engine must handle type coercion gracefully: attempt to compare as the expected type, and if coercion fails, treat the condition as non-matching. Never crash on a type mismatch.

The Fallback Chain

When the engine evaluates a flag, it walks a fallback chain:

1. **Is the flag enabled?** If the `enabled` field on the FlagEnvironment is `false`, return the default value immediately with reason `FLAG_DISABLED`. Do not evaluate rules.
2. **Do any rules match?** Walk the rules array top to bottom. If a rule's conditions match the user context, return that rule's value with reason `TARGETING_MATCH`.
3. **Does the percentage rollout include this user?** If a rollout rule exists and the user's bucket falls within the percentage, return the rollout value with reason `TARGETING_MATCH`.
4. **Environment default.** If the FlagEnvironment has a `defaultValue` override, return it.
5. **Flag default.** Return the flag's project-level `defaultValue`.
6. **Hardcoded fallback.** If everything else somehow fails (corrupt data, missing config), return `false` for boolean flags, `""` for string flags, `0` for number flags. This is the last resort that ensures the SDK always gets a usable value.

The principle behind this chain: an evaluation must never fail. It must always return a value and a reason. The SDK should never receive an error that forces it to decide what to do. The "safe default" principle means that when in doubt, the feature is off. A feature that is accidentally disabled is a minor inconvenience. A feature that is accidentally enabled for the wrong users is a potential security or data integrity incident.

> **Coming from Laravel:** Think of this like a chain of null coalescing: `$value = $ruleMatch ?? $rolloutMatch ?? $envDefault ?? $flagDefault ?? false`. Each step is a fallback. The chain always terminates with a safe value.

---

5. Redis Thinking

Redis Wears Three Hats

In Flagline's architecture, Redis serves three distinct roles, and it is important to think about each one separately because they have different failure modes, different performance characteristics, and different operational concerns.

Hat 1: Cache. Redis stores flag configurations and API key metadata so the evaluation API can serve requests without hitting PostgreSQL on every call. This is the classic cache-aside pattern.

Hat 2: Message Bus. Redis pub/sub broadcasts flag change events from the dashboard to the evaluation API, which then pushes updates to connected SSE clients. This is a fire-and-forget broadcast channel.

Hat 3: Rate Limiter. Redis counters enforce per-key and per-tenant rate limits using atomic increment operations. This requires Redis's atomic primitives but not its persistence.

Each hat has a different answer to the question "what happens if Redis goes down?"

Cache Thinking: What to Cache, How Long, and What Happens on Miss

The cache-aside pattern works like this: the evaluation API checks Redis for the flag configs for the requested environment. If the data is there (cache hit), use it. If not (cache miss), query PostgreSQL, use the result, and write it to Redis for the next request.

What to cache. The entire flag configuration for an environment, serialized as a single JSON string. Not individual flags – the whole set. The reason: the most common operation is batch evaluation (evaluate all flags at once), so caching the complete set avoids N cache lookups for N flags.

API key metadata is cached separately as a Redis hash. Hashes let you read individual fields (`HGET flagline:apikey:{hash} permissions`) without deserializing the entire object, which is useful when you only need to check permissions.

TTL strategy. Flag configs have a 60-second TTL. API keys have a 300-second (5-minute) TTL. These numbers are not arbitrary.

For flag configs: SDK polling intervals are typically 30-60 seconds. A 60-second cache means that in the worst case, a flag change takes 60 seconds to propagate via the cache (though pub/sub propagates it much faster – the cache TTL is the backup mechanism). Making the TTL shorter increases database load without meaningfully improving freshness, because the pub/sub channel handles real-time propagation.

For API keys: keys are created or revoked infrequently. A 5-minute cache is aggressive, but safe because revocation explicitly invalidates the cache entry. The 5-minute TTL is a safety net for edge cases (like a Redis pipeline failure during explicit invalidation), not the primary invalidation mechanism.

