ChatGPT

Next.js Financial Dashboard – Infrastructure and DevOps Guide

This guide outlines a production-grade Infrastructure and DevOps plan for a Next.js + React + Tailwind CSS + TypeScript financial dashboard application. It covers architecture, environments, CI/CD, runtime options, background jobs, database operations, security, monitoring, backup, scaling, and more. Each section addresses the specific requirements and includes best practices, code samples, and diagrams for clarity.

1. Target Architecture (Overview)

The application is a single-page financial dashboard built with Next.js (App Router) for a rich interactive UI. The core architecture follows a modular monolithic approach, with a clear separation of concerns between the front-end interface, the business logic (intelligence layer), an AI integration layer, and a design system for consistent UI components. In **Phase 1**, all modules run within a single Next.js application deployed on Vercel, connected to a managed PostgreSQL database and a Redis cache. **Phase 2** will evolve this architecture for multi-tenant SaaS (adding stricter data isolation and possibly microservices or separate workers as needed).

```
flowchart LR
    subgraph Client
        Browser["User's Web Browser<br/>(React SPA)"]
    end
    subgraph Server["Next.js App on Vercel"]
        direction TB
        UI[Front-End UI<br/>(Next.js React + Tailwind + Design System)]
        API[Next.js API Routes<br/>(Intelligence & AI logic)]
        Jobs[Background Jobs<br/>(Scheduled via Cron APIs)]
    end
    subgraph Data
        DB[(PostgreSQL Database<br/>(Managed, via Prisma))]
        Cache[(Redis Cache<br/>(Managed, for cache/queue))]
    end
    Browser -- HTTP+WSS --> UI
    UI -- API calls --> API
    API -- Prisma ORM --> DB
    API -- Cache (sessions/flags) --> Cache
    API -- External APIs --> ThirdParty[Third-Party Services<br/>(OpenFinance,
Belvo, Stripe, Gmail)]
    API -- Logging --> Monitoring[(Sentry / APM<br/>+ Logging)]
    Jobs -- read/write --> DB
    Jobs -- schedule via Cron --> API
```

```
    Jobs -- queue tasks --> Cache
    classDef cloud fill:#e8f0fe,stroke:#555,stroke-width:1px;
    class Server,Data,ThirdParty,Monitoring cloud;
```

Narrative: End-users access the Next.js single-page application in their browser. The front-end (UI) is served by Vercel, which handles static assets and server-side rendering as needed. The design system ensures consistent look-and-feel across dashboard modules (Dashboard, Revenue, Expenses, Bank, Credit Card, Forecast, etc.), and the app supports advanced UI features like keyboard navigation and dynamic theming (using OKLCH/APCA color standards). User interactions (e.g., viewing charts or editing budgets) trigger calls to Next.js API routes or server actions, which encapsulate the **Intelligence Layer** – the business logic for financial calculations, projections, and data transformations. There is also an **AI Layer** integrated (e.g., for forecasting or categorizing transactions) which might call external AI services or use in-app AI models; these calls are made server-side for security (with API keys stored as secrets).

On the data side, the application uses **PostgreSQL** as the primary database (with Prisma as the ORM). The schema is designed to handle a single user in Phase 1, but anticipates multi-tenant data for Phase 2 (e.g. tables include a `tenant_id` or `user_id` to segregate data). A **Redis** service is used for caching (e.g. caching expensive queries, storing short-lived session data, and coordinating an offline edit queue for the autosave/offline features). Redis may also be utilized as a lightweight job queue to buffer background tasks. The **Third-Party Services** (Open Finance APIs, Belvo, Pluggy for bank data; Stripe/Mercado Pago for payments; Gmail API for receipt parsing) are accessed via secure server-side integrations. For example, a bank sync job may call Belvo's API to fetch transactions, which are then normalized and stored in the database. These external calls require proper scheduling and error handling to respect rate limits and ensure data consistency.

All components are deployed on **serverless infrastructure** by default. The Next.js app is hosted on Vercel (which provides automatic scaling of serverless functions and a global CDN for static content). The PostgreSQL database is a managed service (e.g. Neon, Supabase, or Amazon RDS), and Redis is a managed cache service (e.g. Upstash Redis). This serverless and managed approach minimizes ops overhead, allowing us to focus on product functionality. Below, we detail each aspect of the infrastructure and DevOps plan.

2. Environments and Deployment Workflow

We will maintain **multiple isolated environments**: Development, Staging, and Production. Each environment has its own resources and data isolation to prevent cross-contamination:

- **Development (Dev)**: Used by developers for local testing and experimentation. Developers run the Next.js app locally (e.g. `vercel dev` or `next dev`) connecting to a local Postgres (via Docker) or a isolated dev cloud database. Developers can also use a *shared dev environment* on Vercel for collaboration if needed. In dev, we use sample data and possibly bypass certain integrations (or use sandbox API keys). Data here can be reset frequently. Environment variables for dev are kept in a `.env.local` (not committed) or managed via Vercel's development environment settings.

- **Staging (Test/QA)**: Mirrors the production setup as closely as possible for final testing. The staging environment is deployed on Vercel as a separate deployment (e.g. using a `staging` branch or

Vercel environment alias). It uses a separate **staging database** and **staging Redis** instance. Sensitive integrations use test-mode keys (e.g., Stripe test API keys, sandbox OpenFinance credentials, a test Gmail account). Staging allows internal testers or QA to verify new features and do regression tests with realistic data without impacting production. We enforce **data isolation** – e.g., staging has its own database schema or entirely separate DB instance, so no production data is accessible from staging. If needed, staging data can be periodically refreshed from prod with scrubbing of sensitive info. Staging deployments may be triggered by merging into a `main` or `develop` branch (depending on our git workflow).

- **Production (Prod)**: The live environment for end users. Deployed on Vercel's production environment, backed by the production managed Postgres and Redis. All secrets and API keys here are for live services. We enable stricter monitoring and alerting on prod. Production data is critical, so all migrations and releases are done carefully (e.g. using blue-green or canary deployments if possible via Vercel's mechanisms). Only approved changes (after passing tests and possibly staging verification) are deployed to prod.

- **Preview Deployments**: Thanks to Vercel's integration with Git, every pull request can generate a **Preview Deployment** (an ephemeral environment) with its own unique URL. This allows us to test feature branches in isolation. Preview deployments use the **development configuration** by default – for example, they might connect to the dev or a ephemeral database. To enable full end-to-end testing on previews, we can automate provisioning of temporary databases. Tools like Neon's database branching allow creating short-lived Postgres instances from a base backup for each preview [1]. This means each PR could get a fresh database branch seeded with test data, which is destroyed when the PR is closed. Using database branching and isolated preview URLs ensures that testing multiple feature branches in parallel is safe and reproducible, without manual setup.

Environment Configuration & Secrets: Each environment has its own set of configuration in environment variables (managed via Vercel's env settings or a .env file in development). For instance, `DATABASE_URL` will point to the dev DB in development, staging DB in staging, etc. API keys like `STRIPE_API_KEY` have distinct values for test vs live. We never hard-code secrets; instead, they are injected via the environment. We validate all required env vars at startup using a schema (with **Zod** or a similar library) to avoid missing configuration in any environment. For example, a small script can use Zod to parse `process.env` and throw a clear error if something like `BELVO_CLIENT_ID` or `JWT_SECRET` is missing or misformatted in the current env.

Deployment Workflow: Developers work on feature branches, writing code and tests. When a feature is ready, they push to GitHub, where a PR triggers a CI pipeline (details in the next section). If tests pass and code is approved, the branch is merged. Merging to the main branch triggers deployment: Vercel will build and deploy the new version to the **Production** environment (for continuous deployment) or to Staging if we follow a staged release process. We could adopt a pattern such as: - Merge to `main` -> auto-deploy to staging, then a manual promotion or separate `prod` branch/tag to deploy to production. - Alternatively, use trunk-based development where `main` is always deployable, and every successful commit to `main` is auto-deployed to production (with feature flags for incomplete features if needed).

During development, the **design system** (if maintained as a separate package or Storybook) can be developed and published (e.g., to npm or a private registry) and versioned, but since it's part of the

monorepo, it can also be referenced directly. In CI, we can build the design system and run tests on it as well, before integrating into the app.

3. CI/CD Pipelines (GitHub Actions & Quality Gates)

We will implement a robust **CI/CD pipeline using GitHub Actions** to ensure code quality and automate deployments. Every code change goes through the pipeline's "quality gates" before it can reach an environment:

- **Continuous Integration (CI) on Pull Requests:** On every pull request or push to any feature branch, the pipeline will run automated checks:
- **Install & Build:** Set up Node (using a specific LTS version, e.g. Node 18), install dependencies (`npm ci`), and compile the project (e.g. `next build --no-production-browser-source-maps` for a test build) to ensure there are no build errors or type errors. We leverage caching (actions/setup-node with dependency caching) to speed up builds.
- **Linters & Static Analysis:** Run ESLint and TypeScript type checking (`npm run lint` and `tsc --noEmit`) to enforce code standards and catch obvious issues. Also run style checks (Prettier or stylelint for Tailwind if applicable).
- **Unit and Integration Tests:** Execute the test suite (`npm test`) which uses a test runner like Jest or Vitest. This includes tests for utility functions, React components, and any backend logic (with Prisma, we can use a test DB or SQLite for tests). We ensure a good coverage on critical modules (e.g., budget calculations, forecasting algorithms).
- **End-to-End Tests (optional):** For critical user flows, we can run e2e tests (e.g. with Playwright or Cypress) against a deployed preview or a local instance started in CI. These might run on merges to staging or nightly due to length.

- **Code Coverage and Quality Gate:** Optionally, enforce a minimum code coverage. The pipeline can upload coverage reports and fail if coverage drops below a threshold, ensuring we don't merge untested code for critical parts.

- **Security Scanning in CI:** We integrate security tools into the pipeline to catch vulnerabilities early:

- **Dependency Vulnerability Scan:** Use Snyk or npm audit to scan `package-lock.json` for known vulnerabilities. For example, a Snyk GitHub Action can run `snyk test` on the project. If high-severity vulns are found in dependencies, the build can fail or at least warn.
- **Static Application Security Testing:** Enable **CodeQL** analysis on the repository [2] [3]. GitHub CodeQL scans the code for security issues (e.g., SQL injection, XSS, insecure use of APIs) and surfaces alerts in the Security tab. We add GitHub's CodeQL workflow (which runs language-appropriate queries for JavaScript/TypeScript) on a schedule and on pull requests. This provides a Software Composition Analysis (SCA) as well as custom query scanning. Results must show no new critical issues before merging.
- **Secret Scanning:** Use a tool like **gitleaks** in CI to detect if any secrets (API keys, credentials) were accidentally committed. The pipeline can run gitleaks with a ruleset to scan the diff or entire repo and fail if it finds sensitive patterns. This helps enforce that secrets are only in secure storage, not in code.

