FH University of Applied Sciences TECHNIKUM WIEN

Secret Notes
Project Documentation

Semester Project | Continuous Integration

Alejo Flores Lucey (ID: if24x390)
Hongseok Choi (ID: if24x387)

https://github.com/alejofl/secret-notes

FH University of Applied Sciences TECHNIKUM WIEN

Table of contents

FH University of Applied Sciences TECHNIKUM WIEN

Project Overview

Secret Notes is a secure note-taking web application that implements end-to-end encryption for storing sensitive information. The application demonstrates modern DevOps practices including containerization, CI/CD pipelines, automated testing, and blue/green deployment strategies.

The key features of the application are:

- Symmetric Encryption: AES-256-GCM with PBKDF2 key derivation
- Modern UI: React 19 with TypeScript and Tailwind CSS
- Secure Backend: Fastify API with PostgreSQL database
- Markdown Support: Full GitHub Flavored Markdown rendering
- Analytics Integration: PostHog for feature toggles and A/B testing
- Comprehensive Testing: Unit, E2E, and performance tests

Applications and Docker Files

The application consists of two main projects: one NodeJS project for the backend application, featuring a Fastify REST API, and a React project for the frontend website.

The choice to use React to create the webapp is because of its component-based architecture, which promotes modularity and reusability, making development more efficient and maintainable. React's virtual DOM enhances performance by updating only the parts of the UI that change, resulting in a smoother user experience. Additionally, its large community, rich ecosystem of tools and libraries, and strong support for modern JavaScript practices make it a reliable and scalable choice for building dynamic, responsive interfaces.

Furthermore, the React application is developed using Tanstack Router and Tanstack Query. TanStack Router is a fully type-safe routing library for React that provides a powerful and flexible way to manage navigation and layout in web applications. It offers nested routing, route-level data loading, and strong TypeScript integration, making it ideal for scalable and maintainable codebases. Unlike traditional routers, it emphasizes composability and fine-grained control, allowing for a more organized and predictable structure. On the other hand, TanStack Query is a powerful data-fetching and caching library that simplifies asynchronous state management by handling queries, caching, background updates, and more, all with minimal boilerplate. It improves performance and user experience by reducing redundant network requests and keeping the UI in sync with server data. Together, these tools provide a modern, efficient approach to building robust, responsive web applications.

Lastly, it was decided to use Shadcn/ui as the component library for the webapp. The package is a modern component library for React that gives developers full control and flexibility by allowing them to copy customizable, production-ready, components directly into their projects. Built on top of Radix UI for accessibility and styled with Tailwind CSS, it offers a strong foundation for building consistent, accessible, and visually appealing UIs. Unlike traditional libraries that ship as black-box packages, Shadcn promotes code ownership, letting you freely adapt components to your design system without unnecessary dependencies. It's especially well-suited for developers who value customization, clean architecture, and maintainability.

For the backend, respectively, the REST API was developed entirely using Fastify and TypeScript, mainly because it was a requirement of the assignment. These two frameworks are a powerful combination for building high-performance, type-safe backend applications. Fastify is a lightweight and extremely fast Node.js web framework optimized for low overhead and high throughput, making it ideal for scalable APIs. When paired with TypeScript, it provides strong static typing, improving code reliability, catching bugs early, and enhancing developer productivity through better tooling and autocompletion. Fastify also has built-in TypeScript support, making it easy to define and enforce request/response schemas, which leads to more predictable and maintainable code. This stack is particularly effective for building modern, robust APIs with confidence and speed.

Dockerized Frontend

Using Docker images to build and deliver the frontend offers consistency, portability, and ease of deployment across different environments. By containerizing the frontend application, we ensure that it runs with the same dependencies, configurations, and environment, whether in development, staging, or production, eliminating "it works on my machine" issues. Docker also simplifies CI/CD

workflows by allowing us to define the build process in a Dockerfile, making builds reproducible and version-controlled. Additionally, delivering the frontend as a Docker image makes scaling and deployment easier in cloud-native infrastructures like Kubernetes, and can integrate seamlessly with Nginx or other static file servers for efficient delivery.

As it was part of the assignment, the frontend application is served using a multi-stage Dockerfile, in which we first build the static files for the frontend using Node as the base image and then serve these static files using Nginx.

Nginx, as it is a HTTP web server, exposes the files under the port 80 by default. We make use of Docker Compose and environment variables to bind this port to one chosen by the developer in the host machine. For this, we use the FRONTEND_PORT_BINDING environment variable.

It is worth mentioning that all environment variables used in the frontend application must be present at build time, as the files generated are static. Because of this, a few workarounds were needed to make it work on both local machines and CI servers. We took advantage of the build arguments one can send to Docker when building images, to make the following environment variables available at build time: VITE_API_BASE_URL, VITE_PUBLIC_POSTHOG_KEY and VITE_PUBLIC_POSTHOG_HOST.

