Contents

08 – Project Architecture & Monorepo Thinking Guide

Audience: Backend developer with 5+ years PHP/Laravel experience moving into React, Next.js, Node.js, TypeScript, PostgreSQL (Prisma), and Redis.

Project: Flagline – a feature-flag SaaS.

What this document is: A thinking guide. Mental models, rationale, decision frameworks, and reasoning that helps you set up and understand the monorepo architecture yourself. This is the document a senior engineer would walk you through at a whiteboard.

What this document is not: A code reference. You will not find complete config files, full YAML manifests, or Dockerfile implementations here. When a small snippet clarifies a concept, it appears inline. For implementation details, the codebase itself is the reference.

Table of Contents

10. Scaling Thinking

---

1. Why a Monorepo

The Problem Stated Simply

Flagline ships five distinct pieces of software: a dashboard (Next.js), an evaluation API (Fastify), a JavaScript SDK, a React SDK, and shared infrastructure like types and the database client. In the Laravel world, you would probably put each of these in its own repository. You would publish the shared code as a Composer package on Packagist (or a private Satis server), and every consumer would pin a version and run `composer update` when the shared package changed.

This workflow is familiar, and it works. But it introduces a category of problems that become painful at exactly the wrong time – during fast iteration.

The Version Drift Problem

Imagine you are building the flag evaluation logic. The API defines what a `TargetingRule` looks like. The SDK consumes that same shape. The dashboard renders it in a form. If these three consumers live in separate repositories, changing the shape of `TargetingRule` requires you to:

1. Update the shared types package.
2. Publish a new version.
3. Open a PR in the API repo, bump the dependency, verify everything works.
4. Open a PR in the SDK repo, bump the dependency, verify everything works.
5. Open a PR in the dashboard repo, bump the dependency, verify everything works.

That is four coordinated pull requests for what is conceptually one change. And between step 2 and step 5, there is a window where different repos are running different versions of the shared types. This is version drift, and it is the source of a specific kind of bug that is maddeningly hard to debug – everything works in isolation, but the system breaks when the pieces talk to each other.

What a Monorepo Solves

A monorepo puts all five projects in one Git repository. When you change `TargetingRule` in the shared types package, every consumer sees the change immediately. You open one pull request. The type-checker runs across the entire codebase and tells you, in one pass, every place that needs to be updated. You fix them all in the same PR, and the entire system stays consistent.

> **Coming from Laravel:** Think of it this way. In a Laravel app, you would never put your `app/Models/` in a separate repository from your `app/Http/Controllers/`. They change together too often. A monorepo applies that same reasoning across service boundaries. The dashboard, the API, and the SDKs are separate projects with separate deployment targets, but they share a contract (the types), and that contract changes frequently during active development.

There are other benefits that compound over time:

Single tooling setup. You configure TypeScript, ESLint, Prettier, and your test runner once. Every package inherits the same rules. There is no “the dashboard uses ESLint 8 but the API is still on 7”

drift. In Laravel terms, imagine if every microservice in your system shared a single `phpstan.neon` and a single `pint.json`, and updating the rules applied everywhere in one commit.

Atomic cross-cutting changes. When you update a Prisma schema, you can update the database migration, the API route that uses the new field, the dashboard page that displays it, and the SDK type that represents it – all in one PR. The reviewer sees the full picture. The CI pipeline validates the full picture.

Easier onboarding. A new developer clones one repo, runs one setup command, and has the entire system running locally. They do not need to discover and clone five repositories, figure out which versions of shared packages to use, or set up five separate development environments.

The Trade-Off: Tooling Complexity

Monorepos are not free. They require a build system that understands the dependency relationships between packages (Turborepo). They require a package manager that handles workspaces well (pnpm). The initial setup is more complex than a single-project repository.

In the Laravel world, you have a single `composer.json` and a single `artisan` entry point. Everything is simple because everything is one project. A monorepo trades that simplicity for the ability to manage multiple projects as a coordinated whole. The question is whether the coordination benefits outweigh the tooling overhead.

For Flagline, they do. You have an API and a dashboard that share a database client and type definitions. You have two SDKs that need to stay in sync with the API they call. These are not independent projects – they are tightly coupled by shared contracts. A monorepo reflects that reality in the repository structure.

When a Monorepo Is NOT the Right Choice

A monorepo is the wrong choice when the projects inside it are genuinely independent. If you have two SaaS products that share nothing – different databases, different teams, different customers, different release cadences – putting them in one repository adds tooling complexity without coordination benefits.

A monorepo is also the wrong choice when the team is not willing to invest in the tooling. If nobody understands Turborepo and nobody wants to learn it, the monorepo will rot. Developers will work around the build system instead of with it, and you end up with the worst of both worlds – the complexity of a monorepo with the isolation of separate repos.

For a team of one to five developers building a product with tightly coupled pieces (like Flagline), a monorepo is almost always the right call. For a large organization with 50 teams building unrelated services, the calculus is different and usually points toward a polyrepo strategy with a shared package registry.

---

2. Turborepo Mental Model

What Turborepo Actually Is

Turborepo is a task runner that understands dependencies between packages. That is the entire concept. Everything else is an optimization on top of that idea.

When you run `turbo run build`, Turborepo does not just run `build` in every package. It reads the dependency graph – which packages depend on which other packages – and figures out the correct order. If `sdk-react` depends on `sdk-js`, and `sdk-js` depends on `shared-types`, then Turborepo builds `shared-types` first, then `sdk-js`, then `sdk-react`. Packages that do not depend on each other build in parallel.

> **Coming from Laravel:** You have probably written deployment scripts or Makefiles that look like `npm run build && php artisan migrate && php artisan optimize`. The order matters because each step depends on the previous one. Turborepo does the same thing, but it reads the dependency graph from your `package.json` files and figures out the order automatically. You declare the relationships, and Turborepo computes the execution plan.

The ^ Operator

The single most important concept in Turborepo's configuration is the `^` prefix in dependency declarations. When a task says it depends on `^build`, the `^` means "my dependencies' build tasks, not my own." So when the dashboard's `build` task declares `"dependsOn": ["^build"]`, it means: "before you build the dashboard, first build every package the dashboard depends on."

This is what creates the topological ordering. Turborepo walks the dependency graph, finds all the leaf packages (packages that depend on nothing), builds those first, then works its way up to the packages that depend on them, and so on until everything is built.