- **Container/Image Scan (if applicable):** Although we are using Vercel (no custom container by default), if we build any Docker images (for jobs or later phases), we would use **Trivy** or similar to scan them for OS package vulnerabilities. Trivy can also generate an SBOM (Software Bill of Materials) for our app, listing all dependencies. (Both Snyk and Trivy support SBOM generation and scanning [4].)

- **Continuous Delivery (CD):** After CI passes on the main branch, we deploy:

- For **Staging**: We can set up an action that, on push to `main`, triggers a deployment to the staging environment. If using Vercel, we might rely on Vercel's git integration: Vercel can automatically deploy the `main` branch to a staging URL or even directly to production. Alternatively, we use the Vercel CLI in a CI step to push the new build (e.g., `vercel --prod` for production). We ensure environment variables are properly configured in Vercel for each environment (Vercel allows defining env vars for Development, Preview, Production).

- For **Production**: We might choose a manual promotion (to reduce risk of accidental deploys). For example, require a git tag or GitHub Release to deploy to production, or use a protected branch. In a simple setup, merging to `main` could directly deploy to production on Vercel's production URL. We will use Vercel's **Instant Rollback** feature as needed – if a deployment is found to cause issues, Vercel allows quickly reverting to the previous build.

- **CI/CD Pipeline Definition:** The pipeline is defined in YAML under `.github/workflows`. Below is a simplified example of a GitHub Actions workflow file that incorporates some of the above:

```
# .github/workflows/ci.yml
name: CI Pipeline
on:
  push: [main]
  pull_request: []
jobs:
  build-test-and-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
          cache: "npm"
      - name: Install dependencies
        run: npm ci
      - name: Lint and Type-check
        run: npm run lint && npm run type-check
      - name: Run Tests
        run: npm run test:ci  # run tests in CI mode (no watch)
      - name: Check Coverage
        run: npx coverage-check --threshold=80
      - name: Snyk Dependency Scan
```

```
      uses: snyk/actions/node@v1
      with:
        command: test
      env:
        SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
    - name: CodeQL Init
      uses: github/codeql-action/init@v3
      with:
        languages: javascript
    - name: CodeQL Analyze
      uses: github/codeql-action/analyze@v3
    - name: Run gitleaks (Secret Scan)
      uses: zricethezav/gitleaks-action@v2
      with:
        config: gitleaks.toml
    # (Deployment steps or separate job can be added for staging/prod)
```

(The above is a condensed illustration: in practice, we might split jobs for parallelism, e.g., run tests and lint in one job, security scans in another. Also, deploying might be in a separate workflow triggered after main pipeline succeeds, or via Vercel integration.)

- **Quality Gates:** The branch protection settings on GitHub will require that the CI pipeline passes before pull requests can be merged. This means tests must be green, and security scans show no critical issues. We treat any CI failure as a stop; developers must fix the issue and push updates before merge. Only when the code meets quality criteria do we allow deployment. This gatekeeping ensures a high level of code quality in the main branch.

- **Continuous Monitoring in CI:** We also plan to incorporate **Dependabot** for automated dependency updates. Dependabot PRs will undergo the same CI, so we catch incompatible updates safely. Additionally, we can schedule a **nightly CI run** (using GitHub Actions `schedule`) to run full scans (like a deeper Snyk scan or license compliance checks, SBOM generation, etc.) to catch issues that might not be tied to a specific PR.

4. Runtime Topology Options and Justification

By default, we target **Vercel** for hosting the Next.js application, as it offers an excellent serverless platform optimized for Next.js. Vercel provides ease of deployment (just push code, and it builds and deploys automatically), a global edge network for assets, and built-in support for Next.js features like ISR (Incremental Static Regeneration) and serverless API routes. It also supports custom domains, SSL, and auto-scaling without explicit configuration. For Phase 1 (single-user, low-to-medium traffic), Vercel's serverless functions are sufficient for our backend needs.

Vercel – Pros and Cons: Vercel excels at running **lightweight backend functions** – things like API routes that quickly query the database or perform small tasks per request [5]. It automatically scales these functions horizontally to handle concurrent requests. However, Vercel imposes limits on execution time and cannot run truly background or long-lived processes easily. On the Hobby plan, functions time out around

10 seconds, and even on Pro plan, a Node serverless function maxes out at ~300 seconds by default (configurable up to 800s on Pro/Enterprise) [6]. Vercel functions are stateless and ephemeral – which is fine for request/response cycles, but not for jobs that must run continuously in the background or maintain persistent connections. **In short: Vercel works for our front-end and HTTP APIs, but if we later require long-running workers (for heavy data crunching, live WebSocket servers, etc.), we need an alternative or augmentation** [7].

Alternative Runtime Options: We evaluate a few alternatives for running the application or parts of it, especially anticipating Phase 2:

- **Fly.io:** Fly.io allows us to run Dockerized applications (e.g., a Node.js server or Go service) on their global platform with close-to-metal performance. We could package the Next.js app in a Node server and run it on Fly with a persistent process. Fly supports TCP connections and long-lived processes, which could handle WebSocket notifications or background job workers better than Vercel. It also allows us to run multiple regions easily for low latency. The trade-off is more DevOps overhead (we'd manage Docker, health checks, scaling rules) and losing some Vercel niceties (like automatic Next.js optimization). Fly might be considered in Phase 2 for running separate worker services or if we outgrow Vercel's limits.

- **Render.com or Heroku:** These provide a simpler managed platform for containers or Node servers. Render can auto-deploy from git, similar to Vercel, but runs your app as a persistent service. We could use Render to host a Next.js app with SSR, but it might not scale as effortlessly as Vercel's edge network for static content. Render could be useful for hosting a separate backend service (for example, a Python ML service in the AI layer or a background job worker). The cost model on Render is different (pay for instance uptime vs. Vercel pay-per-use). Given our default stack is heavily Next.js, sticking to Vercel in the beginning is likely more efficient.

- **Google Cloud Run / AWS Lambda (self-managed):** Cloud Run allows running a container on demand, scaling to 0 like serverless, but with more flexibility on runtime and up to 15 minutes execution per request. We could containerize the Next.js app or especially containerize a separate worker service for background jobs. Using Cloud Run (or AWS Lambda with custom deployment) would require us to set up CI to build and push images and manage secrets via GCP/AWS. This gives more control (e.g., choose memory/CPU precisely, no cold start issues beyond initial image start, and possibly cheaper at scale), but requires more engineering time. For Phase 1, this is likely not needed, but it's a viable path if we need to **own the backend infra** more directly or avoid Vercel costs.

Recommendation: We will **deploy the Next.js front-end and API on Vercel** by default, leveraging its serverless scaling and minimal ops burden. The managed Postgres DB (e.g., Supabase or Neon) and a managed Redis will complement this. For background jobs (next section), we'll utilize serverless-friendly approaches (like scheduled functions or external job queues). As the app grows (Phase 2+), if we encounter the limitations of Vercel (e.g., needing long-lived services, high memory tasks, or lower-level network control), we will consider migrating those parts to a more suitable platform. For example, we might keep the user-facing frontend on Vercel for speed, but run an auxiliary **worker service on Fly.io or Kubernetes** to handle heavy background processing. This hybrid approach ensures we use the right tool for each job.

In summary, **Vercel covers our needs in the short term** – fast, serverless deployment and scaling for the web app. We justify not immediately using more complex infrastructure (Kubernetes, etc.) because of YAGNI

(you aren't gonna need it) at small scale and the maintenance cost. But we keep the option open: the architecture is designed such that swapping out the runtime (e.g., moving to a container-based deploy) or splitting components is feasible without a complete rewrite.

5. Background Jobs and Queues

The application has several **background processing needs**: data ingestion sync (pulling transactions from external APIs on a schedule), running financial projections and forecasts, periodic data quality (DQ) checks, sending email alerts or reports, etc. In a traditional setup, we'd implement a job worker or cron service to handle these outside of the request/response cycle. On a serverless platform like Vercel, we must approach background jobs carefully due to the stateless and time-limited nature of functions.

Scheduled Jobs (Cron): Vercel now supports Cron Jobs for Serverless Functions [8] [9]. We can schedule an API route (for example, `/api/cron/daily-sync`) to be invoked periodically. In our `vercel.json` configuration, we will define cron schedules for various tasks: - e.g., `"cron": [{"path": "/api/cron/nightly-batch", "schedule": "0 3 * * *"}]` to run a nightly job at 3 AM UTC. Vercel's system will trigger a GET request to that endpoint on schedule. Inside that API route, we can implement logic (e.g., call the OpenFinance API to fetch the latest transactions for the user, or run forecast calculations).

Because serverless functions have limited execution time, **long jobs need to be broken down**. For instance, if fetching transactions from an API for 5 banks could take a while, the cron function might fetch for one bank at a time, or dispatch sub-tasks.

Queue Mechanism: To handle more complex or long-running jobs, we introduce a queue using either Redis or a third-party service: - We can use **Redis** as a job queue by pushing job payloads into a list (e.g., list of bank accounts to sync) and having a function pop and process them. However, without a persistent worker process, we'd need to orchestrate invocation of the worker. One approach is to have the scheduled function process a certain number of items each time it's invoked (say the cron runs every minute and processes up to N queued jobs from Redis). - Alternatively, leverage **Upstash QStash** or similar: QStash is a serverless message queue that can **POST messages to your endpoints with retries**. For example, when a user triggers an import of data, instead of processing synchronously, we enqueue a job to QStash which will call our `/api/processData` endpoint asynchronously. QStash ensures delivery and will retry on failure automatically, which increases reliability [10] [11]. With this approach, the user immediately gets feedback ("your data is being processed") and the heavy work happens out-of-band, triggered by QStash. - For simple periodic tasks, we might not need a complex queue; Vercel cron hitting our API and perhaps looping internally could suffice as long as we mind the timeouts.

Idempotency: It's critical that jobs be designed to handle duplicates or retries without causing inconsistent data. We implement **idempotency keys** or checks. For example, if a "sync bank transactions" job runs, it can record the last synced transaction ID or timestamp in the DB; if it accidentally runs twice, the second run will see there's no new data and do nothing. If a job involves multiple steps (fetch -> process -> save), and it fails midway, the next run should be able to resume safely. We might maintain a `jobs` table to track job status (pending, completed, last run time, etc.), or use Redis locks to ensure only one instance runs certain critical jobs at a time.

Retries & Error Handling: With Vercel cron or QStash, we get some retry mechanism out of the box (e.g., QStash will retry a few times if the endpoint returns an error). We also implement our own retry logic when calling external APIs that might fail transiently. For example, if a call to Stripe's API fails due to rate limiting, the job function should catch that and schedule itself to try again after a delay. In a serverless context, a pattern is to respond with an error and rely on the scheduler's retry, or re-queue a follow-up job for later. We will ensure any background task that fails will log an error (to Sentry) and either mark the job as failed (for manual intervention) or schedule a retry. Tasks should be designed as small units so that a retry won't repeat a huge amount of work (avoid giant monolithic jobs that are hard to retry).