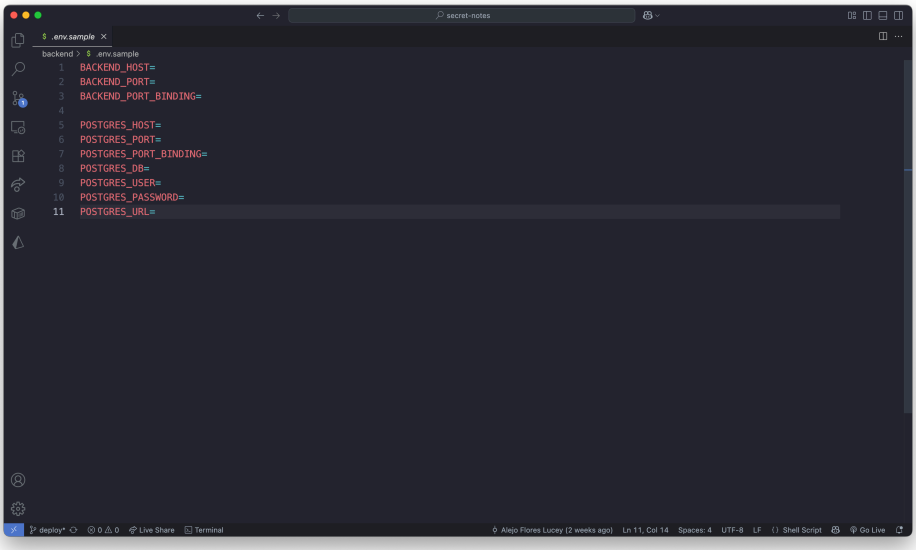
One drawback of static files being served via AWS Academy is that every time that the AWS Console is started, a new set of IPs and DNS names are assigned to the running instances. This causes all frontend images generated previously to become stale, as now the API base URL will be different.

Dockerized Backend

As well as the frontend, the backend is also containerized and delivered using Docker images. However, the process is different, as the Fastify server is not static and can execute code at runtime. This made the handling of environment variables much easier.

The backend application expects a file called .env with all the environment variables defined, from the connection with the database, to the port in which to expose the server.

In the following image we can see the list of environment variables needed to start the backend application.



Version Control

For version control, we decided to structure the project as a monorepo using npm workspaces, which allows us to manage multiple packages (such as frontend, backend, and shared utilities) within a single repository. This setup simplifies dependency management, ensures consistency across components, and improves collaboration by centralizing the codebase.

We used Git as our version control system and GitHub as our remote repository and collaboration platform. Git allowed us to track changes, manage branches, and work simultaneously on different features without conflicts. GitHub provided a centralized space for code reviews, pull requests, and issue tracking, fostering better communication and coordination among team members. Features like GitHub Actions enabled us to automate parts of our workflow, such as testing and deployment. Together, Git and GitHub ensured a smooth, transparent, and collaborative development process throughout the project.

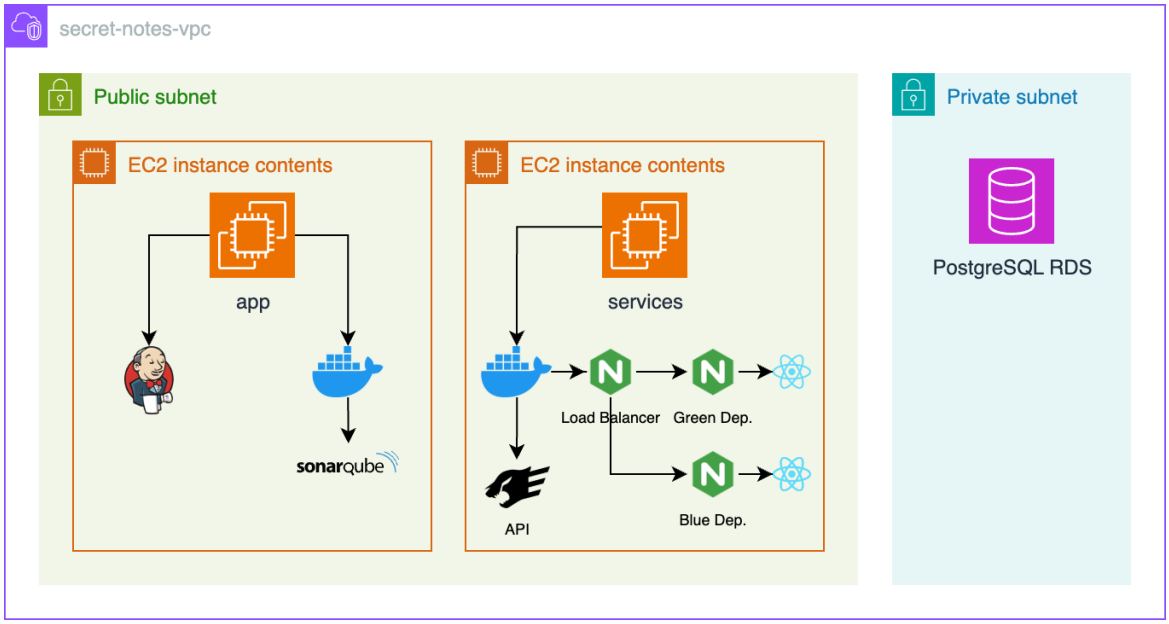
We followed the Conventional Commits specification to keep commit messages clear, standardized, and machine-readable. This practice enables better automation (e.g., changelog generation, semantic versioning) and makes the history easier to understand and navigate.