The Caching Mental Model

After Turborepo runs a task, it stores the outputs. The next time you ask it to run the same task, it checks whether the inputs have changed. If nothing has changed in a package's source files, configuration, or dependencies, Turborepo replays the cached outputs instead of running the task again.

The cache key is a hash of: the source files listed in the task's `inputs` configuration, the relevant environment variables, the outputs of dependency packages, and the task configuration itself. If any of those change, the cache misses and the task runs fresh.

In practice, this means that if you change a file in the dashboard, running `turbo run build` only rebuilds the dashboard. The SDK packages, the API, and the shared types all get cache hits and complete instantly. A full monorepo build that takes 90 seconds becomes a 15-second incremental build for the one package you changed.

> **Coming from Laravel:** There is no direct equivalent, but the closest analogy is asset compilation caching in Vite. If you run `npm run build` and only a CSS file changed, Vite does not recompile your JavaScript. Turbo applies that same principle across

every package in the monorepo, and across every type of task – not just builds, but also linting, type-checking, and testing.

Remote Caching

Local caching speeds up your own workflow. Remote caching extends that to the whole team. When Turborepo builds a package, it can push the cache to a remote store (Vercel provides this). When another developer – or a CI runner – tries to build the same package with the same inputs, it downloads the cached outputs instead of rebuilding.

The effect is dramatic in CI. If your PR only changes the dashboard, the CI pipeline gets cache hits for every other package. The build step that would take minutes completes in seconds because the SDK and API packages were already built by someone else (or by a previous CI run) and the results are in the remote cache.

You enable remote caching with `turbo login` and `turbo link`. In CI, you set `TURBO_TOKEN` and `TURBO_TEAM` environment variables. That is the entire setup.

Parallelism and the Pipeline

Turborepo runs independent tasks in parallel by default. If `sdk-js` and `apps/dashboard` do not depend on each other, their build tasks run at the same time. You do not need to configure this – it happens automatically based on the dependency graph.

The `persistent` flag marks tasks that run indefinitely, like development servers. Turborepo starts them and does not wait for them to finish. The `cache: false` flag marks tasks that should never be cached, like dev servers or database operations, because their outputs are not meaningful to replay.

The mental model is: define what each task depends on, what its inputs and outputs are, and let Turborepo figure out the execution plan. You describe the "what," Turborepo handles the "how" and "when."

---

3. Workspace Structure Thinking

The `apps/` vs `packages/` Split

The organizing principle is simple: `apps/` contains things that get deployed, `packages/` contains things that get imported.

An app is a deployable artifact with its own entry point and runtime. The dashboard is a Next.js app that gets deployed to Vercel. The API is a Fastify server that gets deployed to Fly.io or Railway. Each has its own build command, its own start command, and its own deployment target.

A package is a library that is consumed by apps (or by other packages). It does not run on its own. The shared types package exports TypeScript interfaces. The database package exports a configured Prisma client. The SDKs export functions and React components. Config packages export ESLint rules and TypeScript configurations.

> **Coming from Laravel:** Think of `apps/` as your `routes/` directory – these are entry points. Think of `packages/` as your `app/` directory plus any first-party Composer packages – these are the libraries that the entry points use.

The Flagline Workspace Map

For Flagline, the split looks like this:

Apps (deployable): - `apps/dashboard` – Next.js App Router, deployed to Vercel. This is the SaaS dashboard where customers manage their flags. - `apps/api` – Fastify server, deployed to Fly.io. This is the evaluation endpoint that customer SDKs call to get flag values.

Packages (importable, published to npm): - `packages/sdk-js` – The core JavaScript SDK (`@flagline/js`). Customers install this in their Node.js or browser applications. - `packages/sdk-react` – The React SDK (`@flagline/react`). Wraps `sdk-js` with React hooks and a context provider.

Packages (importable, internal only): - `packages/shared-types` – TypeScript type definitions shared between all other packages. The contract that keeps everything in sync. - `packages/db` – The Prisma client and schema. Both the dashboard and the API need to query the database; this package gives them a shared, type-safe client.

Packages (tooling, internal only): - `packages/config-eslint` – Shared ESLint configuration. Every package extends from a base config defined here. - `packages/config-typescript` – Shared `tsconfig.json` base files for different environments (Next.js, Node.js, React libraries).

The Principle: One Clear Purpose

Every package should have one clear reason to exist. If you find yourself asking "what is this package for?" the package is either doing too much or should not be a package at all.

Shared types exist because multiple packages need the same type definitions. The database package exists because multiple apps need the Prisma client. The config packages exist because you want consistent tooling rules across the monorepo.

The inverse of this principle: do not create a package just because you can. If a utility function is only used by the dashboard, it belongs in the dashboard's `lib/` directory, not in a `packages/utils` grab-bag. When a piece of code is used by exactly one consumer, it does not need to be shared. Extract it into a package only when a second consumer needs it, and only if both consumers are better served by sharing the code than duplicating it.

> **Coming from Laravel:** This is the same reasoning you apply when deciding whether to extract a service class into a reusable package versus keeping it in `app/Services/`. The threshold for extraction is higher in a monorepo because the overhead of creating a new package – even a small one – includes a `package.json`, a `tsconfig.json`, and Turborepo integration. That overhead is small, but it is not zero.

When to Create a New Package

Ask yourself three questions:

1. **Is this code used by more than one app or published package?** If yes, it should probably be a package.
2. **Does this code change independently from its consumers?** If a piece of code changes at a different cadence than the app that uses it, extracting it into a package lets Turborepo cache it independently. Changes to the app do not invalidate the package's cache.
3. **Does this code have a clear, nameable purpose?** If you cannot give the package a two-word name that accurately describes what it does, it might be too broad or too vague to be a good package.

If the answer to at least two of these is yes, create a package. Otherwise, keep the code where it is.

---

4. Shared Types and the Dependency Graph

The Contract Layer

The `shared-types` package is the most important package in the monorepo, even though it contains zero runtime code. It is the contract between every other piece of the system.

When the API returns an evaluation result, the shape of that result is defined in `shared-types`. When the SDK parses that result, it imports the same type. When the dashboard displays flag configurations, it uses the same `FlagConfig` interface. If any of these consumers disagree about the shape of the data, the system breaks. Shared types prevent that disagreement by giving everyone a single source of truth.

> **Coming from Laravel:** This is equivalent to a `App\DataTransferObjects` namespace or an `App\Contracts` namespace that defines the shapes your application works with. The difference is that in Laravel, everything lives in one codebase anyway, so sharing types is automatic. In a monorepo with multiple packages, you need an explicit package to hold the shared contracts.