Cache miss behavior. On a cache miss, the evaluation API queries PostgreSQL and populates the cache before returning. This means the first request after a cache expiry or a cold start is slower (by the cost of a PostgreSQL round-trip), but all subsequent requests within the TTL are fast.

The important nuance: the cache miss path must be non-blocking for other requests. In Node.js, this happens naturally because database queries are asynchronous. While one request is waiting for PostgreSQL, the event loop continues serving other requests from the cache. This is one of the cases where Node.js's async model is genuinely advantageous over a synchronous model.

> **Coming from Laravel:** This is `Cache::remember()` with a TTL. The pattern is identical. The difference is that in Laravel, if the cache is cold and 100 requests arrive simultaneously, each PHP-FPM worker independently queries the database (the "thundering herd" problem). In Node.js, you can use a request coalescing pattern where the first request initiates the query and subsequent concurrent requests wait for the same promise to resolve.

Pub/Sub Thinking: The Real-Time Propagation Path

When a dashboard user toggles a flag, the following chain executes:

1. The dashboard writes the change to PostgreSQL.
2. The dashboard publishes a flag change event to the Redis channel for that environment.
3. Every evaluation API instance subscribed to that channel receives the event.
4. Each instance invalidates its local cache for that environment.
5. Each instance pushes the update to all connected SSE clients for that environment.

The key insight: pub/sub is not a replacement for the cache. It is a complement. Pub/sub provides near-instant notification of changes (milliseconds). The cache provides a fallback for cases where pub/sub messages are missed (process restart, network blip, Redis reconnection). Together, they give you real-time propagation with eventual consistency as a safety net.

A subtle but important point about Redis pub/sub: it is fire-and-forget. If an evaluation API instance is not connected to Redis at the moment a message is published, it misses that message forever. There is no message queue, no replay, no acknowledgment. This is acceptable because the cache TTL ensures that even if a pub/sub message is missed, the cache will expire and the fresh data will be loaded from PostgreSQL within 60 seconds.

If you need stronger delivery guarantees in the future (guaranteed delivery, message ordering, replay), Redis Streams would be the upgrade path. But for Flagline's current requirements, pub/sub's simplicity and low overhead are the right trade-off.

> **Coming from Laravel:** Redis pub/sub is similar to Laravel's event broadcasting with the Redis driver. The `broadcastOn()` method specifies a channel, and listeners on that channel receive the event. The same fire-and-forget semantics apply – if no one is listening, the event vanishes.

Rate Limiter Thinking: Atomic Counters

Rate limiting requires three properties: atomicity (no race conditions between checking and incrementing), speed (it runs on every request), and self-cleaning (expired windows should not accumulate forever).

Redis provides all three. The `INCR` command is atomic – even under high concurrency, two simultaneous requests will get different counter values. The operation is O(1) and completes in microseconds. And the `EXPIRE` command ensures that old window counters are automatically deleted.

The fixed-window algorithm works by dividing time into windows (say, 60-second windows) and counting requests per window. The Redis key includes the window timestamp, so a new window automatically starts a new counter. When the window expires, the TTL deletes the key.

The trade-off of fixed windows versus sliding windows: at the boundary between two windows, a burst of requests can momentarily exceed the intended rate. If the limit is 100 per minute and a client sends 100 requests in the last second of one window and 100 in the first second of the next window, they effectively sent 200 requests in 2 seconds. For Flagline's use case, this edge case is acceptable. A sliding window algorithm (using Redis sorted sets) eliminates this but adds complexity. Section 7 discusses this in more detail.

6. SSE Architecture Thinking

Why SSE, Not WebSockets

Server-Sent Events and WebSockets both enable the server to push data to the client without polling. The choice between them depends on the communication pattern.

WebSockets provide a full-duplex channel – the client and server can send messages in both directions at any time. This is the right choice for chat applications, collaborative editors, or any scenario where the client needs to send data to the server through the persistent connection.

SSE provides a unidirectional channel – the server pushes events to the client, and the client receives them. The client cannot send data back through the SSE connection. If the client needs to send data, it uses a separate HTTP request.