Metrics & Monitoring for Jobs: Each background job will emit logs and metrics. For instance, on job start and completion, log how many items were processed and in how much time. We will integrate these with monitoring (e.g., sending a custom event to an analytics or monitoring system, or at minimum, having Sentry capture a breadcrumb or message). If a job consistently fails, it should trigger an alert (covered in Monitoring section). Additionally, we might implement a lightweight in-app dashboard for admin to see job runs history (e.g., a "Jobs" section listing last run timestamp, success/fail status, etc., since a DB-backed job log table can record these).

Example – Autosave Offline Queue: The app supports offline edits: changes are queued locally and then synced when online. On the backend, when those queued changes arrive (perhaps bundled via an API call), we treat them as a batch job: apply each change in the database under a single transaction or in sequence. We might push these changes into a background processor (to not lock up the main thread). However, since this needs to happen when the user comes online and expects quick feedback, we likely handle small offline changes inline (the user's device may send them one by one). For larger offline syncs (if user was offline and did many operations), we could accept them and quickly enqueue processing, then respond that "data is being saved" and let a job finalize it. The design must ensure consistency (perhaps using a "change-set ledger" where each change is recorded and requires approval – the "write approval required" feature likely means some changes only take effect after an approval step, so those might naturally be queued in a pending state in DB).

Email and Notification Jobs: If sending emails (like summary reports or alerts), we will not send directly in an API request. Instead, we use a job. For instance, schedule a daily job to send a summary email via Gmail API or a service like SendGrid. The job would iterate through the relevant data and send emails. This will be done via a cron-triggered function or a queued task triggered by some event (e.g., user enabled a certain alert).

Scaling the Job System: For Phase 1 with one user, the job load is minimal. As we scale to many users, we will likely need a dedicated background job runner service. At that point (Phase 2 or beyond), we might deploy a worker on a separate platform (e.g., a small Node.js worker on Fly.io or AWS Fargate) that continuously pulls from a job queue (Redis or a message broker). This worker could handle jobs more robustly than the serverless approach. But until the load requires it, we can manage with scheduled triggers and on-demand tasks using the serverless approach to keep things simple.

Metrics for Background Jobs: We will keep track of key metrics such as: - Number of jobs run, success vs failure count. - Duration of jobs (to spot if something is taking too long). - Frequency of certain tasks (to ensure they run as scheduled). - We can use a simple approach like logging these and using a monitoring tool to create a dashboard, or integrate OpenTelemetry (instrument the job functions to emit trace spans).

To summarize, **background tasks will be handled by scheduled serverless functions and an asynchronous queueing mechanism** in our stack. We ensure reliability through idempotency and retries. This design works within Vercel's constraints while the app is small. We also plan the path to a more scalable solution (dedicated workers) as an extension when needed.

6. Database Operations (Migrations, RLS, Branching, Rollbacks)

Our primary database is PostgreSQL, with Prisma ORM managing the schema and migrations. We adopt a disciplined approach to database changes and access control:

- **Migrations with Prisma:** The schema is defined in Prisma's `schema.prisma`. Whenever we need to change the schema (create a new table for a new module like "Category Mapper" or alter a column), we create a migration using Prisma Migrate (`npx prisma migrate dev` for local dev which creates a SQL migration file, and `prisma migrate deploy` to apply in production). Migrations are version-controlled. In CI, we can run `prisma migrate deploy` against a test database to ensure migrations are reproducible. We treat the migration history as an append-only log. In production, migration application is part of the deployment process (for example, run migrations **before** deploying the new code that depends on them, or use a strategy to handle backwards compatibility if deploying first – typically we do backwards-compatible DB changes when possible).

- **Rollback Strategy:** Prisma's migrate doesn't natively support automatic down-migrations (it assumes roll-forward). To handle rollbacks, we have a few strategies:

- For simple cases (deployment fails after a migration was applied), the fastest "rollback" is to redeploy code that is compatible with the current schema (possibly the previous version) if the migration didn't irreversibly change data.
- We can manually write down migration scripts for critical changes (e.g., if we drop a column, have a way to put it back if needed quickly).
- In addition, having database backups (point-in-time recovery) means if a catastrophic migration goes wrong, as a last resort we could restore the database to a point before the migration and re-run the deployment (accepting some downtime depending on RTO).
- To minimize the need for rollback, practice deploying non-breaking changes: e.g., instead of renaming a column directly, add new column, migrate data in code, then later drop old column once not used.

- **Branch-based migration testing:** When a developer creates a migration, they test it on their local or dev DB. We might also spin up a ephemeral database (using Neon or Supabase) for the CI run of that branch to apply migrations and run tests, ensuring the migrations won't fail on production. (Neon's branching allows quickly forking the production schema to test applying new migrations on real data copy [1], which is extremely useful to gain confidence.)

- **Row-Level Security (RLS) for Multi-Tenancy:** In Phase 2, when we support multiple users/tenants on the same database (assuming a shared schema, multi-tenant design), we will enforce data isolation at the database level using Postgres RLS policies. **Row-Level Security** allows us to define policies on tables such that a user (or tenant) can only see their own rows [12] [13]. For example, we might set up each user's session such that upon connection we `SET role = 'user_<ID>'` or use

a `current_user` or JWT claim that the RLS policy checks. A policy on the `transactions` table might be: *"tenant_id = current_setting('app.current_tenant')"*, ensuring every query only affects that tenant's data. This way, even if our application code has a bug and tries to query all data, the database will automatically filter it. RLS serves as a safety net to **centralize isolation enforcement** and ease the burden on developers to always remember tenancy filters [14]. In our code, we'll integrate Prisma with RLS by ensuring that upon each request, after verifying the user, we set the appropriate tenant context (there are Prisma middleware or Postgres `SET` commands we can execute via Prisma). We'll test RLS thoroughly in staging, as it can be tricky (making sure admin roles can still see all data, etc.). Phase 1 (single user) likely doesn't need RLS yet, but we'll keep the schema ready (every table has a user/tenant id) so that enabling RLS is straightforward later.

- **Database Branching & Preview Databases:** As touched on in Environments, using a database that supports branching (like Neon or PlanetScale for MySQL) significantly enhances our development workflow. We could implement ephemeral databases for preview deployments – for instance, when a developer opens a PR, our CI could call Neon's API to create a new database branch from the latest production data (with sensitive data masked). The app in that preview URL would use that branch's connection string. This means even schema changes can be tested in an isolated environment that has realistic data. On merge or PR close, the branch is torn down. Neon's copy-on-write branches spin up very fast and cost-efficiently [1]. If not using Neon, we can approximate this by having a single "preview" database that is migrated and used for all previews with test data, but that's less ideal if multiple PRs need different schema states. For now, we plan to at least maintain separate **Dev/Staging/Prod** databases. The branching approach is a nice-to-have that we will evaluate.

- **Managing Data Changes and Seeds:** When the app starts (in development or in tests), we can use Prisma's seeding feature to populate some reference data (like default budget categories, etc.). In production, real data will be created through the app or migrations. We might have occasional data migrations (e.g., backfilling a new column for all existing records). Such data migrations can be done via either a Prisma script or in raw SQL inside a migration file. We must be cautious with large data operations; if needed, do them in batches to avoid locking the DB for too long.

- **Database Access and Performance:** We will enable PG Bouncer or connection pooling as needed (many serverless functions could open many connections; using a pooler like Neon's built-in or Supabase's pooler helps maintain efficiency). We also consider using read replicas if certain read queries (like analytics for charts) become heavy – but for Phase 1, a single instance is fine. The Prisma client should be used in a long-lived context if possible (with Vercel serverless, we might not have long-lived processes, but we can optimize by reusing the Prisma instance across calls if supported via Next.js App Router's built-in request reuse or using the Edge runtime for specific cases).

- **Role-Based Access Control (beyond RLS):** We may create multiple database roles:

- An application role with minimal rights (only can CRUD within its schema, no superuser).
- Perhaps a read-only role for analytics or for the "DB Viewer" module if we allow some safe querying.
- If RLS is on, the app role is the one that has RLS policies applied. An admin role could bypass RLS (for debugging, but that should only be used in admin contexts).
- We will use Prisma's capabilities or direct SQL to manage roles and grants as part of migrations (for example, create a read-only role and grant select on certain views).

In summary, our DB ops strategy revolves around **safe migrations, strong isolation, and flexible dev/test branching**. We treat the database as an integral part of the DevOps cycle, not something static. Proper tooling (Prisma, branching, backups) ensures we can evolve the schema confidently and recover from issues.

7. Secrets and API Tokens Management

Handling secrets securely is paramount, especially for a financial application that integrates with sensitive APIs. We follow best practices for **Secrets Management**:

- **Centralized Storage:** All secrets (API keys, database credentials, third-party tokens) are stored in the environment configuration of the deployment platform or a secret manager – never in code or config files that are committed. On Vercel, we will use the built-in Environment Variables feature for Production, Preview, and Development envs. Each secret is added via the Vercel dashboard or `vercel-cli` and is encrypted at rest by Vercel. For local development, developers keep a `.env` file that is listed in `.gitignore` (never committed). Alternatively, use a tool like Doppler or 1Password Integrations to share dev secrets securely among the team.

- **Examples and Validation:** We maintain a `.env.example` (with dummy values) to document the required configuration keys. We also use **Zod for validation**: define a schema of expected env vars with types (e.g., `PORT: z.string().regex(/^\d+$/)`, `NODE_ENV: z.enum(['development','production','test'])`, etc.) and call `schema.parse(process.env)` on server startup. This fails fast if something is misconfigured, avoiding subtle bugs due to missing keys [15]. It's especially useful across different environments — we catch if, say, in staging we forgot to set the Gmail API credentials, as the app will throw on boot with a clear message.

- **Rotation of Secrets:** We plan for periodic rotation of sensitive keys:

- Database passwords can be rotated by generating a new credential and updating the env var (ensuring the old one remains valid until all services use the new one, then revoke old).
- API keys (Stripe, etc.) should be rotated especially if an incident occurs (e.g., if we suspect a leak, immediately revoke and replace).
- We could use a secrets manager like AWS Secrets Manager or HashiCorp Vault in a larger setup, but with Vercel, we rely on their env management. For rotation, it might be manual unless we script calls to update Vercel env via API. Given a small team, manual rotation scheduled (e.g., every 90 days for certain keys) is acceptable.

- For user OAuth tokens (like Gmail API tokens for reading receipts), we store refresh tokens encrypted in our database (using a key stored in env for encryption) and retrieve new access tokens as needed. Those user tokens have their own expiry and rotation mechanism by design (OAuth refresh flow).

- **KMS (Key Management Service):** For an extra layer, if we had to handle extremely sensitive data or our own encryption keys, we'd use a KMS. For instance, if we encrypt user financial data in the database, we might use AWS KMS to store the data encryption key and only allow the app to decrypt when needed. In Phase 1, this might be overkill, but it's noted for future. Another scenario: sign/

verify webhooks (Stripe webhooks secret) – that secret is in env, but for rotating it, we could fetch from a secure store at runtime if required.