What Goes In Shared Types

The rule of thumb is: a type goes in `shared-types` if it crosses a package boundary. Specifically:

- **Flag shapes** – `FlagConfig`, `FlagVariation`, `TargetingRule`. The dashboard creates these, the API evaluates them, the SDKs consume the evaluation results.
- **Evaluation types** – `EvaluationContext`, `EvaluationResult`, `EvaluationReason`. The SDKs send the context, the API returns the result. Both sides need to agree on the shape.
- **API response envelopes** – `ApiResponse<T>`, `ApiError`, `ApiMeta`. Every API consumer (dashboard, SDKs) needs to know how responses are wrapped.
- **SSE event types** – `SSEEvent`, `SSEEventType`. The API emits these, the SDKs subscribe to them.
- **Billing types** – `PlanTier`, `PlanLimits`. The dashboard displays plan information, the API enforces rate limits based on it.

What Does NOT Go In Shared Types

- **Database model types.** The Prisma-generated types live in the `db` package. They describe the database schema, not the API contract. The dashboard and API import Prisma types from `@flagline/db`, not from `shared-types`. The distinction matters because database schemas and API contracts evolve independently – you might add a database column that is never exposed through the API.
- **UI-specific types.** The shape of a form state, a component's props, or a dashboard-specific view model belong in the dashboard. The SDK does not care about how you render a flag targeting editor.
- **Implementation details.** A type that describes an internal data structure in the API's evaluation engine is not a shared type. It is an implementation detail of the API.

The Database Package

The `db` package exists because both the dashboard and the API need to talk to the database. Rather than duplicating the Prisma configuration and generating the client in two places, you generate it once in the `db` package and import it from both apps.

The `db` package exports two things: the Prisma client instance (configured as a singleton to avoid exhausting connection pools during hot reload) and the generated Prisma types. Any app that lists `"@flagline/db": "workspace:*"` in its dependencies can `import { prisma } from "@flagline/db"` and get a fully typed database client.

> **Coming from Laravel:** The `db` package is like having a dedicated `database/` package that includes your models, migrations, seeders, and a configured database connection. In Laravel, this is all built into the framework. In Node.js, you wire it yourself, and the monorepo package structure gives you a natural place to put it.

How the Workspace Protocol Wires Things Together

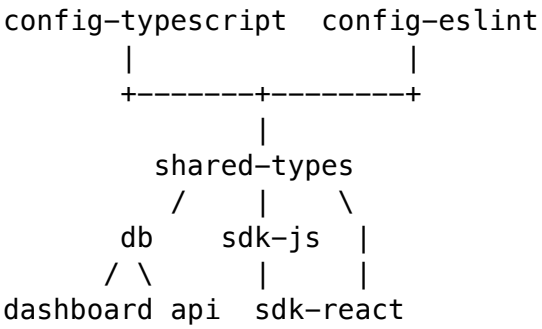
When the dashboard's `package.json` lists `"@flagline/types": "workspace:*"`, pnpm does not look for that package on npm. The `workspace:*` protocol tells pnpm to resolve the dependency from the monorepo itself. Under the hood, pnpm creates a symlink from `apps/dashboard/node_modules/@flagline/types` to `packages/shared-types/`. Imports resolve to the actual source directory, not to a published tarball.

The `*` means "any version." For internal packages that are never published to npm, the version in their `package.json` is irrelevant. For published packages like the SDKs, `workspace:*` is automatically replaced with the actual version when you publish. So `"@flagline/js": "workspace:*"` in `sdk-react` becomes `"@flagline/js": "^0.1.0"` in the published npm tarball.

This is the mechanism that makes the monorepo's shared code feel like local code. There is no publish step, no version bump, no waiting for a registry. You change a type, save the file, and every consumer sees it immediately.

The Dependency Graph

Understanding the dependency graph is essential for reasoning about build order, cache invalidation, and the blast radius of changes. For Flagline, it looks like this:



Reading this graph tells you several things:

- Changing `shared-types` invalidates everything. Every package depends on it, so every package needs to be rebuilt, re-linted, and re-tested.
- Changing `sdk-js` invalidates `sdk-react` (which depends on it) but does not affect the dashboard or the API.
- Changing the dashboard affects nothing else. It is a leaf node with no dependents.
- The config packages are at the root. Changing an ESLint rule re-lints everything.

This is why shared-types should be stable. It is the most expensive package to change, not in terms of code, but in terms of downstream effects. Every change to it ripples through the entire monorepo.

---

5. Development Workflow Thinking

The First-Time Experience

When a new developer joins the team, the setup experience should be: clone the repo, run one command, and have a working development environment. Every additional manual step is a potential point of confusion and wasted time.

For Flagline, the first-time flow is:

1. **Clone the repo.** Standard Git.
2. **Run the setup command.** A Makefile target or script that does everything: enables Corepack (so pnpm is available), installs dependencies, starts Docker Compose for Postgres and Redis, copies `.env.example` to `.env`, generates the Prisma client, runs migrations, seeds the database.
3. **Start development.** `pnpm dev` starts everything – the Next.js dashboard on port 3000, the Fastify API on port 3001, and any SDK watchers.

   Coming from Laravel: This maps directly to the Laravel onboarding experience: `git clone`, `composer install`, `cp .env.example .env`, `php artisan key:generate`, `php artisan migrate --seed`, `php artisan serve`. The

monorepo version has more steps internally, but the Makefile abstracts them into one command, just like Laravel Sail's `sail up` abstracts Docker setup.

Why Docker Compose for Infrastructure Only

The Flagline development setup uses Docker Compose for Postgres and Redis, but runs the Node.js applications directly on the developer's machine. This is a deliberate choice.

Containerizing the infrastructure (databases) gives you: consistent versions across the team, easy teardown and reset, and no need to install Postgres or Redis natively. Containerizing them costs almost nothing in terms of development speed because you interact with them over TCP, and the network overhead of Docker's bridge is negligible.

Running the Node.js applications natively gives you: fast hot-reload (no file system sync delays between host and container), native file watching, and simpler debugging. Next.js and Fastify both have excellent hot-reload capabilities that work best when they have direct access to the file system.

> **Coming from Laravel:** This is the opposite of Laravel Sail, which containerizes everything including PHP. The difference is that PHP's development server does not have the same hot-reload story as Node.js. In the JS ecosystem, the dev servers are designed to run natively and watch the file system directly. Putting them in Docker adds latency to every file save.