For Flagline, the communication is strictly one-directional. The server pushes flag change events to the SDK. The SDK never needs to send data back through the event stream. The SDK's evaluation requests use a separate `POST /v1/evaluate` endpoint.

SSE has three practical advantages over WebSockets for this use case:

Simpler protocol. SSE is plain HTTP. It uses a standard `GET` request with `Accept: text/event-stream`, and the response is a long-lived stream of text lines. There is no upgrade handshake, no frame encoding, no ping/pong protocol. The server writes lines like `data: {"flagKey":"dark-mode"}\n\n` and the client receives them.

Better proxy compatibility. Many corporate proxies, load balancers, and CDNs handle SSE correctly because it is regular HTTP. WebSockets require an HTTP Upgrade handshake that some proxies do not support or misconfigure. In a B2B SaaS where customers run your SDK behind corporate firewalls, this matters.

Automatic reconnection. The browser's `EventSource` API (and well-implemented SDK equivalents) automatically reconnects when the connection drops, and sends a `Last-Event-ID` header so the server can resume from where it left off. WebSockets require the application to implement reconnection logic manually.

> **Coming from Laravel:** If you have used Laravel's Broadcasting with Pusher or Ably, SSE is a simpler version of the same concept – the server pushes events to connected clients. The difference is that SSE runs directly on your API server with no third-party service in between.

The Connection Lifecycle

The mental model for an SSE connection:

1. **Client connects.** The SDK sends `GET /v1/flags/stream` with the API key in the `Authorization` header. The server authenticates the key, resolves the environment, and holds the connection open.
2. **Server subscribes.** The server subscribes to the Redis pub/sub channel for that environment. Any flag change published to that channel will be forwarded to this client.

3. **Idle waiting.** Most of the time, nothing happens. The connection is open but silent. The server periodically sends a heartbeat (a comment line `:ping\n\n`) to keep the connection alive through proxies.
4. **Flag change.** A dashboard user toggles a flag. The pub/sub channel fires. The server receives the event and writes it to the SSE stream as a `data:` line. The client receives it and updates its local flag state.
5. **Disconnect.** The client closes the connection (page unload, app shutdown, network drop). The server detects the disconnect, unsubscribes from pub/sub for that environment (if no other clients are connected), and cleans up.

The Fan-Out Problem

Consider: one flag change in one environment needs to be delivered to every SDK client connected to that environment. If 10,000 SDKs are connected, that is 10,000 writes to 10,000 open connections. This is the fan-out problem.

In practice, the fan-out is bounded by the number of connections per evaluation API instance. If you run 5 instances of the API behind a load balancer, each instance holds roughly 2,000 connections. The Redis pub/sub message is received by all 5 instances (Redis pub/sub broadcasts to all subscribers), and each instance handles its own 2,000 connection writes.

The writes themselves are cheap – writing a few hundred bytes to a TCP socket is a microseconds-level operation. The bottleneck is not the write but the I/O multiplexing. Node.js handles this well because the event loop can queue thousands of socket writes without blocking. This is one of the architectural reasons Node.js was chosen – its non-blocking I/O model is purpose-built for this kind of concurrent connection handling.

Connection Management and Memory

Each open SSE connection consumes a small amount of memory on the server: a file descriptor, a buffer, and the metadata you associate with it (environment ID, API key ID, connection timestamp). On a typical server, this is roughly 10-50 KB per connection, meaning 10,000 connections consume 100-500 MB.

The important discipline is cleanup. When a client disconnects, you must: - Remove the connection from your tracking map. - Decrement the SSE connection counter in Redis. - If this was the last connection for an environment, unsubscribe from the pub/sub channel.

If you fail to clean up, you accumulate zombie connections that consume memory and distort your connection count metrics. In Node.js, you detect client disconnection by listening for the `close` event on the response object or the request's `AbortSignal`.