- **Secret Injection and Scope:** We ensure that secrets are only available to the parts of the app that need them. For example, API route functions can access server-side secrets (like OPENFINANCE_API_KEY) but these are never exposed to the client. Next.js provides `NEXT_PUBLIC_` prefix convention – only env vars with that prefix get exposed to front-end. We are careful to only prefix truly safe values (e.g., a Google Maps public API key). Private keys (like database URL, API secrets) do **not** use NEXT_PUBLIC and thus never reach the client bundle.

- **Token Handling & Validation:** Within the app, we use short-lived tokens where possible:

- For user authentication (if any login system exists in Phase 2), use JWTs or NextAuth – tokens signed with a secret that rotates periodically (we'd have a primary and secondary signing key during rotation to avoid logout of all users immediately).
- The app might interact with OpenAI or other AI APIs in the AI Layer; those API keys are sensitive and rate-limited. We keep them in env and may implement an internal usage limit to avoid abusing them or getting unexpectedly large bills.

- For outgoing webhooks or interactions (like receiving webhooks from Stripe), we validate signatures using the secret (e.g., Stripe webhook secret from env) to ensure authenticity.

- **Security Scanning for Secrets:** As mentioned, we run **gitleaks** to ensure no secret accidentally got committed. We also enable GitHub's Secret Scanning on the repo (which can detect patterns of common API keys in any commit and alert us). This is an important safeguard especially with many API integrations.

- **Example** `.env.example`**:** (illustrative)

```
# .env.example – Document required env variables (values are placeholders)

# Basic environment settings
NODE_ENV=production
NEXT_PUBLIC_APP_NAME="FinDash"

# Database
DATABASE_URL="postgres://username:password@host:5432/database"
SHADOW_DATABASE_URL="postgres://username:password@host:5432/database_shadow"  #
for Prisma migrations

# Cache
REDIS_URL="redis://user:pass@host:6379"

# API keys for third-party integrations (use test keys in non-prod)
OPENFINANCE_API_KEY="sk_test_yourkey"
BELVO_CLIENT_ID="your-belvo-id"
```

```
BELVO_SECRET="your-belvo-secret"
STRIPE_SECRET_KEY="sk_test_stripe..."
STRIPE_WEBHOOK_SECRET="whsec_..."
MERCADOPAGO_ACCESS_TOKEN="TEST-abc123..."
GOOGLE_OAUTH_CLIENT_ID="your-google-oauth-client-id"
GOOGLE_OAUTH_CLIENT_SECRET="your-google-secret"
# etc...

# Application secrets
JWT_SECRET="randombytesforjwt"
ENCRYPTION_KEY="base64encodedkeyforencryptingdata"
SENTRY_DSN="https://your-sentry-key.ingest.sentry.io/project"
```

This file would be updated as new integrations are added. The actual secure values are set in each environment (dev/staging/prod) separately.

- **Runtime Secret Access:** Since we're mostly serverless, on each function invocation the necessary secrets are already in environment variables. We avoid passing secrets via client. If any config needs to be exposed to client (like feature flags or non-sensitive config), we may provide an API that returns it or embed in the build (for example, a list of supported bank names might be public).

- **Audit and Rotation Policy:** We will maintain a log of who has access to production secrets (likely only a couple of senior devs). When someone leaves the team, all secrets they had knowledge of should be rotated. We plan periodic audits (ensuring that no unused secrets linger in config, and checking that all secrets in config are actually used by the app – this prevents secret sprawl).

- **Input Validation with Zod:** Not only environment variables, we use Zod to validate any external data coming into the system, including API request bodies. This ensures that our APIs are robust (e.g., if a background job endpoint receives a payload, it should validate it's well-formed to avoid poison messages causing errors). This is more application-level, but worth mentioning as part of secure and reliable operations.

8. Monitoring, Logging, and Tracing

To maintain a **high level of reliability**, we set up comprehensive monitoring for the application:

- **Error Monitoring (Sentry):** We integrate Sentry for both front-end and back-end error tracking. On the Next.js app, we use the official `@sentry/nextjs` SDK, which automatically instruments both the client and server. This will capture any exceptions, unhandled promise rejections, etc., and send them to Sentry. We configure environment tagging (so we know if an error came from dev, staging, prod) and release version tagging (each deployment can be tagged with a git commit or semver). Sentry will alert us (via email/Slack) on new errors or spikes of errors in production. This helps us catch runtime issues that slipped past tests (e.g., an edge-case in parsing bank data). We also use Sentry's performance monitoring to track API response times and front-end page load times – this gives us APM-lite capabilities (like tracing an operation from the client click through the server function to the DB query, if configured with distributed tracing).

- **Structured Logging:** While Sentry is great for exceptions, we also need logs for general debugging and audit. Vercel captures `console.log` output in function logs accessible in their dashboard. We will use structured logs (JSON) for key events, which can later be aggregated if needed. For example, logging an info message when a background job runs: `{ "event": "job_run", "job": "daily_sync", "status": "success", "duration_ms": 5200 }`. If we outgrow Vercel's basic logging, we might integrate an external log service (e.g., Logtail, Datadog, or ELK stack). In the short term, we can export Vercel logs periodically or use their `vercel logs` CLI to inspect issues.

- **OpenTelemetry Tracing:** For deeper **distributed tracing**, we can instrument our services with OpenTelemetry (OTEL). Given the simplicity (a single Next.js service and a DB), full tracing might be optional. However, if we integrate with external services or have background workers, OTEL can trace a request across components. For instance, a request comes in -> triggers a background job -> job queries DB and calls an API. We can propagate a trace ID and have each log or error include it. Some of this we get via Sentry (since Sentry can act as a tracing system too). If needed, we could run a lightweight OTEL Collector and use a tracing backend (Jaeger, Zipkin, or a vendor like DataDog APM). For now, we note this as an extensible area.

- **Metrics and Dashboards:** We will define key **Service Level Indicators (SLIs)** to monitor:

- Uptime of the app (we can use an external ping service or Vercel's checks).
- Response time of key API endpoints (average and p95 latency).
- Error rate (number of 5xx responses or unhandled exceptions).
- Resource usage: Since on serverless we can't directly see CPU, etc., we rely on indirect metrics like function invocation count and duration. Vercel's analytics or our own monitoring can track how many times a cron job ran successfully, how many DB queries per minute, etc.
- Custom metrics: e.g., number of transactions synced per hour, or count of active users.

Since we may not have a dedicated metrics infrastructure in Phase 1, we can use a combination of: - **Vercel Analytics** for web vitals (if using Next's built-in web-vitals collection or Vercel's frontend analytics). - **Database Monitoring:** The managed DB likely provides some metrics (CPU, connections, cache hit rate). We ensure alerts on the DB: e.g., if CPU goes beyond 80% or connections near limit, send alert. If using Supabase, their dashboard would show this; for RDS, we'd rely on CloudWatch metrics. - **Cron Monitoring:** For scheduled jobs, it's good to have an external monitor to ensure they actually run. A service like Cronitor or Healthchecks.io can ping an endpoint which our jobs call at start or end. If a job doesn't run on time, we get alerted. Alternatively, configure Sentry's cron monitoring feature (Sentry can notify if a scheduled job doesn't report in a timeframe).

- **Alerts:** We set up alerts for the above metrics:
- If the uptime check fails (site is down) – immediate page to on-call (if we have on-call rotation; in a small team likely an SMS/Slack to all).
- If error rate spikes above some threshold or a critical error appears (e.g., database connection failure, out-of-memory, etc.) – alert the team via Slack/email.
- Performance degradation: e.g., p95 response time goes above 1s for 5 minutes – trigger an investigation alert (not wake someone at 3am unless it's impacting users significantly).

- Cost anomaly: if possible, track usage that correlates to cost (like function invocations or DB/storage usage) and alert if suddenly double normal – could indicate something runaway (infinite loop, abusive traffic, etc.).

- **Logging sensitive data:** We will **sanitize logs** to avoid leaking PII or secrets. For instance, we won't log full bank API responses or user personal data except when necessary, and even then, scrub identifiers. Sentry has data scrubbing rules to remove things like credit card numbers or auth tokens from events.

- **User Activity Logging:** For the "ledger" of changes or audit trails, we might store some logs in the database (for business auditing, not only technical monitoring). E.g., track who changed a budget amount and when (especially in multi-tenant, this is important). Those application-level logs are separate from system logs and will be part of the feature design (perhaps visible in the UI for an audit trail).

- **Real-time Monitoring Tools:** In case of diagnosing issues, we might use tools like Vercel's live logs, or connect a Node inspector if running locally. We will also ensure our CI pipeline has tests for memory leaks or heavy CPU usage in critical loops (maybe using profiling tools in development).

- **Capacity Planning from Metrics:** Over time, monitoring data will show our average and peak loads. For example, if we see the DB CPU steadily rising with more users, we know when to scale the DB instance size or add an index. Similarly, if serverless function duration is high, we might adjust our approach.

By implementing this monitoring stack early, we can meet our SLOs (see Acceptance Criteria section) and quickly respond to incidents with good observability into the system's behavior.

9. Backup and Disaster Recovery

Data backups and disaster recovery are especially important in a financial application (to prevent data loss of transactions, budgets, etc.). We plan for robust backups and defined RPO/RTO goals:

- **Database Backups (PITR):** Our managed Postgres will have daily automated backups enabled. For example, on Supabase Pro plan, daily backups are standard with 7-day retention, and **Point-in-Time Recovery (PITR)** can be enabled as an add-on [16]. PITR means the database continuously logs changes (WAL) allowing restore to any specific second within the retention window, achieving an RPO (Recovery Point Objective) as low as ~2 minutes [17]. We will likely enable PITR for production data given its importance. This means if something goes wrong (say accidental mass deletion), we can restore the DB to a state just before the incident with minimal data loss (only a couple minutes of data might be lost in worst case). The RPO we aim for is **< 5 minutes** of data loss for prod. For dev/staging, RPO can be more since those are not critical.

- **Backup Storage and Verification:** The backups (full dumps or base backups + WAL segments) are stored by the provider (e.g., Supabase or AWS S3 if using RDS). We will also periodically take manual snapshots before major changes, just for safety. Importantly, we will **test our backups**: at least in staging, perform a restore from backup to ensure the process works and backups are not corrupted. Too often backups are taken but never tested. We can automate a quarterly drill: spin up a new DB instance from a backup and run a subset of tests or data checks on it.

- **File Storage Backups:** If the app stored files (like receipt images or exports) in something like S3, we would also version or backup those. In our case, most data is structured in the DB. Perhaps user uploaded attachments might be a feature; if so, enabling S3 versioning or periodic backup of the bucket is needed. For now, we primarily focus on DB.

- **Redis Data:** We use Redis mostly as cache or transient queue, so losing it isn't catastrophic (we can rebuild cache). We might not back up Redis at all (for performance reasons), or rely on its AOF snapshot if the provider offers persistence. But since it's not our source of truth, we are fine with RPO = time of failure for cache (i.e., if Redis fails, we lose recent sessions but users can log in again, etc.). If we did use Redis for any persistent data in future (like a job schedule state), we'd then consider its backup (like an RDB snapshot daily).