How `pnpm dev` Works

When you run `pnpm dev` from the repo root, pnpm delegates to Turborepo, which reads the dependency graph and starts tasks in the correct order:

1. First, Turborepo builds the leaf packages – `shared-types` and `db`. These need to be ready before anything that depends on them can start.
2. Then, Turborepo starts the dev servers for the apps in parallel. The dashboard starts Next.js, the API starts Fastify with file watching.
3. If the SDK packages have dev scripts (watch mode for `tsup`), those start too.

All of this output is multiplexed into a single terminal with colored labels indicating which package each log line came from. You see dashboard logs, API logs, and build output interleaved in one view.

Environment Variables: How to Think About Them

Each app in the monorepo has its own set of environment variables, and this is a source of confusion for developers coming from a single-app world.

The root `.env.example` serves as documentation – it lists every variable the system needs. But each app only reads the variables relevant to it. The dashboard needs `NEXTAUTH_SECRET` and `STRIPE_SECRET_KEY`. The API needs `REDIS_URL` but does not care about Stripe. The `db` package needs `DATABASE_URL`.

In development, you can put all variables in a single root `.env` file, and most tools will find them. In production, each app has its own environment configured on its deployment platform (Vercel for

the dashboard, Fly.io for the API).

The one gotcha: Next.js's `NEXT_PUBLIC_` prefix. Any variable that starts with `NEXT_PUBLIC_` is embedded into the client-side JavaScript bundle at build time. This means it is visible to anyone who views your page source. Never put secrets in a `NEXT_PUBLIC_` variable. Use it only for values that are safe to expose, like the API URL or a Stripe publishable key.

> **Coming from Laravel:** This is identical to the `MIX_*` prefix in Laravel Mix or the `VITE_*` prefix in Vite. Same concept, different naming convention.

Hot Reload Across Packages

This is where the monorepo pays off in daily development. When you change a type in `packages/shared-types/src/flags.ts`:

- The dashboard's Next.js dev server detects the change (because the import resolves to the source file via the symlink) and performs a fast refresh. You see the change in the browser in under a second.
- The API's file watcher (`tsx watch`) detects the change and restarts the Fastify process.
- If the SDK watch mode is running, `tsup --watch` rebuilds the SDK output.

You change one file, and the entire system updates. No manual rebuild, no version bump, no publish cycle. This is the monorepo's killer feature for day-to-day development.

> **Coming from Laravel:** In a Laravel app, changing a model or a service class takes effect immediately on the next request because PHP re-reads the files. The monorepo hot-reload experience is similar in spirit – change a shared type, and every consumer picks it up – but the mechanism is different. Instead of re-reading files on each request, the dev servers watch for file changes and rebuild or restart as needed.

The Complexity Tax

It is worth acknowledging that this is more complex than `php artisan serve`. In Laravel, you start one process and everything works. In the monorepo, you start multiple processes coordinated by Turborepo, backed by Docker Compose for infrastructure.

The payoff is that you are running an entire distributed system locally – dashboard, API, databases – and changes propagate instantly across all of it. You catch integration issues during development, not after deployment. But the initial setup is heavier, and when things go wrong (a port conflict, a Docker volume corruption, a stale Prisma client), the debugging surface area is larger.

The Makefile helps. Common operations – setup, dev, reset, seed – should be one command. The developer should not need to remember the order of steps.

---

6. CI/CD Thinking

The Key Insight: Build Only What Changed

In a monorepo with eight packages, running every check on every package for every pull request is wasteful. If a PR only touches the dashboard, there is no reason to rebuild the SDKs or re-run

the API's test suite.

Turborepo's `--filter` flag is the mechanism for this. When you run `turbo run test --filter=...[origin/main]`, Turborepo compares the current branch to `origin/main`, identifies which packages have changed files, and runs tests only in those packages plus their dependents. The `...` prefix means "include packages that depend on the changed packages." If you change `shared-types`, it tests `shared-types` and everything that depends on it. If you change only the dashboard, it tests only the dashboard.

This is the same principle as running `php artisan test --filter=SomeTest` in Laravel, but applied at the package level across the entire monorepo.

The PR Pipeline: Order Matters

The PR pipeline runs four stages in this order: lint, type-check, test, build. The order is deliberate: fast failures first.

Lint runs in seconds. If there is a formatting issue or a lint rule violation, the developer finds out immediately, before waiting for anything slower.

Type-check runs in 10-30 seconds. If a type import is broken or a function signature is wrong, you find out next.

Test runs in 30-120 seconds (depending on the scope). This is where business logic correctness is verified.

Build runs last. It takes the longest and catches the rarest class of errors – things that lint, type-check, and tests did not catch but that prevent the code from compiling to a production artifact.

If you ordered this differently – say, build first, then lint – a developer with a simple formatting issue would wait two minutes for the build before finding out they forgot a semicolon. The fast-first ordering minimizes wasted CI time and developer frustration.

> **Coming from Laravel:** This is equivalent to running `pint --test` (fast), then `phpstan analyse` (medium), then `phpunit` (slow) in your CI pipeline. The reasoning is identical: fail fast on cheap checks before investing time in expensive ones.

Turborepo Remote Caching in CI

CI runners start from a clean slate on every run. Without remote caching, every CI run rebuilds every package from scratch. With remote caching, the CI runner downloads cached build outputs from previous runs. If the SDK packages have not changed since the last CI run, their build results are pulled from the cache instead of being rebuilt.

To enable this, you set two environment variables in your CI environment: `TURBO_TOKEN` (an authentication token from Vercel) and `TURBO_TEAM` (your Vercel team slug). Turborepo handles the rest – pushing cache after builds, pulling cache before builds.

The effect is significant. A full monorepo build that takes three minutes might complete in 40 seconds when only one package changed and everything else has cache hits. Over hundreds of CI runs per week, this saves substantial time and money on CI minutes.

Separate Deployment Workflows

The PR pipeline validates code quality. Deployment is a separate concern with separate triggers. Flagline has three deployment targets, and each has its own workflow:

Dashboard deploys to Vercel when changes to `apps/dashboard/`, `packages/shared-types/`, or `packages/db/` are merged to `main`. The `paths:` filter in the GitHub Actions workflow ensures that an SDK-only change does not trigger a dashboard deploy. This is like having separate Forge deployment triggers for different parts of your application.

API deploys to Fly.io when changes to `apps/api/`, `packages/shared-types/`, or `packages/db/` are merged to `main`. Same principle – only relevant changes trigger the deploy.