The Heartbeat / Keep-Alive Problem

Proxies, load balancers, and even some operating systems will terminate idle TCP connections after a timeout (typically 60-120 seconds). If no data flows through an SSE connection for this period, an intermediate proxy may close it, and neither the server nor the client will immediately know.

The solution is a periodic heartbeat. The server sends a comment line (`:ping\n\n`) every 15-30 seconds. SSE comments (lines starting with `:`) are ignored by the client but keep the TCP connection alive through proxies. This is not application-level data – it is purely a keep-alive mechanism.

The heartbeat interval should be shorter than the shortest proxy timeout in your infrastructure. If Cloudflare has a 100-second idle timeout and your load balancer has a 60-second timeout, your heartbeat must be under 60 seconds.

> **Coming from Laravel:** PHP's process-per-request model makes long-lived connections impractical. Each PHP-FPM worker holding an SSE connection open would consume an entire OS process for the lifetime of the connection. With a pool of, say, 50 workers, you could handle 50 SSE connections before the server stops responding to other requests. This is why Laravel uses external services (Pusher, Ably, Soketi) for real-time features – they offload the connection management to a purpose-built server. Node.js handles this natively because connections are just entries in the event loop, not dedicated processes.

---

7. Rate Limiting Thinking

Who to Rate Limit and Why

Rate limiting in a multi-tenant SaaS serves two audiences: it protects the system from abuse (intentional or accidental), and it enforces billing tiers.

Per API key is the primary rate limit axis. Each API key maps to a project, an environment, and a tenant. The rate limit is determined by the tenant's billing plan (Free: 1,000 evaluations/day, Starter: 100,000/day, Pro: 1,000,000/day). This is the limit that matters for billing and fair usage.

Per IP is a secondary defense against abuse. If someone is brute-forcing API keys or sending garbage requests without valid authentication, per-IP rate limiting catches them before the request even reaches the authentication layer. This limit is generous (much higher than per-key limits) because it is a safety net, not a billing mechanism.

Per tenant (aggregated across all keys) is relevant when a tenant has multiple API keys. Their total usage across all keys should not exceed their plan limit. This is more of an accounting concern than a real-time enforcement concern – you check it periodically rather than on every request.

The Sliding Window Mental Model

The fixed-window algorithm divides time into discrete buckets (e.g., 60-second windows). The problem with fixed windows is the boundary condition: at the edge of two windows, a burst can temporarily double the effective rate.

The sliding window algorithm eliminates this by weighting the previous window's count. Imagine two consecutive 60-second windows:

- Window 1 (ending now): 80 requests
- Window 2 (current, 15 seconds in): 30 requests
- You are 25% through the current window (15/60)
- Estimated rate = 80 * (1 - 0.25) + 30 = 90

This weighted estimate gives a smoother rate calculation without the computational cost of tracking individual request timestamps.

For Flagline, the fixed-window approach is used initially because it is simpler and the boundary condition is acceptable for evaluation rate limits. The difference between "1,000 requests per minute" and "occasionally allowing 1,200 at a window boundary" is not meaningful for this use case. If precise rate limiting becomes a requirement (e.g., for a metered billing model), the sliding window approach using Redis sorted sets is the upgrade path.

Rate Limits and Billing Tiers

The mapping from plan to rate limit is defined in a shared constants file (`src/lib/billing/plans.ts`) that both the dashboard and the evaluation API reference. This is the single source of truth. The evaluation API reads the tenant's plan from the cached API key metadata and looks up the corresponding limit.

When a request is rate limited, the response must communicate three things: - HTTP status 429 (Too Many Requests) – the status code that well-behaved clients understand. - `Retry-After` header – how many seconds until the window resets. - `X-RateLimit-Limit`, `X-RateLimit-Remaining`, `X-RateLimit-Reset` headers – the total limit, how many remain, and when the window resets.

These headers are not just for human debugging. Well-built SDKs read them and implement automatic backoff. Flagline's own SDKs respect `Retry-After` and exponentially back off when rate limited.