- **Disaster Recovery Plan:** We define scenarios:

- *Minor Outage:* e.g., the database instance goes down due to cloud zone issue. For high availability, we might choose a provider that has HA (like AWS Aurora or a primary-read-replica that can be promoted). Supabase for instance at the moment doesn't auto-failover across regions, but we could restore quickly in the same region. Our RTO (Recovery Time Objective) for such a minor outage is maybe **< 15 minutes** (the time to failover or restore service). This might involve promoting a replica or restoring from the latest WAL to a new instance.
- *Major Outage (region-wide or cloud-wide):* If an entire region of Vercel or our DB provider is down, we might face longer downtime. For mitigation, we consider cross-region backups and ability to redeploy to a different region. For example, Neon allows creating branches in different regions; or use a read replica in another region for DR. We set RTO for catastrophic scenario perhaps **< 4 hours**, acknowledging that in worst case we might have to restore from backup in a new region and repoint the app.
- *Data corruption/user error:* If a bad code deploy corrupts data or an admin deletes something by mistake, we rely on PITR to rewind. RPO in this case is how far back we need to go; maybe a few minutes if caught quickly. We'd lose minimal data but that might be acceptable vs keeping corrupted data.

- *Code rollback:* If the application code is causing issues (but data is fine), Vercel's instant rollback helps – we can revert to previous deployment in seconds. This is part of our DR approach for code-level incidents.

- **Drills:** We will create an **Incident Response runbook** (see Runbooks section) which includes procedures for restoring from backup. The team will do drills, e.g., simulate "DB got wiped at 2am" and execute the steps to rebuild or restore in staging environment. This prepares us for real incidents so we aren't learning under pressure. We also verify that our backup retention is sufficient (maybe keep daily backups for 7 days and weekly for a month, etc., depending on business needs).

- **Disaster Recovery for Third-Party Services:** If a third-party API (like Belvo or Stripe) is down, that's not exactly our system's disaster, but it affects functionality. We consider fallback: e.g., if bank API is down, the app should handle it gracefully (show last cached data with a warning rather than crashing). For receipts via Gmail, if Gmail API is down, queue and retry later. These are more resilience in integration design rather than our infra, but worth noting in continuity planning.

- **Infrastructure as Code and Redeploy:** To speed up recovery, we ensure we can redeploy our infra quickly:

- Re-provisioning Vercel is trivial (code is in git), just need to set envs.
- Re-provision DB: if using a managed service, have scripts or documentation to create a new instance from backup quickly. Possibly maintain Terraform or at least documented steps for provisioning the DB and connecting.
- If our infra expands (say we add Cloud Run or Kubernetes), we will have those as code (Helm, Terraform) such that in DR we can recreate clusters.

- Essentially, treat the setup as reproducible. In Phase 1, because it's simple, manual steps documented might suffice, but we lean towards automation as things grow.

- **Backups of Configuration:** Besides data, we also backup config like environment variables (e.g., we could keep an encrypted copy of our env vars in our repo or a secure vault, so if Vercel's config is lost or if moving to another platform, we have them). It's rare but can save time if accidentally someone deletes the project or something.

In conclusion, we aim for **RPO ~ 0-5 minutes** for critical data and **RTO ~ 15-60 minutes** for most incidents (small scale allows faster response; in a larger scale, might formalize these numbers). By leveraging managed services with strong backup features and practicing recovery, we mitigate the risk of permanent data loss or prolonged downtime.

10. Cost and Scaling Plan

Our infrastructure choices should be cost-effective for the current scale (single user in Phase 1) but able to scale to a multi-tenant SaaS in Phase 2. We outline a plan for capacity planning, cost monitoring, and scaling, along with an estimated cost table:

Current Usage (Phase 1): With one primary user (and maybe a few testers), load is minimal: - Few concurrent requests, small database size (a few thousand records). - We can utilize free tiers or lowest-tier plans initially (e.g., Vercel Hobby, Supabase free).

Phase 2 (SaaS readiness): We might anticipate dozens or hundreds of users, requiring more robust plans: - More function invocations on Vercel, more database load, etc. - Possibly additional costs for background jobs, monitoring, etc.

Headroom and Auto-Scaling: - **Vercel:** automatically scales serverless function instances with traffic (up to large concurrency limits, e.g. 100k concurrent executions on Pro plan) [6] . We don't manually scale Vercel, but we must monitor usage (execution time and bandwidth) since costs can grow usage-based. We ensure headroom by choosing an appropriate plan: Vercel Pro ($20/mo) offers higher limits than Hobby, including longer function timeouts and more concurrent builds. If our usage spikes (e.g., heavy data processing in functions), Vercel charges for extra usage (CPU time). We will keep an eye on the billing page. If we approach Enterprise-level usage (unlikely soon), we'd consider a custom plan or moving heavy compute off-platform. - **Postgres DB:** We choose a plan with enough CPU/RAM for our workload. For one user, a small free or ~$15/mo instance suffices. As we onboard more users, we scale vertically (bump CPU/RAM) or horizontally (add read replicas). Some providers like Neon autoscale the compute on demand (Neon has an

autoscaling concept). If using a fixed instance, we set up alerts on high CPU or connections to upgrade timely. We also ensure storage auto-expansion (most managed DBs do this). - **Redis:** If using a free tier (like Upstash free), it has limits (maybe ~10k commands/day, etc.). For a single user, fine. For many, likely upgrade to a paid plan ($10-20/mo) to get more memory and throughput. Redis usage in our app (caching, small queue) is not huge; the main headroom needed is for caching reads if we get traffic spikes. - **External API costs:** We consider these too: Belvo, Stripe, etc., might charge per API call or have rate limits. We design the app to not over-poll those APIs (use webhooks where possible, e.g., Stripe webhook for new payments instead of polling). Also cache results so we don't call external APIs redundantly (for example, if the dashboard is refreshed 5 times in a minute, don't fetch the same bank data each time from Belvo – fetch once and cache for a few minutes). This not only respects rate limits but controls cost if those APIs cost money per call.

Rate Limiting: - We implement rate limiting on our own API to prevent abuse (especially by Phase 2 when external clients might hit our endpoints). For example, using a package like `next-rate-limit` or Vercel Middleware to limit, say, 100 requests per minute per IP or per user. This ensures no single user can accidentally or maliciously overload our system or rack up external API calls. - For third-party APIs, many have built-in limits (e.g., Gmail API might allow X requests per second). We adhere to their guidelines: possibly adding delays or not parallelizing too much. If we ever approach their limits, consider requesting higher quota or using a backoff strategy. - We'll also design heavy operations (like generating a yearly report PDF, if any) to be done asynchronously and perhaps with an explicit user action rather than on every page load.

Capacity Scaling Path: - If the app usage grows, the first likely bottleneck is the database (since serverless can scale out easily, but a single DB instance can only handle so many QPS). In preparation, we use performance best practices (indexes on frequently queried fields, query optimization via Prisma for n+1 issues, caching results). When nearing limits, we can scale up the DB plan (more vCPU, more RAM). If writes become heavy, consider sharding or at least separating analytics vs OLTP load by using read replicas for heavy read queries (like generating charts). - For Vercel functions, if one particular function is heavy (e.g. an AI analysis that takes 20s), that might not scale well on Vercel due to timeout and cost. For that scenario, moving that to a background job off-platform or using Vercel's Edge Functions (which have different performance characteristics) might help. Vercel also introduced **Fluid Compute** mode which allows reuse of function instances and can reduce cold starts [18] – we'll use such features if beneficial and available on our plan. - We keep an eye on the **latency** for global users. Vercel routes requests to nearest region and we can opt to deploy functions in multiple regions if needed (Pro plan can set regions). The DB, however, is likely single-region. For global scale, we might consider a globally distributed database (or multi-region replicas). That's beyond initial needs but on the radar if we get users in different continents expecting low latency.

Cost Estimation Table (per environment):

Service	Dev (Single Dev)	Staging (Team Testing)	Production (Initial)
Vercel Hosting (Next.js)	Free Hobby plan (0$) – limited builds, function 10s timeout.	Pro plan for team ($20/month) – faster builds, password-protected preview domain.	Pro plan ($20/month) for production domain. Additional usage fees if traffic grows (e.g., [$___ per 100k function invocations] – but initial usage within free quota).

Service	Dev (Single Dev)	Staging (Team Testing)	Production (Initial)
PostgreSQL Database	Dev: free tier (e.g., Supabase free: 500MB, limited throughput) or local Docker (0$).	Small cloud instance ($25/month) – enough for testing with realistic data.	Managed Postgres Standard ($50/month) – e.g., 2vCPU, 4GB RAM, 50GB storage. Enable PITR (add ~$100/month if needed for continuous backups) [16]. Scale up as users increase (next tier ~$100-200/mo for more CPU/RAM).
Redis Cache (Upstash or etc.)	Free tier (limited storage and ops, 0$).	Free or low-tier (maybe $10/mo for higher throughput).	~$20/month managed. Primarily for caching to reduce DB load. Could scale to $50+ if heavy usage (more memory for caching larger data sets).
Background Jobs / Queue	Using Vercel Cron & free Upstash QStash (first 10k messages free).	Same as dev (low usage, within free).	Upstash QStash ~$10/month if volume increases (pricing by messages). Alternatively, minimal cost if using just Vercel Cron (included in Vercel plan, but heavy usage of cron triggers might count towards function invocations cost).
Monitoring & Error Tracking	Sentry free plan (good for dev).	Sentry free (up to quota) or small team plan (~$29/mo) for more users & history.	Sentry – possibly $79/month (for higher event volume and retention) depending on error throughput. Could stick to free if volume is low. Other monitoring (Pingdom/UptimeRobot) could be free or a few dollars per month.
Misc (CI/CD, etc.)	GitHub Actions – included, but watch minutes (if using free GitHub, 2000 min/month free, which is fine).	GitHub Actions – might need paid minutes if many runs (or use self-hosted runner). Likely negligible cost at our scale.	Domain name – ~$10-15/year. Third-party API costs: Stripe free unless transactions (they take % fees), Belvo/Pluggy might charge per connection (e.g., Belvo ~$50/month for X data connections – needs checking), OpenAI API if used (cost per request, likely a few cents each). We will monitor and cap these.

(Note: The above costs are illustrative; actual prices in 2025 may vary. We'll continuously monitor usage and adjust tiers. In early phase, total monthly cost could be under $100. As we scale to many users, expect database and Vercel usage to dominate costs, perhaps a few hundred dollars a month for moderate user counts, scaling to more as needed.)

Scaling Plan: We will use these services efficiently to delay costs until necessary (e.g., only go to a higher DB tier when metrics show >70% capacity regularly). By implementing caching (Redis) and optimizing code,

we reduce unnecessary load (saving cost). We also implement **autoscaling where available**: e.g., if using AWS, enable Aurora Serverless or Cloud Run auto-scaling. With Vercel, autoscale is implicit. For the DB, if using a cloud that supports it (like Aurora with auto-scaling replicas or Neon autoscaling), we enable that to handle burst without manual intervention.