SDKs publish to npm when a version tag (`v*`) is pushed. This is a manual, intentional process using Changesets (more on this in Section 9). You do not want SDK publishes triggered by every merge to main – your customers depend on those packages, and you want explicit control over when new versions ship.

The `paths:` filter deserves emphasis because it is a common source of mistakes. If the dashboard deployment workflow does not include `packages/shared-types/**` in its paths, a breaking type change could be merged without triggering a dashboard deploy, and the production dashboard would still be running the old code. The paths must include every package that the app depends on, directly or transitively.

> **Coming from Laravel:** In the Laravel world, Forge or Envoyer deploys everything on every push to `main`. There is no concept of "only deploy if these files changed" because there is only one deployable artifact. In a monorepo, you have multiple artifacts, and deploying all of them on every push wastes time and creates unnecessary risk. The path-filtered workflows are the monorepo equivalent of targeted deployment.

Preview Deployments

Vercel creates a preview deployment for every pull request automatically. Each PR gets a unique URL where the reviewer can see the dashboard changes running in an isolated environment.

The powerful extension of this is database branching. If you use Neon for Postgres, each preview deployment can get its own isolated database branch – a copy-on-write fork of the production database that is instant to create and costs nothing for unchanged data. This means the reviewer can test the PR against real-ish data without touching the production database.

This is transformative compared to the Laravel staging workflow where everyone shares one staging database and deployments step on each other. Each PR gets its own preview URL and its own database branch. When the PR is closed, both are cleaned up automatically.

The API is harder to preview because Fly.io does not have the same per-PR deployment story as Vercel. For the evaluation API, you can either skip preview deployments (the dashboard preview points to the production API) or set up a staging API instance that preview deployments share. For most changes, pointing preview dashboards at the production API is fine – the dashboard is just displaying data, not making irreversible changes.

---

7. Deployment Architecture Thinking

The Big Picture

Flagline's production architecture has four main pieces: the dashboard, the evaluation API, the database, and the cache. Each is deployed to a different platform, and understanding why helps you make the right trade-offs.

The guiding principle is: use the right tool for each job's constraints. The dashboard has different requirements than the API. The API has different requirements than the database. Trying to run everything on a single platform (the way a Laravel app runs entirely on a Forge-managed server) works for simpler architectures but creates friction when the pieces have genuinely different needs.

Dashboard on Vercel: Why It Fits

The dashboard is a Next.js application, and Vercel is the company that builds Next.js. This is not just a marketing alignment – Vercel's infrastructure is purpose-built for Next.js's rendering model. Server Components, Server Actions, Edge Functions, ISR (Incremental Static Regeneration), and streaming all work out of the box with zero configuration.

Vercel is a serverless platform. Each request to the dashboard spins up a function (or reuses a warm one), handles the request, and returns the response. There is no long-lived server process, no nginx to configure, no SSL certificates to manage. Vercel handles scaling, CDN caching, and global distribution automatically.

The trade-off is control. You cannot SSH into a Vercel machine and debug a production issue. You cannot install arbitrary system packages. You cannot run long-lived background processes. For the dashboard, none of these are needed. It serves pages, processes form submissions via Server Actions, and handles webhooks. Serverless is a perfect fit.

> **Coming from Laravel:** Vercel is what you would get if Forge provisioned a server, configured nginx, set up SSL, configured a CDN, and managed auto-scaling – all automatically, with zero configuration, and you could never SSH into it. The loss of SSH access sounds scary until you realize you rarely SSH into production to do anything other than run Artisan commands, and in the serverless model, those operations are handled differently (migrations run in CI, not on the server).

Evaluation API on Fly.io or Railway: Why Not Vercel

The evaluation API cannot run on Vercel for a specific technical reason: SSE (Server-Sent Events). The SDK keeps a persistent connection open to the API to receive real-time flag updates. Vercel's serverless functions have a maximum execution time (10 seconds on Hobby, 60 seconds on Pro, 300 seconds on Enterprise). A persistent SSE connection that stays open for minutes or hours is fundamentally incompatible with the serverless model.

This is the kind of constraint that is easy to overlook during architecture planning and painful to discover in production. The lesson is: understand your platform's execution model before committing to it. Serverless is great for request-response patterns. It is wrong for long-lived connections.

Fly.io deploys Docker containers to bare-metal servers in data centers around the world. You choose which regions to deploy to, and Fly handles routing users to the nearest region. The API

runs as a normal, long-lived process that can hold thousands of SSE connections open simultaneously. You get full control over the runtime environment, and you can deploy to 30+ regions for low-latency evaluation.

Railway is simpler. You push code, Railway detects the runtime, builds a container, and deploys it. Less configuration, fewer knobs, faster time to first deployment. The trade-off is that Railway runs in a single region (or a limited number of regions), so users far from that region experience higher latency.

How to choose: If you are building an MVP and latency is not yet a concern, start with Railway. It is simpler, cheaper to get started, and you can migrate to Fly.io later. If you know from the start that evaluation latency matters (because your customers' page loads depend on it), go with Fly.io and deploy to the regions where your customers are.

For Flagline, the recommendation is Fly.io because the evaluation API's core value proposition is speed. A flag evaluation that takes 200ms because the API is in US East and the customer is in Singapore undermines the product. Multi-region deployment is not a premature optimization for a feature flag service – it is a core product requirement.

Database: How to Think About Managed Postgres

You need managed Postgres, not a self-hosted instance. Self-hosting means you are responsible for backups, failover, monitoring, patching, and scaling. For a SaaS startup, that operational burden is not worth it. Managed Postgres costs slightly more per month but saves dozens of hours of operations work.

The three main options are **Neon**, **Supabase**, and **Railway Postgres**. Here is how to think about each:

Neon is serverless Postgres with a standout feature: database branching. Each branch is a copy-on-write fork of another branch, instant to create and near-zero cost for unchanged data. This integrates with Vercel preview deployments, giving each PR its own isolated database. Neon also includes built-in connection pooling and supports read replicas.

Supabase is a broader platform that includes Postgres, authentication, file storage, real-time subscriptions, and edge functions. If you need those additional features, Supabase is compelling. For Flagline, which uses Auth.js for authentication and has its own API, most of Supabase's extras are redundant. You would be using it as managed Postgres, which it does well, but you are paying (in complexity, if not money) for features you do not use.

Railway Postgres is straightforward managed Postgres. No branching, no read replicas, fewer features. But it is simple, and it lives on the same platform as your API if you choose Railway for that. One dashboard, one billing account.