> **Coming from Laravel:** Laravel's `ThrottleRequests` middleware does exactly this – it uses Redis to track request counts per key and returns 429 with `Retry-After` and `X-RateLimit-*` headers. The mental model is the same. The difference is that Laravel's throttle middleware is generic (rate limit any route), while Flagline's rate limiting is tightly integrated with billing tiers and API key metadata.

---

8. Error Handling Philosophy

Errors as Data, Not Exceptions

In the evaluation API, errors are not exceptional – they are expected data. A request with an invalid API key is not a surprise; it is a normal part of operating a public API. A database timeout is not a catastrophe; it is a condition the system is designed to handle gracefully.

The mental model: every possible failure mode has a planned response. The API never returns an unstructured error or a stack trace. It always returns the standard envelope with a machine-readable error code and a human-readable message.

Error Categories

Client errors (4xx) are caused by the caller. Bad API key, malformed request body, missing required fields, rate limit exceeded. These are the caller's responsibility to fix. The response should be clear enough that the developer can fix the issue without contacting support.

Server errors (5xx) are caused by the system. Database down, Redis timeout, out of memory, unhandled exception. These are your responsibility to fix. The response should be generic ("An internal error occurred") because exposing implementation details in error messages is a security risk.

The critical design principle for the evaluation API: **server errors should be invisible to the end user.** If your database goes down, the customer's application should not break. The evaluation API should serve from cache, serve defaults, or serve the last known good state. The customer's dashboard might show stale data, but their application keeps running.

Graceful Degradation: The Fallback Pyramid

Think about the dependencies in a typical evaluation request:

```
Ideal:          Redis cache (fastest)
Fallback 1:     PostgreSQL (slower, but authoritative)
Fallback 2:     Stale cache (expired TTL, but still in memory)
Fallback 3:     Default values (hardcoded, always available)
```

If Redis is down, the evaluation API falls back to PostgreSQL. If PostgreSQL is also down, and there is stale cached data in application memory, use it (it is better than nothing). If there is no cached data at all, return the flag's default value. The evaluation must always succeed. The caller must always receive a response.

This philosophy extends to every dependency:

- **Redis down for caching?** Query PostgreSQL directly. Latency increases but correctness is preserved.
- **Redis down for pub/sub?** SSE clients stop receiving real-time updates, but they continue polling via the evaluate endpoint. No data is lost – just delayed.
- **Redis down for rate limiting?** Allow the request through. It is better to over-serve than to reject legitimate requests because you cannot check the rate limit counter. Log the rate limiter failure so you know it happened.
- **PostgreSQL down?** Serve from Redis cache (even if stale). If the cache is empty, serve defaults.

  Coming from Laravel: This is like wrapping every Cache or DB call in a try-catch with a sensible fallback, but elevated to a system-wide design principle. In Laravel, you might do `try { Cache::get(...) } catch { DB::query(...) }`. The same pattern applies here, but you think about it upfront as part of the architecture, not as an afterthought.

Structured Logging

In a production Node.js service, logs are consumed by machines (log aggregation services like Datadog, Grafana Loki, CloudWatch), not by humans reading terminal output. This means logs must be structured JSON, not formatted strings.

Fastify uses Pino by default, which outputs one JSON object per log line. Each log entry includes a timestamp, a log level, a request ID, and any context you attach. The request ID is the single most

important field – it lets you trace a single request across all the log entries it generated, including across service boundaries if you propagate it in headers.

The mental model for what to log: - **Always log:** Request received (with method, path, key prefix), request completed (with status code, latency), errors (with stack trace), rate limit events, cache miss events. - **Never log:** Full API keys, user PII from evaluation context, full request/response bodies in production (they are too large and may contain sensitive data). - **Conditionally log:** Slow queries (over a threshold), evaluation rule matches (for debugging, toggled via config), pub/sub events.

Health Checks: Liveness vs Readiness

Two health check endpoints serve different purposes:

Liveness (`GET /health`) answers: "Is the process alive and able to accept HTTP requests?" This check should always return 200 if the server is running. It does not check dependencies. If this fails, the orchestrator (Kubernetes, Fly.io, Railway) should restart the process.