Cost Monitoring: We treat cost as another aspect to monitor: - Use the provider dashboards and maybe set up alerts (many have budget alerts, e.g., an alert if Vercel usage exceeds $X). - Regularly review bills to see if any service is spiking. For instance, if suddenly we see a lot of egress bandwidth (maybe a bug causing large downloads repeatedly), we catch and fix it. - In design, prefer cost-effective patterns: e.g., use streaming responses or pagination to avoid massive data transfers, use CDN caching for static content (Vercel does this by default) to reduce function hits, etc.

Scaling beyond Phase 2: If the app becomes very popular (hundreds of tenants, heavy AI usage, etc.), we might consider migrating parts of the stack: - Database: possibly move to a more scalable solution (managed cluster or splitting by tenant). - App: containerize and use Kubernetes or another PaaS to have more control (especially if we need persistent WebSocket connections for live updates, which Vercel can't easily do at scale). - Caching: introduce a CDN for certain API responses if they can be cached globally (might use Vercel's Edge Cache or a service like Cloudflare for an API caching layer). - At that stage, costs shift but also unit costs might reduce (running our own infra might be cheaper at huge scale, but requires more ops).

In conclusion, our plan is to **start small on costs, iterate and scale as usage grows**, with proactive monitoring to avoid surprises. We always ensure a buffer (headroom) of capacity so that typical peaks (like end-of-month report generation for all users) don't overwhelm the system. The table above provides a rough budget guideline per environment to plan ahead.

11. Security Scanning and SBOM

Security is a continuous concern. We integrate multiple tools and processes to keep the code and dependencies secure, and produce a **Software Bill of Materials (SBOM)** for transparency:

- **SAST & Dependency Scanning:** (Already discussed in CI section) – GitHub CodeQL scans our code for security vulnerabilities (e.g., use of `eval`, potential SQL injection in raw queries, etc.) [2] [3]. Snyk (or npm audit) scans for known vulnerable packages (e.g., if tomorrow a vulnerability in `express` or `next` is announced, we'd know). We treat high-severity findings as release blockers: either upgrade the library or apply patches. Medium/low issues are triaged and fixed as appropriate.

- **Secrets Scanning:** We employ **gitleaks** in CI and also enable GitHub's own secret scanning. This way, if an API key or password ever gets into a commit, we get notified immediately and can invalidate that secret. Our developers are trained never to hardcode secrets, but this serves as a safety net.

- **Container and Infrastructure Scanning:** If we use any Docker images (like for a future worker or if using a Next.js image for Cloud Run), we will use **Trivy** to scan the images for CVEs. Trivy can also scan our code repository (`trivy fs .`) for config issues (e.g., insecure Dockerfiles, etc.) and generate an SBOM (Trivy has an `--format cyclonedx` option to output CycloneDX SBOM). We

might incorporate that into CI to generate an `sbom.xml` on each release. Snyk also has SBOM capabilities, but Trivy's CLI is straightforward [4].

- **SBOM Usage:** The SBOM lists all our dependencies (and their versions and licenses). This is useful for compliance and quickly assessing impact of a new vulnerability (e.g., "log4j" incident style – we could search our SBOM to see if we had that lib). We can publish the SBOM internally or even with our app if we ever distribute on-premises (likely not relevant for SaaS). For now, it's mainly for our internal use to track components.

- **Dynamic Application Security Testing (DAST):** In staging, we could run a security scan of the running app – e.g., use OWASP ZAP or Nikto to scan for common web vulnerabilities. Given Next.js is pretty secure out-of-the-box (against XSS, etc.) and we don't accept SQL via input (thanks to using ORM), the surface is limited. But scanning can catch misconfigurations (like HTTP headers missing). We'll ensure we send security headers (Next can set those, like Content Security Policy, etc.). Automated DAST could be a periodic thing (maybe monthly or pre-release).

- **Infrastructure as Code Scanning:** If we add Terraform or other IaC, we'd use tools like Checkov or Terraform Cloud's scan to ensure no misconfigured cloud resources (like open security groups, etc.). In our current serverless approach, much is managed by the provider, reducing that attack surface (we don't manage VMs or networks directly).

- **Code Review with Security in mind:** We require at least one peer review for PRs, where the reviewer also considers security implications (e.g., "Are we sanitizing this user input?", "Should this API be auth-protected?", etc.). Over time, we might create a checklist for code reviewers to systematically cover security.

- **Penetration Testing:** Before going live to many users, we might do a lightweight internal pen-test or hire a third-party to audit the app. They would use the SBOM to focus on any known vulns in components and probe our application (this is more of a later stage activity).

- **Maintaining Dependencies:** We rely on Dependabot to keep libs updated. Outdated libraries can have vulnerabilities. So, frequent small updates are preferred over big jumps after a year. We also watch for announcements in the Next.js community or Prisma community about security releases.

- **Security Headers & Practices:** We ensure the deployed app has proper HTTPS (Vercel auto TLS) and uses HSTS. We set content security policy (especially important if we ever allow user-generated content or integrate external scripts). We also guard against common web app issues: CSRF (Next.js has built-in CSRF tokens for forms via NextAuth or similar if needed), XSS (React escapes content by default; any dangerouslySetInnerHTML usage must be reviewed).

- **SBOM Storage:** We can attach the SBOM file as an artifact in GitHub Actions for each release, so we have a historical record. This artifact, combined with git tags, lets us reproduce what was deployed. If a vulnerability is discovered later in a version, we can go back and see if that version was affected via its SBOM.

- **Tool Updates:** Keep the security tools themselves updated (e.g., CodeQL updated to latest queries, Snyk to latest DB). Also, consider adding **CodeQL custom queries** if we develop patterns (for instance, if we always use a certain sanitizer function, we could have CodeQL ensure it's used on certain inputs).

- **Continuous Improvement:** Security scanning isn't a one-time setup. We will regularly review the output of these tools. CodeQL might produce some false positives; we triage them, possibly adjust the queries. Snyk might alert on something not immediately fixable (e.g., an upstream library needs a fix); we then monitor or apply temporary mitigations. This becomes part of our backlog to address.

By using Snyk, CodeQL, gitleaks, and Trivy in our pipeline, we aim for **defense in depth**: - Snyk/Trivy cover known issues in dependencies and environment (SBOM helps here). - CodeQL covers custom code vulnerabilities. - Gitleaks covers secrets management issues. These complement our secure coding practices and reviews to create a secure product.

12. Backlog (Work Breakdown Structure of Tasks)

To implement this infrastructure and DevOps plan, we prepare a **work breakdown structure (WBS)** and backlog of tasks. This serves as a checklist for the team. The backlog is organized by major areas (jobs) and detailed steps:

1. Project Setup & Architecture Design - 1.1 **Architecture Diagram & Documentation** – *Create the high-level system diagram (mermaid/ASCII) and write narrative explaining components.* **(Done as part of this guide)**. - 1.2 **Monorepo Setup** – Initialize a Turborepo (if using) or single Next.js repo. Set up basic project structure (Next.js app, add design system package if separate, etc.). - 1.3 **Design System Integration** – Set up a shared UI library (if separate package) and configure Tailwind CSS with design tokens (OKLCH colors, etc.). - 1.4 **Module Scaffolding** – Generate scaffold pages/routes for main modules (Dashboard, Revenue, Expenses, etc.) and ensure navigation works (this is more application work but listed for completeness). - 1.5 **State Management & Offline Setup** – Choose approach for offline queue (maybe a service worker or IndexedDB). This is partially infra (capability for offline) – backlog item to implement offline persistence. - 1.6 **Keyboard Navigation Standards** – Draft guidelines or utility for keyboard shortcuts (again more app feature).

2. Environment & Infrastructure Provisioning - 2.1 **Vercel Project** – Create Vercel project, link GitHub repo. Define Production, Preview, Development envs. Add team members. - 2.2 **Custom Domain** – Configure custom domain for production (if applicable). - 2.3 **Provision Database (Dev)** – Set up local Postgres or a dev cloud DB. Possibly use Docker for local DB. - 2.4 **Provision Database (Prod)** – Create managed Postgres instance for prod (e.g., on Supabase/Neon). Configure credentials. - 2.5 **Provision Database (Staging)** – Create separate staging DB instance. - 2.6 **Set up Prisma** – Initialize Prisma in the project, configure the connection string env for dev. - 2.7 **Design Schema** – Design initial database schema for Phase 1 entities (accounts, transactions, budgets, etc.). Review for future multi-tenancy. - 2.8 **Run Initial Migration** – Execute `prisma migrate dev` to create initial migration SQL and apply to dev DB. - 2.9 **Provision Redis (Prod)** – Set up Upstash Redis (or chosen provider) for prod. Note the URL, configure env var. - 2.10 **Provision Redis (Staging/Dev)** – Perhaps a free Upstash instance or local Redis for dev. - 2.11 **Environment Variables** – Populate Vercel env vars for dev/staging/prod from `.env.example` needed values (dummy or real keys as appropriate).

3. CI/CD Pipeline - 3.1 **GitHub Actions: CI Workflow** – Write `.github/workflows/ci.yml` with steps: checkout, setup node, install, lint, test. Include caching for node modules. - 3.2 **Add CodeQL Action** – Use GitHub's auto-generation or manual config to add CodeQL analysis on push. - 3.3 **Add Snyk Action** – Configure Snyk test action, store SNYK_TOKEN in GitHub secrets. - 3.4 **Add gitleaks Action** – Integrate gitleaks action in CI. - 3.5 **Test CI on PRs** – Open a test PR to verify pipeline runs all steps and passes. - 3.6 **Set up branch protection** – Require CI checks to pass before merge on main branch. - 3.7 **CD Workflow** – (If using Actions to deploy) Write a deploy step or separate workflow that triggers on push to main: e.g., `vercel deploy` or use Vercel's GH integration (in which case, no action needed here aside from maybe an API call to trigger production deploy after staging approval). - 3.8 **Quality Gates** – Define thresholds (coverage %, etc.) and implement enforcement (maybe in package.json scripts or GH Actions steps). - 3.9 **Notification on CI Failures** – (Optional) Set up Slack/GitHub notifications when pipeline fails.

4. Runtime & Scaling Considerations - 4.1 **Evaluate Build Output** – Ensure Next.js output fits Vercel limits (if not, code-split or use dynamic imports to keep Lambdas under size limit). - 4.2 **Edge Function Evaluation** – Identify if any functionality (like auth middleware or geolocation) should use Edge Functions for performance. - 4.3 **Document Fly/Render Option** – Write an internal note on how to deploy on Fly if needed, including a Dockerfile for Next.js (just to have if contingency). - 4.4 **WebSocket/Realtime Plan** – If needed for live updates, research alternatives (Pusher, Supabase realtime, etc.) since Vercel can't hold socket long – backlog for Phase 2.