Our branching strategy revolves around three main concepts:

- The master branch holds the latest stable version of the code and serves as the base for feature development.
- Developers create feature branches off master, work on them independently, and then merge them back once the feature is complete and reviewed.
- A dedicated deploy branch is used for production releases. When we are ready to deploy, we rebase deploy from master, triggering an automatic deployment to production. This ensures that only explicitly promoted and tested code reaches the live environment.

This approach gives us a clear, controlled, and scalable version control workflow.

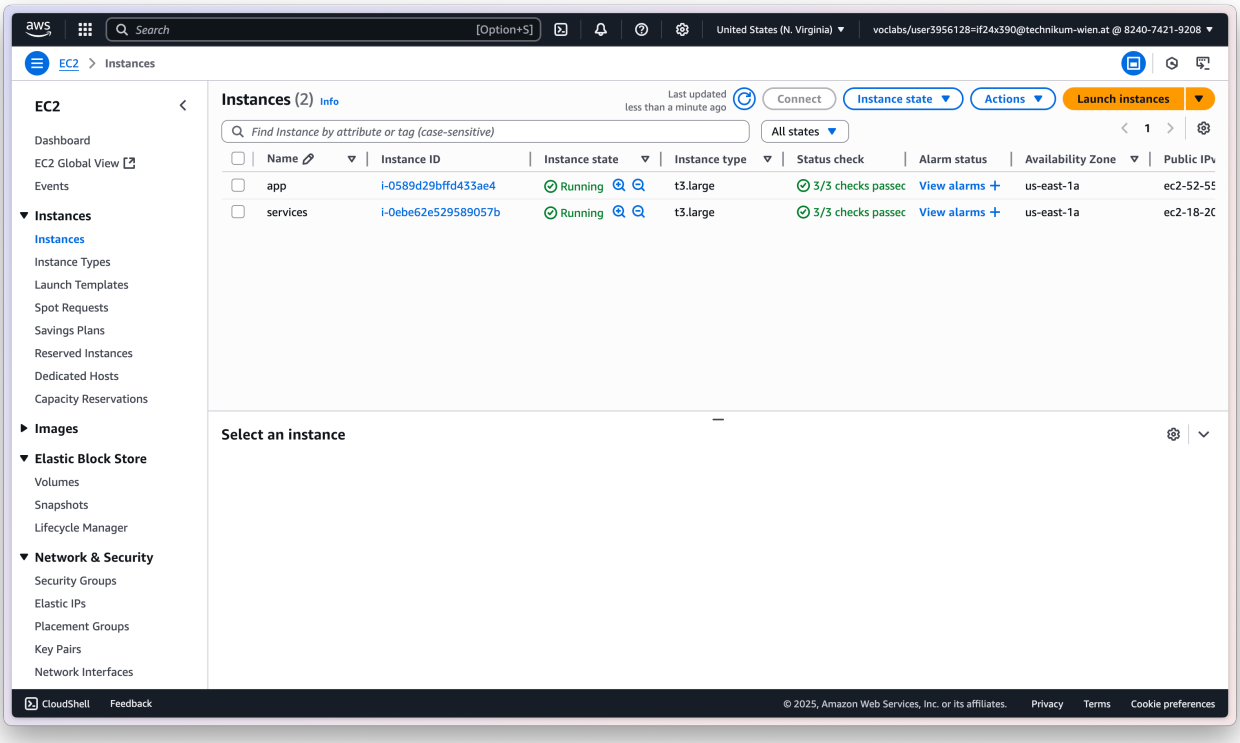
Infrastructure



For the infrastructure, we leveraged AWS Academy, which provided us with a managed environment to experiment and deploy real-world cloud solutions. Due to AWS Academy restrictions, all resources were provisioned in the us-east-1 region, specifically within the us-east-1a availability zone.