Readiness (`GET /ready`) answers: "Is the process ready to serve real traffic?" This check verifies that critical dependencies are reachable: can we connect to Redis? Can we connect to PostgreSQL? If this fails, the load balancer should stop sending traffic to this instance, but the orchestrator should not restart it (the dependency might recover on its own).

The distinction matters for deployment. During a deployment, a new instance starts (passes liveness) but has not yet warmed its Redis cache or established database connections (fails readiness). The load balancer should not route traffic to it until readiness passes. Meanwhile, the old instances continue serving.

> **Coming from Laravel:** Laravel's health check is typically a single `/health` route that returns 200. The liveness/readiness distinction comes from container orchestration and is less common in traditional PHP deployments. If you have deployed to Kubernetes, you have seen `livenessProbe` and `readinessProbe` – this is the same concept.

---

9. The Async / Event-Driven Mental Model

The Fundamental Shift from PHP

PHP processes one request at a time, synchronously. When a PHP-FPM worker hits a database query, it blocks – the entire process waits for the database to respond. To handle concurrent requests, you run multiple workers (typically 10-50 per server), each in its own OS process. Concurrency is achieved through parallelism: multiple processes, each handling one request.

Node.js uses a single thread with an event loop. When a Node.js server hits a database query, it registers a callback and immediately returns to processing other work. The database query happens in the background (via the operating system's async I/O facilities), and when it completes, the callback is placed in the event loop's queue. Concurrency is achieved through asynchronicity: one process, handling many requests by never waiting.

This is not just a technical difference – it changes how you think about server capacity.

A PHP server with 50 FPM workers can handle 50 concurrent requests. If each request takes 50ms, you can handle roughly 1,000 requests per second. If each request takes 500ms (slow database query), you can handle 100 requests per second. The throughput is directly tied to worker count and response time.

A Node.js server can handle thousands of concurrent connections because holding an open connection costs almost nothing when the server is not actively computing for that connection. The bottleneck is CPU time: how long the single thread spends doing actual computation (parsing JSON, evaluating rules, hashing) versus how long it spends waiting for I/O (database queries, Redis lookups).

Why This Matters for SSE

This architectural difference is why SSE is practical in Node.js and impractical in PHP.

An SSE connection is 99.9% idle. The client connects, and the server holds the connection open. Occasionally (when a flag changes), the server writes a few bytes. The rest of the time, the connection just… exists. In Node.js, an idle connection is just an entry in the event loop's tracking structure – it consumes a file descriptor and a small buffer, but no CPU and no thread. A single Node.js process can hold 10,000 idle SSE connections without breaking a sweat.

In PHP, each SSE connection would occupy an entire FPM worker for the lifetime of the connection. A server with 50 workers could serve 50 SSE clients, and then it would be unable to handle any other requests. This is why the PHP ecosystem uses external services (Pusher, Ably, Centrifugo) for real-time features.

> **Coming from Laravel:** If you have ever tried to implement long-polling or SSE in Laravel, you know the pain. You either used `set_time_limit(0)` and occupied a worker indefinitely, or you used Pusher/Ably. Node.js eliminates this entire class of problem by design.

The Practical Implications

The async model has concrete implications for how you write code:

No blocking operations. In PHP, `file_get_contents()` or `DB::select()` blocks until completion, and that is fine because each worker only serves one request. In Node.js, you must never call a synchronous I/O function (like `fs.readFileSync()` or a CPU-intensive computation without yielding) because it blocks the event loop and freezes every concurrent request.

Callbacks, Promises, async/await. The historical evolution of Node.js async patterns went from callbacks (nested, error-prone) to Promises (chainable, composable) to async/await (synchronous-looking syntax over Promises). Modern Node.js code exclusively uses async/await. When you write `const result = await prisma.flagEnvironment.findMany(...)`, it looks synchronous but is actually yielding control to the event loop while the query runs.