The recommendation for Flagline is Neon. Database branching for preview environments is a workflow improvement that compounds over time. Connection pooling and read replicas are important for the evaluation API's scaling story (discussed in Section 10). The serverless pricing model means you pay based on usage, not for an always-on instance.

Connection pooling is worth understanding. Serverless functions (like Vercel's) create new database connections for each invocation. Without connection pooling, a traffic spike can exhaust the database's connection limit (typically 100-200 for managed Postgres). A connection pooler

(built into Neon via PgBouncer) sits between the application and the database, reusing a small pool of actual connections across many application requests. You use two connection strings: a pooled one for the application (`DATABASE_URL`) and a direct one for Prisma migrations that need DDL access (`DIRECT_URL`).

> **Coming from Laravel:** Connection pooling is something Forge handles transparently. In the serverless world, you need to be aware of it because the connection pattern is fundamentally different. A Forge server opens a connection pool when PHP-FPM starts and reuses it across requests. A serverless function opens a new connection on every cold start. Without pooling, this quickly becomes a problem.

Redis: Upstash vs Self-Hosted

Redis serves two purposes in Flagline: caching flag configurations for fast evaluation, and pub/sub for real-time flag update notifications.

Upstash is serverless Redis with an HTTP/REST API. This matters because Vercel Edge Functions and serverless functions cannot open raw TCP connections in all environments. Upstash's REST API works everywhere – edge, serverless, and traditional servers.

For the Fastify API running on Fly.io, you use a standard Redis client (`ioredis`) over TCP for best performance. For the Next.js dashboard on Vercel, you use Upstash's HTTP client (`@upstash/redis`) for edge compatibility. Same Redis instance, two different access patterns.

The trade-off with Upstash's HTTP API is slightly higher latency per operation (1-2ms per call instead of sub-millisecond over TCP). For the dashboard, where Redis calls happen during page renders, this is negligible. For the evaluation API, where microseconds matter, you use the TCP client.

> **Coming from Laravel:** Upstash replaces your self-managed Redis instance (or the Redis that comes with Forge). The HTTP API is a new concept – imagine `Redis::get('key')` making an HTTPS request instead of a TCP connection. It sounds wrong, but it solves a real problem in the serverless world where TCP connections are not always available.

Domain Architecture

Flagline uses separate subdomains for each service:

- `flagline.dev` – Marketing site (landing page, pricing, docs)
- `app.flagline.dev` – Dashboard (the SaaS product)
- `api.flagline.dev` – Evaluation API (what SDKs call)
- `docs.flagline.dev` – API documentation

Why separate subdomains instead of paths (`flagline.dev/app`, `flagline.dev/api`)? Three reasons:

1. **Independent deployment.** Each subdomain points to a different platform. `app.flagline.dev` is a CNAME to Vercel. `api.flagline.dev` is a CNAME to Fly.io. Path-based routing would require a reverse proxy in front of everything, adding latency and a single point of failure.

2. **Independent scaling.** The API might get 10,000 requests per second while the dashboard gets 10. Separate subdomains mean separate infrastructure that scales independently.
3. **Security isolation.** Cookies set on `app.flagline.dev` are not sent to `api.flagline.dev`. This is browser security working in your favor – authentication cookies for the dashboard are not leaked to the evaluation API.

   Coming from Laravel: In a typical Laravel setup, everything lives on one domain – `app.com/dashboard`, `app.com/api`. The web server (nginx) routes requests to the right handler. This works because everything runs on one server. When you distribute across multiple platforms, separate subdomains are the natural approach.

---

8. Monitoring and Observability Thinking

The Three Pillars

Observability is about answering the question: "What is happening in production right now, and what happened when something went wrong?" There are three categories of signal that help you answer this.

Errors (Sentry): Something broke. An unhandled exception, a failed database query, a 500 response. You need to know about it immediately, with enough context to debug it.

Logs (structured JSON): What happened during a request. Which flag was evaluated, which rule matched, how long the evaluation took. Logs are the detailed narrative of your system's behavior.

Metrics (latency, throughput, error rates): Aggregate patterns over time. Is the average evaluation latency increasing? Is the error rate above 1%? Is the database connection pool exhausted? Metrics tell you about the system's health, not individual requests.

Start with Errors, Add the Rest Later

If you do one thing for observability, integrate Sentry. It takes 15 minutes to set up, and it catches errors that would otherwise go unnoticed. Without error tracking, you find out about bugs from your customers. With Sentry, you find out from an alert, often before any customer notices.

The minimum viable Sentry setup: install `@sentry/nextjs` for the dashboard, install `@sentry/node` for the API, configure the DSN, and you are done. Sentry captures unhandled exceptions with full stack traces, breadcrumbs (what happened before the error), and request context.

Structured logging is the second priority. When something goes wrong and Sentry shows you the error, you often need to understand the broader context – what was the request? What were the flag evaluation parameters? Which targeting rule was being evaluated? Structured logs answer these questions.

The key word is "structured." A log line like `"Evaluated flag dark-mode for user_123"` is human-readable but machine-hostile. You cannot search for all evaluations of `dark-mode`, or all evaluations for `user_123`, without regex. A structured log entry like `{"flagKey": "dark-mode", "userId": "user_123", "result": true, "latencyMs": 2}` is searchable, filterable, and aggregatable.

Metrics dashboards come last. You need traffic before metrics are meaningful, and you need to know what questions to ask before you know what to measure. Start with error tracking and logging. Add metrics when you have production traffic and need to understand performance trends.

> **Coming from Laravel:** Sentry replaces the combination of Laravel's exception handler and a service like Bugsnag or Flare. Structured logging replaces `Log::info('message', ['context' => $data])` – same idea, but enforced as the default format rather than an option. Metrics replace the dashboards you might see in Horizon or Telescope.

Request ID Tracing

When a user clicks "Create Flag" in the dashboard, the request flows through multiple services: the Next.js server action creates a database record, then calls the evaluation API to invalidate the cache, which triggers a Redis pub/sub message to connected SDKs. If something goes wrong anywhere in that chain, you need to follow the thread.

A request ID is a unique identifier (typically a UUID) generated at the start of the chain and passed to every subsequent service. The dashboard generates it and sends it as an `X-Request-ID` header when calling the API. The API includes it in every log entry for that request. The API returns it in the response header.

Now, when you see a Sentry error in the API, you can search the dashboard logs for the same request ID and understand what triggered the request. When a user reports a problem, they can give you the request ID from the response headers, and you can trace the entire chain.