5. Background Jobs Implementation - 5.1 **Design Job System** – Decide on using Vercel Cron vs QStash for each job type. Document the approach. - 5.2 **Implement Cron Jobs** – Create endpoints under `pages/api/cron/*.ts` or the new `/app/api/cron/*.ts` routes for each scheduled task (e.g., `cron/dailySummary.ts`). In vercel.json, schedule them (e.g., daily, hourly as required). - 5.3 **Implement QStash Integration** – Set up Upstash QStash if using. Secure the endpoints (verify QStash signature as shown in docs). Write a small example job (like a test job that logs something after 1 min) to verify end-to-end. - 5.4 **Ingestion Job** – Code the job that fetches transactions from OpenFinance/Belvo. Possibly this is triggered by cron daily or by user action (link account then initial sync job). Ensure it uses env credentials and parses data into DB via Prisma. - 5.5 **Forecast Job** – Code a scheduled job for forecasting (maybe nightly update of projections). Could call an internal service or use AI API. - 5.6 **DQ Check Job** – Code a job that scans recent data for anomalies (e.g., balance mismatches). Decide schedule (maybe weekly). - 5.7 **Email Sending Job** – If using Gmail API to send or parse receipts, implement a job that checks for new receipts (via Gmail API) periodically, or triggers when user forwards an email. Ensure API limits are respected. - 5.8 **Job Idempotency** – Implement a mechanism in each job to skip if already done or handle overlapping runs. E.g., use a Redis lock or check a "lastRun" in DB. - 5.9 **Job Failure Handling** – Wrap job logic in try/catch, on error log to Sentry, maybe update a "job_status" table. If critical, maybe send an alert email to admin. - 5.10 **Metrics Collection in Jobs** – Instrument jobs to record how many items processed, duration, etc. Possibly push these to a logging system or simply log.

6. Database Enhancements - 6.1 **Write Initial Seed Script** – A script (Prisma or SQL) to populate lookup tables (like Categories, if any default categories). - 6.2 **Configure RLS for Future** – In staging or a separate branch, experiment with enabling RLS. Create a Postgres role for "app_user". Write a policy on a test table. Document steps to enable RLS in production when multi-tenant goes live. - 6.3 **Performance Indices** – Identify critical queries and ensure indexes exist. E.g., if frequently querying transactions by date and account, add index on (account_id, date). - 6.4 **Prisma Optimization** – Enable Prisma logging in dev to catch N+1 queries and refactor to use joins or `include` as needed. - 6.5 **Backup Verification** – Write a small runbook task: restore latest backup to dev and verify data integrity (perform at least once). - 6.6 **Set Up DB**

Monitoring – If using a provider dashboard, ensure we can see slow query logs. If not, enable pg_stat_statements and set up a way to review slow queries monthly.

7. Secrets & Config - 7.1 **Implement Zod Env Validation** – Create a `config.ts` that uses zod to parse env and export config object. Integrate it in app startup (so that missing env causes crash in non-prod to notice). - 7.2 **Vault for Secrets (optional)** – If decided, set up an alternative secrets store (like use Doppler for local dev syncing). - 7.3 **Rotation Policy Documentation** – Write an internal wiki on how/when to rotate keys (which keys every 90 days, which upon specific events, etc.). - 7.4 **Generate Strong Secrets** – Ensure JWT_SECRET, ENCRYPTION_KEY are strong (script to generate if needed). - 7.5 **Third-Party OAuth Setup** – For Gmail API, set up OAuth consent screen, get client ID/secret, save in env. Document the setup (since that's configuration). - 7.6 **Test that no secrets in logs** – Do a trial run with verbose logging to ensure we didn't accidentally log something sensitive (like logging full request which might include auth header – not good).

8. Monitoring & Alerts Setup - 8.1 **Sentry Project** – Create a Sentry project (one for front-end, one for backend or unified since it's Next.js). Install SDK in code (`npm install @sentry/nextjs`) and configure DSN. - 8.2 **Sentry DSN in env** – Add SENTRY_DSN to env. Initialize Sentry in `_app.tsx` or middleware. - 8.3 **Verify Sentry** – Induce a test error in dev or staging and confirm it appears in Sentry. - 8.4 **Configure Sentry Alerts** – Set up alert rules: e.g., email on every prod release error, Slack integration for new issue. - 8.5 **Set up Uptime Monitoring** – Use UptimeRobot (or similar) to ping the app's health endpoint (maybe we create `/api/health` that checks DB). Configure alert to team email on downtime > 5 minutes. - 8.6 **Performance Monitoring** – Decide on any APM. Perhaps use Vercel Analytics for web performance. Alternatively, integrate a simple Google Analytics or internal logging for page load times. - 8.7 **Create Grafana Dashboard (optional)** – If using a service like Grafana Cloud or Datadog, pipe metrics (from DB or custom) to it. Possibly skip now but leave in plan. - 8.8 **Cron Monitoring** – Sign up for Cronitor or Healthchecks and add a ping at end of critical cron jobs. - 8.9 **Log Aggregation** – Evaluate if Vercel's logging is sufficient. If not, set up Logflare (now part of Cloudflare) or another service. Backlog item to integrate if needed. - 8.10 **Trace Context** – Add a middleware that creates a request ID for each request (could use `x-request-id` header) and include it in logs, to correlate logs for a single operation.

9. Backup & DR - 9.1 **Enable PITR** – Turn on Point-in-Time Recovery on prod DB instance (if not default). Note the additional cost. - 9.2 **Document Restore Procedure** – Write step-by-step for restoring DB from backup (who to contact or what console steps). - 9.3 **Periodic Backup Test** – Schedule an issue every N months to do a backup restoration test in staging. - 9.4 **Implement Export Data** – Provide a way to export data (CSV of transactions, etc.) for user – doubles as a form of user-level backup. (This is more a feature but beneficial). - 9.5 **DR Simulation** – Simulate a failover: e.g., shutdown primary DB (if possible in test) and see if read replica can be promoted. Or simulate Vercel outage by running app locally connecting to DB backup. - 9.6 **Emergency Contacts** – List out contacts for each service (DB provider support, etc.) and ensure team knows how to reach them 24/7 if needed.

10. Security & Compliance - 10.1 **Dependency Updates (Dependabot)** – Enable Dependabot on the repo for npm dependencies and GitHub Actions. - 10.2 **Address CodeQL Findings** – After initial CodeQL run, fix any highlighted issue (or mark as safe with justification). - 10.3 **Address Snyk Findings** – Same for Snyk: update libraries or apply patches for any vuln found. - 10.4 **Run OWASP ZAP Scan** – Against staging, run a zap baseline scan. Analyze results (common missing headers etc.) and fix. - 10.5 **Implement Security Headers** – Add Helmet or configure Next.js to send CSP, HSTS, etc. (especially when custom headers needed). - 10.6 **SBOM Generation** – Add a CI job to generate SBOM (using `npm ls` or Trivy's CycloneDX

output) and upload it as artifact. - 10.7 **Penetration Test (external)** – (For later) Schedule a security audit when nearing multi-tenant launch. - 10.8 **Compliance Considerations** – If dealing with financial data, consider encryption at rest (the DB likely already does on SSD). Also, if any user PII, ensure GDPR compliance (allow data deletion, etc.). Add tasks for those as needed.

Each of these tasks can be tracked in our project management tool. As we implement, we'll adjust and add tasks (e.g., feature-specific infra tasks like setting up an AI API key, etc.). The above backlog ensures that all critical infrastructure and DevOps pieces are built and checked.

13. Acceptance Criteria and Service Levels; Runbooks for Incidents

To consider this infrastructure implementation successful, we define **Acceptance Criteria** in terms of Service Level Objectives (SLOs) and ensure we have runbooks to handle incidents:

Service Level Objectives (SLOs): - **Availability:** The production system should have an uptime of *99.9% or higher* per month (approx <= 45 minutes downtime). This implies robust hosting and quick recovery from incidents. - **Performance:** - Page load times (dashboard initial load) SLO: < *2 seconds* on a fast connection for the main dashboard view (excluding large data visualizations which load async). - API response time SLO: < *500ms* for standard API calls (p95 latency), and < *2s* even for heavier endpoints (p95). Any background processing beyond that should be offloaded. - **Data Freshness:** Bank and financial data should be no more than *24 hours out of date* (since last sync) for linked accounts – essentially an SLO that daily sync jobs run successfully. For critical accounts (like if user opens app, perhaps on-demand refresh happens). - **Data Consistency:** No lost updates – if a user makes a change, it's either applied or they are informed to retry (at most once). This is qualitative but important: essentially zero tolerance for silently losing user changes. - **Error Rate:** SLO that *error responses* (HTTP 5xx) are < 1% of requests. Ideally, none in normal operation. Similarly, client-side runtime errors should be rare; Sentry should not report uncaught exceptions frequently in prod. - **Job Success:** Background jobs (daily sync, etc.) succeed on their schedule > 99% of the time. If a job fails, it is retried and completed within the next schedule or sooner. - **Security:** No high-severity vulnerabilities open in production. This means our scanning and patching should ensure none of the known critical issues remain unresolved for more than e.g. 7 days. Also, no known data breaches – if one occurs, that's a failure beyond just SLO, it's an incident.

We will monitor these SLOs via the tools set up (availability via uptime checks, performance via monitoring and logs, error rate via Sentry or log analysis, etc.). If SLOs start slipping (e.g., response time creeping up), we'll take proactive action (scale up resources or optimize code).

Acceptance Testing: Before go-live, we'll run through scenarios: - Simulate a user doing key tasks with network throttling to ensure performance is acceptable. - Trigger failovers in a test environment to ensure the fallback mechanisms (e.g., a job retry) work. - Security tests as mentioned.

Runbooks (Incident Response Playbooks):

We prepare concise runbooks for various incident types. Each runbook outlines how to detect, diagnose, mitigate, and resolve the issue, including who to contact. Example runbooks:

- **Incident: Database Down or Unresponsive**
  Symptoms: App API requests are failing (500 errors), logs show connection errors to DB, or our external DB monitoring alerts us.
  Immediate Actions: Verify the database status on provider dashboard. If it's a known outage (cloud provider issue), communicate internally and to users (e.g., banner or status page if prolonged). If it's high load, consider scaling up or killing long queries.
  Mitigation: Failover to a read replica if available. If not, attempt a restart of the DB instance via provider. Use recent backup to spin up new instance if needed.
  Resolution: Once DB is back, verify data integrity. If any data lost (shouldn't if we restored PITR), determine if any user actions need to be redone. Write an incident report post-mortem.
  Follow-up: Tweak monitoring to alert sooner if DB CPU or connections spike (to catch issues before downtime).

- **Incident: Deployment Causing Outage**
  Symptoms: Right after a deployment, the app is not functioning (errors on all pages, or important feature broken). Sentry lit up with errors.
  Actions: Immediately use Vercel's **Rollback** to previous successful deployment (since we can do that within seconds on Vercel). Confirm recovery (site works again).
  Diagnosis: Gather logs/errors from the bad release, identify cause (maybe a migration that failed, or a bug). If a hotfix is straightforward, fix in code, run migrations if needed (or roll back migration via backup restore if catastrophic).
  Resolution: Deploy the hotfix or fix behind a feature flag to disable the offending part. Ensure all systems normal.
  Follow-up: Analyze why pipeline didn't catch it – add test for that scenario. If it was a migration issue, improve migration pre-check. Document the incident.