Error handling with async/await. In synchronous code, unhandled exceptions propagate up the call stack. In async code, an unhandled rejection in a Promise used to silently vanish (in older Node.js versions) or crash the process (in newer versions). Fastify handles this correctly for route handlers – if your async handler throws, Fastify catches it and returns a 500. But for background

work (pub/sub handlers, cleanup callbacks), you must explicitly catch errors because there is no framework wrapping those contexts.

The event loop as a bottleneck. The evaluation engine (rule matching, hashing for percentage rollouts) runs synchronously on the event loop. If evaluation takes 1ms and you receive 5,000 requests per second, that is 5 seconds of CPU time per second – the event loop is saturated. In practice, evaluation is fast enough (sub-millisecond for most flag configurations) that this is not a concern. But it is the mental model you should carry: CPU-bound work on the event loop is the scalability limit of a Node.js service.

If you ever encounter a computation-heavy scenario (evaluating thousands of complex rules), the escape hatch is to move that work to a worker thread. Node.js's `worker_threads` module runs JavaScript in a separate thread with its own event loop, communicating with the main thread via message passing. But for Flagline's evaluation engine, this is premature optimization.

---

10. Testing Thinking

The Evaluation Engine: Pure Logic, Perfect for Unit Tests

The evaluation engine is the most important code to test and the easiest to test well. It is a pure function: input in (flag config + user context), output out (evaluation result), no side effects. You do not need mocks, stubs, or a running database. You construct a config object, construct a context object, call the evaluate function, and assert on the result.

The test cases for the evaluation engine should cover:

Basic evaluation. Flag enabled, no rules, returns default value. Flag disabled, returns default with `FLAG_DISABLED` reason.

Rule matching. A rule with an `eq` condition matches when the attribute equals the value. A rule with an `in` condition matches when the attribute is in the values list. Each operator gets its own test case.

Rule priority. When multiple rules match, the first one wins. When no rules match, the default value is returned. When a rule matches but the flag is disabled, the disabled state takes precedence.

Percentage rollouts. A 50% rollout assigns approximately 50% of a set of test user IDs to the enabled group. The same user ID always gets the same result (consistency). Different flag keys produce different assignments for the same user (independence).

Compound conditions. An AND group with three conditions matches only when all three are true. An OR group matches when any one is true. Nested groups evaluate correctly.

Edge cases. Missing attributes in the user context (condition should not match, not crash). Type mismatches (string attribute compared with a number operator). Empty rules array (returns default). Null rules (returns default). Rollout at 0% (no one gets it). Rollout at 100% (everyone gets it).

> **Coming from Laravel:** Think of this like testing a Laravel `Rule` class or a service class that has no Eloquent dependencies. You instantiate it, call its method, and assert the

return value. No `RefreshDatabase` trait, no HTTP test helpers – just pure PHP unit tests. The evaluation engine is the same.

Testing Percentage Rollout Distribution

A common mistake is testing that a specific user ID gets a specific result. This makes the test fragile because it depends on the hash implementation. Instead, test the statistical properties: feed 10,000 random user IDs through a 50% rollout and assert that between 45% and 55% land in the enabled group. This validates the distribution without coupling to specific hash outputs.

Also test consistency separately: evaluate the same user ID 100 times and assert that all 100 results are identical. This validates determinism.

And test independence: evaluate two different flag keys for the same user ID. Assert that the results are not correlated (they may happen to be the same, but over many users, the assignments should be independent).

Integration Testing the API

The evaluation engine's unit tests give you confidence in the core logic. Integration tests give you confidence that the HTTP layer, authentication, serialization, and error handling work correctly end-to-end.

The approach: start the Fastify server in-process (without binding to a port) and use Fastify's `inject()` method to send synthetic requests. This avoids the overhead of actual network calls while exercising the full request lifecycle: routing, authentication, validation, evaluation, serialization, response.

```
fastify.inject({ method: "POST", url: "/v1/evaluate", headers: { ... }, payload: { ... } })
```

This single-line approach replaces the need for a separate test HTTP client. The request goes through every middleware, plugin, and hook that a real request would, but entirely in-process.