This sounds like overkill for a small system, and it is during early development. But adding request IDs later (once you have traffic and real bugs) is much harder than building them in from the start. The implementation is a few lines of middleware in each service and costs nothing at runtime.

Health Checks

Health checks serve two audiences: the infrastructure (Fly.io, Vercel, load balancers) and you.

A **liveness check** (`/health`) answers: "Is the process running?" It returns 200 immediately, without checking any dependencies. If this fails, the process is dead and should be restarted. Keep it simple – literally just return `{ status: "ok" }`.

A **readiness check** (`/health/ready`) answers: "Can this process serve requests?" It pings the database and Redis and returns 200 only if both are healthy. If this fails, the process is alive but cannot do its job, usually because a dependency is down. Infrastructure should stop routing traffic to it until it recovers.

The distinction matters for deployment. During a deploy, the new container starts and becomes "live" (the process is running) before it becomes "ready" (the database connection is established). The load balancer should not route traffic to it until the readiness check passes. Without this distinction, users get errors during deployments because traffic is routed to containers that are not yet ready.

> **Coming from Laravel:** Forge does not have this concept because it manages a single long-lived PHP-FPM process. Health checks are a container-world concept where

processes come and go, and the orchestrator needs to know when each instance can accept traffic.

---

9. Developer Experience Tooling

The Principle: Automate Every Debate

Code formatting, import ordering, commit message format – these are decisions that do not affect the product but generate disproportionate discussion. The solution is to automate them. Configure the tool once, run it automatically, and never discuss it again.

Prettier handles code formatting. You save a file, Prettier reformats it. Tabs vs spaces, semicolons, trailing commas – all decided by the `.prettierrc` configuration, not by a human in a code review.

ESLint handles code quality. Unused variables, missing type annotations, incorrect React hook usage – all caught automatically. Run on save in the editor, run on commit via pre-commit hooks, run in CI as a safety net.

Husky + lint-staged runs Prettier and ESLint on every commit, automatically. You do not need to remember to format your code. The pre-commit hook formats and lints only the files you changed (not the entire codebase), so it runs in seconds.

> **Coming from Laravel:** This is equivalent to combining Pint (formatting), PHPStan (static analysis), and a pre-commit hook that runs both. The difference is that in the JS ecosystem, this toolchain is standard practice and well-supported. Every serious project uses it.

Conventional Commits and Changesets

Conventional commits (`feat(dashboard): add flag targeting editor`) are not just aesthetic. They serve a practical purpose: automated changelog generation for the SDKs.

When you use Changesets for SDK versioning, the commit history helps you write accurate changelog entries. A commit that says `fix(sdk-js): handle null evaluation context` clearly indicates a patch-level bugfix. A commit that says `feat(sdk-react): add useFlags() hook` is a minor version bump.

The workflow for SDK releases:

1. You make changes to the SDK code and commit with conventional commit messages.
2. You run `pnpm changeset`, which prompts you to select which packages changed, what kind of version bump (patch, minor, major), and a changelog summary.
3. Before release, you run `pnpm changeset version`, which updates the `package.json` versions and generates `CHANGELOG.md` entries.
4. You commit the version changes, push a tag, and the CI publish workflow publishes to npm.

This is more ceremony than a Laravel `composer` publish, but your SDK customers depend on semantic versioning being accurate. A breaking change published as a patch will break their applications. The Changesets workflow forces you to think about the version bump, not just the code change.

The "linked" configuration in Changesets ensures @flagline/js and @flagline/react always have the same version number. If you bump one, you bump both. This prevents confusion for customers who install both packages – they always see matching version numbers.

Node.js Version Pinning

An .nvmrc file at the repo root specifies the Node.js version. Every developer runs nvm use (or uses automatic version switching) when entering the project directory. CI reads the same file to install the correct version.

This prevents "works on my machine" bugs caused by Node.js version differences. Node 18 and Node 20 have different behaviors around, for example, the fetch API and crypto module. A test that passes on Node 20 might fail on Node 18. The .nvmrc eliminates that variable entirely.

The "packageManager" field in the root package.json serves the same purpose for pnpm. Corepack reads this field and activates the correct pnpm version automatically. Between .nvmrc and "packageManager", every developer and CI runner uses the exact same versions of the two most fundamental tools.

> **Coming from Laravel:** This is equivalent to the "require": { "php": "^8.2" } constraint in composer.json, but enforced at the environment level. Composer checks the constraint and warns you. .nvmrc and Corepack actually switch to the correct version.

TypeScript Strict Mode Everywhere

The shared TypeScript configuration in packages/config-typescript/base.json enables "strict": true. This is non-negotiable.

Strict mode catches an entire category of bugs at compile time: null pointer dereferences, implicit any types, unchecked array indexing. The trade-off is that you write slightly more explicit code – you need to handle null cases, you need to annotate function parameters, you need to narrow types before accessing properties. This is a good trade-off. The bugs strict mode catches are the kind that cause runtime crashes in production.

The noUncheckedIndexedAccess option is particularly valuable and often overlooked. It makes array access and record lookups return T | undefined instead of T, forcing you to handle the case where the index does not exist. In a feature flag system where you look up flags by key, this prevents an entire class of "flag not found" bugs.

> **Coming from Laravel:** This is the equivalent of running PHPStan at level 8 (max). It is more work to satisfy the analyzer, but it catches real bugs before they reach production. Once you get used to it, going back to untyped code feels like driving without a seatbelt.

---

10. Scaling Thinking

What to Think About Now vs Later

Premature optimization is a real risk, especially for a SaaS that might have ten users for the first six months. But premature optimization and architectural foresight are different things. You do not

need to implement scaling solutions now, but you should understand the scaling characteristics of each component so you do not paint yourself into a corner.

The mental model is: make decisions now that do not close doors later. A stateless API is not harder to build than a stateful one, but it is dramatically easier to scale. Using Redis pub/sub instead of in-memory event emitters adds no complexity now but enables horizontal scaling later.

The Evaluation API Is the Bottleneck

The dashboard serves your team and your customers' teams. Call it hundreds of page views per day. Vercel handles this without you thinking about it.

The SDKs run in your customers' applications. They scale with your customers' traffic, but that is your customers' problem – the SDKs are client-side code running in their infrastructure.

The evaluation API sits in the middle. Every SDK instance calls it. If you have 100 customers, each with 10,000 daily active users, and each user triggers 5 flag evaluations per session, that is 5 million evaluation requests per day. The evaluation API is the component that needs to scale.