- **Incident: Background Job Failure**
  Example: The daily sync job has been failing for the past day (maybe an API changed).
  Detection: We notice via monitoring alert (job didn't report success) or by data being stale.
  Actions: Check logs for the job function (via Vercel logs or custom logs). If error is clear (e.g., auth failure to third-party), address that (maybe the API key expired – then rotate key). If it's a bug in our code, patch it and re-deploy the function.
  Mitigation: Run the job manually (maybe trigger via an API route or script) after fix to catch up on missed data. If partial data got in, ensure no duplicates.
  Resolution: Once job runs successfully, verify data currency.
  Follow-up: Add better error handling in job to avoid future fails (or alert sooner). Possibly add a circuit-breaker: if an API fails due to one bad data, skip that item but continue others.

- **Incident: Security Breach (suspected)**
  Symptoms: Unusual activity such as data discrepancy, or an external report that our API is leaking data. Or we find a leaked credential.
  Actions: This is critical – engage incident response team. Rotate any suspected compromised credentials immediately (e.g., if an API key leaked, revoke it). Potentially take the app offline if user data is at risk of being exfiltrated.

Investigation: Check logs for any unauthorized access. If a specific vulnerability (like an open endpoint) is identified, patch or disable it ASAP.
Communication: Inform stakeholders as required by law (if user data breach, etc., within 72 hours for GDPR for instance).
Resolution: Fix the vulnerability, restore services carefully. Possibly force logouts or password resets if needed.
Follow-up: Very thorough post-mortem. Implement additional security measures (e.g., WAF, more thorough code audits). Possibly engage external security audit to regain confidence.

- **Incident: Third-party API outage** (e.g., Stripe is down so payments can't be fetched)
  Symptoms: Errors from Stripe integration, our app may hang on payment-related pages.
  Actions: Feature toggle: we can temporarily disable that module or show a message "Stripe data is temporarily unavailable" to users rather than making them wait. The system overall remains up.
  Mitigation: If possible, use cached data (serve last known data from our DB) with a warning. Keep retrying in background.
  Resolution: Monitor the third-party status, once resolved, re-enable functionality (and perhaps trigger a fresh sync to reconcile any missed events).
  Follow-up: Possibly implement an automatic circuit breaker for that API in future – e.g., after X failures, don't call it for a while, just use cache.

For each runbook, we maintain a document accessible to the team. During an incident, the on-call (if we have formal on-call) or lead engineer will refer to these steps. We also maintain a contact list (who knows what part of system best, support contacts at providers, etc.).

Runbook: Deployment Process (not an incident, but operational runbook): - We will also have a runbook for how to deploy (in case the CI automation fails and we need to deploy manually, etc.), including how to run migrations manually if needed and how to verify a successful deployment.

Service Level Agreement (SLA): If we were offering this as SaaS to customers, we might promise some of these SLOs as SLAs (with penalties), but internally we treat SLOs as targets to maintain.

Acceptance Criteria Summary: We will consider this implementation complete when: - All items from the backlog that are critical for Phase 1 are done (CI passing, environment configured, monitoring in place, etc.). - We can deploy a change to production with zero downtime and automated tests verifying no regression. - We have demonstrated recovery from at least one type of failure in a test (e.g., simulated DB restore). - The system meets performance targets with test load (simulate a few concurrent users performing actions). - Documentation (this guide, .env.example, runbooks) is up-to-date and reviewed by the team.

14. Starter Configuration Files and Scripts

Finally, we provide starter configuration files and scripts to kick-start the implementation. These serve as templates:

GitHub Actions CI Pipeline (`.github/workflows/ci.yml`):

```yaml
name: CI
on:
  pull_request: [main]
  push: [main]
jobs:
  build_test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
          cache: npm
      - name: Install
        run: npm ci
      - name: Lint
        run: npm run lint
      - name: Type Check
        run: npm run type-check
      - name: Unit Tests
        run: npm run test:ci -- --coverage
      - name: Upload Coverage
        uses: actions/upload-artifact@v3
        with:
          name: coverage-report
          path: coverage
  security_scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Snyk Scan
        uses: snyk/actions/node@v1
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
        with:
          command: test
      - name: CodeQL Init
        uses: github/codeql-action/init@v3
        with:
          languages: javascript
      - name: CodeQL Analyze
        uses: github/codeql-action/analyze@v3
      - name: Secret Scan (Gitleaks)
        uses: zricethezav/gitleaks-action@v2
```

(The above defines two jobs: one for build/test, one for security scans. In practice, we might sequence them or run in parallel. We also would add deploy steps in another workflow or after these succeed, as described earlier.)

Environment Variables Example (`.env.example`):

```
# Application Environment
NODE_ENV=production
NEXT_PUBLIC_BASE_URL="https://myapp.com"

# Database (PostgreSQL) connection
DATABASE_URL="postgresql://user:password@dbhost:5432/appdb"
# Shadow DB for Prisma (for migrations)
SHADOW_DATABASE_URL="postgresql://user:password@dbhost:5432/appdb_shadow"

# Cache (Redis) connection
REDIS_URL="redis://default:password@redishost:6379"

# Third-Party API Credentials
OPENFINANCE_API_KEY="test_openfinance_key_123"
BELVO_CLIENT_ID="belvo_client_id_xyz"
BELVO_SECRET="belvo_secret_abc"
PLUGGY_API_KEY="pluggy_key_xyz"
STRIPE_SECRET_KEY="sk_test_xxx"
STRIPE_WEBHOOK_SECRET="whsec_xxx"
MERCADOPAGO_ACCESS_TOKEN="TEST-xxx"

# Google API (for Gmail integration)
GOOGLE_OAUTH_CLIENT_ID="your-google-client-id.apps.googleusercontent.com"
GOOGLE_OAUTH_CLIENT_SECRET="your-google-oauth-secret"

# AI API (if using OpenAI or others for intelligence layer)
OPENAI_API_KEY="sk-xxxx"

# Application Secrets
JWT_SECRET="super-secret-jwt-signing-key"
NEXTAUTH_SECRET="another-secret-for-nextauth"  # if using NextAuth
ENCRYPTION_KEY="base64-encoded-32-bytes"       # for encrypting sensitive data
at rest

# Misc Configs
SENTRY_DSN="https://examplePublicKey@o0.ingest.sentry.io/0"
```

(This file should be kept in sync with actual needed variables; developers fill in their own values or fetch from secret manager. In production, actual values are set in Vercel or secret store.)

Makefile for Common Tasks (Makefile):

```
# Makefile to help with common devops tasks

.PHONY: dev build migrate deploy lint test format db-seed

# Start local development server
dev:
    @echo "Starting Next.js in development mode..."
    next dev

# Build the application for production
build:
    next build

# Run database migrations (ensure DATABASE_URL is set)
migrate:
    npx prisma migrate deploy

# Deploy to Vercel (requires Vercel CLI and auth)
deploy:
    vercel --prod

# Lint and format the code
lint:
    next lint

format:
    prettier --write .

# Run tests
test:
    jest

# Seed the database with initial data
db-seed:
    node scripts/seed.js

# Generate Prisma client (in case of schema change without migration)
prisma-generate:
    npx prisma generate
```

(This Makefile assumes `next`, `vercel`*, etc., are installed. It provides shortcuts. On Windows, developers might use npm scripts instead, but this is handy for Unix-like environments.)*

Turbo Repo config (turbo.json) – if using Turborepo for multiple packages (app, design system, etc.):

```
{
  "$schema": "https://turbo.build/schema.json",
  "pipeline": {
    "build": {
      "outputs": [".next/**", "dist/**"]
    },
    "lint": {},
    "test": {}
  }
}
```

(This is a minimal example; Turborepo can be configured to cache outputs and run tasks in parallel. In a monorepo, we'd tag which packages to run tasks in, etc.)

Scripts (e.g., seed.js for database seeding):

```
// scripts/seed.js
import prisma from "../prisma/client";  // import configured Prisma client

async function seed() {
  // Example: create default categories
  const defaultCategories = ["Income", "Rent", "Groceries", "Utilities"];
  for (const name of defaultCategories) {
    await prisma.category.upsert({
      where: { name },
      update: {},
      create: { name, type: name === "Income" ? "INCOME" : "EXPENSE" }
    });
  }
  console.log("Seeded default categories.");
}

seed().catch(e => {
  console.error("Seeding failed: ", e);
  process.exit(1);
}).finally(async () => {
  await prisma.$disconnect();
});
```

(This script would be called via `npm run seed` *or* `make db-seed`*. It ensures some initial data is present. Upsert is used to avoid duplicates on re-run.)*

vercel.json (to configure Cron Jobs and other Vercel settings):

```
{
  "version": 2,
  "env": {
    "DATABASE_URL": "@database_url",  // references to Vercel project env vars
    "REDIS_URL": "@redis_url",
    "SENTRY_DSN": "@sentry_dsn",
    // ... etc for all needed env
  },
  "functions": {
    "api/cron/*.js": {
      "memory": 512,
      "maxDuration": 60
    }
  },
  "crons": [
    { "path": "/api/cron/daily-sync", "schedule": "0 2 * * *" },
    { "path": "/api/cron/refresh-rates", "schedule": "*/30 * * * *" }
  ]
}
```

(The env section is for Vercel to link secrets; functions config sets memory/time for cron functions if needed. The crons array schedules two jobs: one daily at 2am, one every 30 minutes as example.)

These starter files should be adapted to the project's specific details, but provide a baseline for implementing the described infrastructure. With this foundation, the team can begin coding and know that the DevOps scaffolding will support a smooth development and deployment lifecycle.

---

1 Ephemeral environments - Neon Branching
https://neon.com/branching/ephemeral-environments

2 3 About code scanning with CodeQL - GitHub Docs
https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql

4 Snyk Vs Trivy Comparison | Aikido Security
https://www.aikido.dev/blog/snyk-vs-trivy

5 7 18 Can you use Vercel for backend? What works and when to use something else | Blog — Northflank
https://northflank.com/blog/vercel-backend-limitations

6 Vercel Functions Limits
https://vercel.com/docs/functions/limitations

8 9 Cron Jobs
https://vercel.com/docs/cron-jobs

10 11 Next.js - Upstash Documentation
https://upstash.com/docs/qstash/quickstarts/vercel-nextjs

12 13 14 Multi-tenant data isolation with PostgreSQL Row Level Security | AWS Database Blog
https://aws.amazon.com/blogs/database/multi-tenant-data-isolation-with-postgresql-row-level-security/

15 Validate Environment Variables With Zod - Catalin's Tech
https://catalins.tech/validate-environment-variables-with-zod/

16 17 Database backups | Supabase Features
https://supabase.com/features/database-backups