> **Coming from Laravel:** This is equivalent to `$this->postJson('/api/evaluate', $payload)` in a Laravel feature test. The HTTP kernel processes the request, but no actual HTTP connection is made. Fastify's `inject()` serves the same purpose.

Where to Draw the Mock Boundary

The question "what do I mock?" has a clear answer in Flagline's architecture: mock at the service boundary.

Mock Redis at the client level. Provide a mock `ioredis` instance that returns predefined values for `get`, `hgetall`, `incr`, etc. This lets you test the caching logic, the rate limiting logic, and the pub/sub subscription logic without a running Redis server.

Mock Prisma at the client level. Prisma supports creating a mock client that returns predefined query results. This lets you test the database fallback path (what happens when Redis misses and we query PostgreSQL?) without a running database.

Do not mock deep inside the evaluation engine. The engine is pure logic – there is nothing to mock. If you find yourself wanting to mock something inside the evaluator, that is a signal that the evaluator has an unwanted dependency on I/O, and you should refactor it out.

Do not mock the HTTP layer. Use Fastify's `inject()` for integration tests instead of mocking the request/response objects. Mocking HTTP internals is brittle and does not catch the bugs you care about (serialization issues, header handling, status codes).

The Testing Pyramid for the Evaluation API

The base of the pyramid (most tests, fastest, cheapest) is unit tests for the evaluation engine. These cover the algorithmic complexity and the edge cases. They run in milliseconds and need no infrastructure.

The middle layer is integration tests using `inject()` with mocked Redis and Prisma. These cover the API surface: authentication, request validation, response shape, error codes, rate limiting behavior. They run in seconds and need no infrastructure.

The top of the pyramid (fewest tests, slowest, most expensive) is end-to-end tests against a real Redis and PostgreSQL. These verify that the caching, pub/sub, and database queries work correctly with real infrastructure. They run in a CI environment with Docker-composed services.

The ratio should be roughly 70% unit, 25% integration, 5% end-to-end. If you find yourself writing more end-to-end tests than unit tests, you are testing at the wrong level of abstraction.

> **Coming from Laravel:** Laravel's testing pyramid is similar: unit tests for service classes, feature tests with `RefreshDatabase` for HTTP endpoints, and the rare browser test with Dusk for UI flows. The proportions are the same – most tests should be fast, isolated, and focused on logic.

Testing SSE Connections

SSE endpoints are harder to test because they involve long-lived connections. The approach:

For unit testing the event dispatch logic, test it like any event emitter. Subscribe a mock listener, trigger a flag change event, assert the listener received the correct data.

For integration testing the SSE endpoint, use Fastify's `inject()` to establish a connection, then simulate a Redis pub/sub message, and assert that the response stream contains the expected SSE-formatted event. Fastify's inject supports streaming responses, so you can read the stream incrementally.

The tricky part is testing the cleanup path: when a client disconnects, does the server correctly unsubscribe from pub/sub, decrement the connection counter, and clean up the connection tracking map? This requires simulating a disconnect, which you can do by aborting the inject request.

Bringing It All Together

The evaluation API is a focused service with a simple job: authenticate a request, look up the flag configuration, evaluate it against the user context, and return the result. Every architectural

decision – the separation from Next.js, the choice of Fastify, the Redis caching strategy, the SSE architecture, the evaluation algorithm – serves that job.

The mental model to carry: this service is an appliance. It takes flag configs in one end and evaluated results out the other. It should be boring, predictable, and fast. The interesting product decisions happen in the dashboard. The interesting scaling decisions happen in the infrastructure. The evaluation API itself should be the simplest, most reliable component in the system.

When you encounter a design decision not covered in this guide, apply the same framework: what does the hot path look like? What are the failure modes? What is the simplest approach that is correct? Optimize for predictability first, performance second, and elegance third.