Scaling Levers for the Evaluation API

Horizontal scaling (more instances). The evaluation API is stateless – it reads flag configurations from Redis, evaluates them against the context, and returns the result. It does not store session data or in-memory state. This means you can run 10 copies behind a load balancer, and each copy is interchangeable. Fly.io supports this natively – you set the number of machines per region and Fly handles routing.

The prerequisite for horizontal scaling is that the API must be truly stateless. This is why SSE connections use Redis pub/sub for flag update notifications instead of in-memory events. If instance A receives a flag update from the dashboard, it publishes to Redis, and instance B (which might have the SSE connection for that flag) receives it via subscription. Without Redis pub/sub, instance B would never know about the update.

Redis proximity (put the cache closer to the API). The evaluation API's hot path is: receive request, look up flag configuration in Redis, evaluate rules, return result. If the Redis call takes 5ms, that is your floor. If you deploy the API to 5 regions but Redis is in one region, the API instances far from Redis pay a network latency tax on every evaluation.

The solutions are either regional Redis instances (Upstash supports this) or local caching in the API process with a short TTL. The local-cache approach means each API instance keeps recently evaluated flags in memory for, say, 5 seconds. Most evaluations hit the local cache and never call Redis. The downside is that flag updates take up to 5 seconds to propagate, which is usually acceptable for feature flags (unlike, say, payment processing).

Read replicas for Postgres. The evaluation API only reads flag configurations from the database (the dashboard is the only writer). This means you can point the API at a read replica instead of the primary database. Neon supports read replicas in multiple regions, so the API in Singapore can read from a Postgres replica in Singapore instead of querying the primary in US East.

In practice, the evaluation API should rarely query Postgres directly. Flag configurations are loaded into Redis, and the API reads from Redis for each evaluation. Postgres is the source of truth, but

Redis is the runtime cache. The API hits Postgres only when Redis does not have the data (cache miss) or when it needs to warm the cache.

When to Think About These Problems

Not now. Flagline's first priority is building the product and getting customers. The architecture described above (stateless API, Redis pub/sub, connection pooling) supports horizontal scaling without additional work. When you need to scale:

1. **First lever: more API instances.** Add machines in Fly.io. This is a configuration change, not a code change.
2. **Second lever: Redis proximity.** Add a regional Redis instance or enable local caching. This is a small code change.
3. **Third lever: read replicas.** Add Neon read replicas in the regions where your API runs. This is a configuration change.
4. **Fourth lever: architecture changes.** If none of the above suffice, you are at a scale where you can afford to hire an infrastructure engineer. Congratulations.

The important thing is that nothing in the current architecture prevents you from pulling these levers. A stateful API, an in-memory event system, or a database without connection pooling would close these doors. The architectural choices described in this guide keep them open.

> **Coming from Laravel:** In the Laravel world, scaling typically means "get a bigger server" (vertical scaling on Forge) or "add more servers behind a load balancer" (horizontal scaling). The principles are the same here. The difference is that the serverless and container-based platforms (Vercel, Fly.io) make horizontal scaling a configuration change rather than a server-provisioning exercise.

The Dashboard Does Not Need to Scale

This is worth stating explicitly because it simplifies your mental model. The dashboard is a low-traffic application used by your customers' teams. Even at 1,000 customers with 5 team members each, you are talking about 5,000 people who use the dashboard occasionally. Vercel's free tier handles this without breaking a sweat.

Do not invest time optimizing the dashboard for performance. Invest in making it correct, usable, and pleasant. The evaluation API is where performance matters because it sits in your customers' hot paths. The dashboard is back- office software.

The SDKs Scale with Your Customers

The JavaScript and React SDKs run in your customers' applications. Your customer with 10 million monthly active users runs your SDK 10 million times – on their infrastructure, in their CDN, in their users' browsers. Your only involvement is the evaluation API calls the SDK makes, which are covered by the API scaling discussion above.

The SDK itself needs to be small (minimal bundle size), fast (minimal initialization time), and reliable (graceful fallback when the API is unreachable). These are quality concerns, not scaling concerns. You achieve them through good engineering practices, not infrastructure.

Appendix: The Mental Model – Laravel to Flagline

This table maps the major concepts for quick reference when you find yourself thinking “how would I do this in the Flagline world?”

Laravel Concept	Flagline Equivalent	Key Difference
Single `composer.json`	Root `package.json` + `pnpm-workspace.yaml`	Workspace-aware; manages multiple projects
`composer install`	`pnpm install`	Single lock file for the entire monorepo
`composer require`	`pnpm add <pkg> --filter @flagline/api`	Target a specific package
Packagist	npm registry	SDKs published here
`php artisan`	`turbo run <task>`	Orchestrates tasks across packages
`app/Models/`	`packages/db/prisma/schema.prisma`	Prisma schema defines all models
`database/migrations/`	`packages/db/prisma/migrations/`	Prisma manages migration SQL
`php artisan migrate`	`pnpm db:migrate`	Runs pending Prisma migrations
`database/seeders/`	`packages/db/prisma/seed.ts`	TypeScript seed script
`routes/web.php`	File-system routing in `apps/dashboard/app/`	Each file is a route
`routes/api.php`	`apps/api/src/routes/`	Fastify route modules
Blade templates	JSX/TSX components	Component-based, not template-based
`.env`	`.env` / `.env.local` per app	Same concept; `NEXT_PUBLIC_*` for client-side
Laravel Forge	Vercel (dashboard) + Fly.io (API)	Different platforms for different workloads
Laravel Envoyer	GitHub Actions workflows	CI/CD with path-filtered triggers
`php artisan serve`	`pnpm dev`	Starts all dev servers via Turborepo
Laravel Sail	`docker-compose.yml` (infra only)	Only databases containerized, not Node.js
Laravel Cashier	Stripe SDK + webhook handlers	Direct integration, no wrapper
Laravel Socialite	Auth.js (NextAuth)	OAuth providers
PHPUnit / Pest	Vitest	JavaScript test runner
PHP-CS-Fixer / Pint	Prettier	Code formatting
PHPStan / Larastan	TypeScript compiler (`tsc --noEmit`)	Static type checking
`composer.lock`	`pnpm-lock.yaml`	Deterministic dependency resolution

Laravel Concept	Flagline Equivalent	Key Difference
Monolog	Pino (API) / structured JSON	Structured logging
Laravel Horizon	Redis + Fly.io metrics	Queue and cache monitoring
Laravel Telescope	Prisma Studio + Sentry + Axiom	Dev debugging + production monitoring