We designed a custom VPC named secret-notes-vpc with two subnets: one public and one private, to separate internet-facing services from internal infrastructure. Within the public subnet, we deployed two EC2 instances running Ubuntu 24.04. The first instance, named services, hosts our Jenkins CI/CD pipeline and SonarQube for static code analysis. The second instance, called app, is responsible for running both the backend and frontend of the application, making it accessible to users.

To securely store and manage application data, we provisioned a PostgreSQL-compatible RDS instance, which resides in the private subnet. This design ensures that our database is shielded from public access, while still being accessible by the app instance via internal networking. Overall, this infrastructure setup provided a practical and secure environment for deploying and managing the project while aligning with best practices and AWS Academy limitations.



The EC2 instances provisioning was challenging at best, because of the resources needed by the platforms we are using, like Jenkins and multiple Docker containers running simultaneously. Because of this, we decided to provision said instances with the following details:

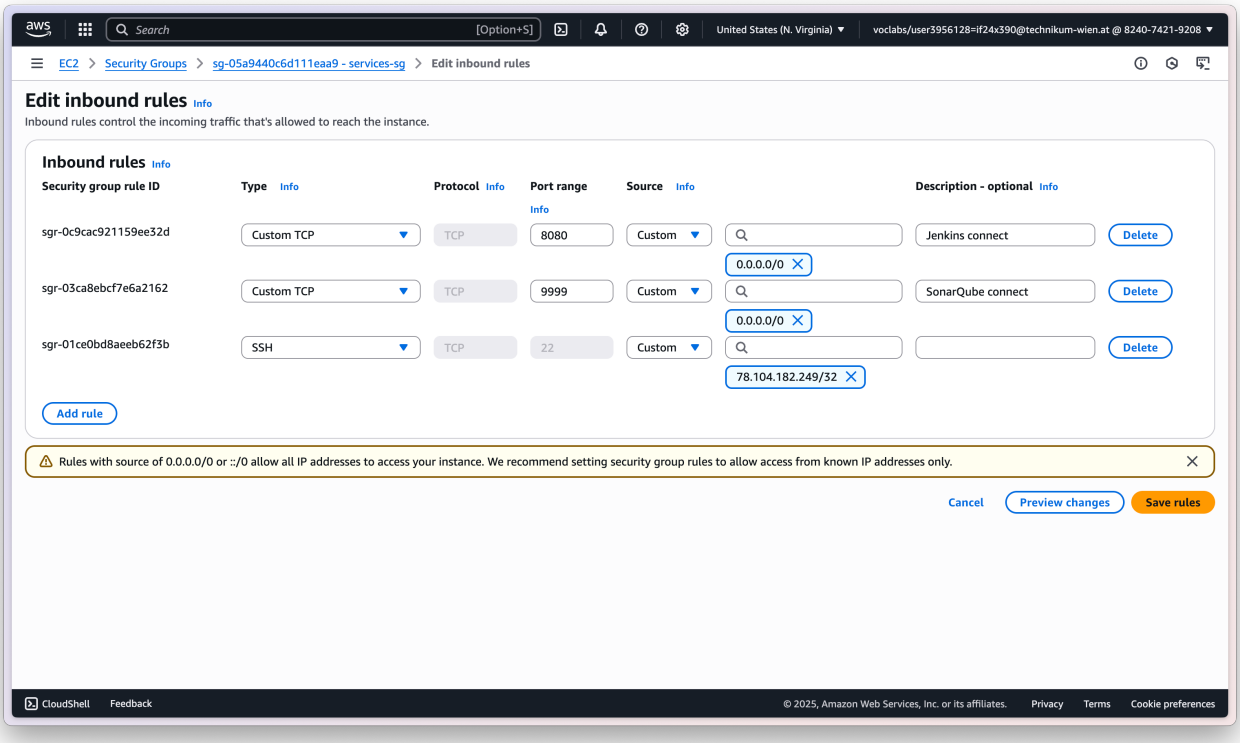
- Instance type: t3.large
- vCPUs: 2 virtual CPUs
- Memory: 8 GiB of RAM
- Architecture: x86_64 (Intel/AMD-based)
- Storage: 32 GiB using EBS

The database, on the other hand, is the default provisionable PostgreSQL-compatible RDS within AWS Academy.

To secure our infrastructure, we configured security groups to control traffic flow to and from each AWS resource. Security groups act as virtual firewalls, allowing us to define which types of network traffic are permitted based on protocol, port, and IP range. For this project, we created specific rules to limit access: the services and app EC2 instances allowed inbound SSH and HTTP traffic only from specific IPs or ports, depending on our needs. Meanwhile, the RDS database in the private subnet was configured to only accept connections from the app instance within the VPC, blocking all external access. This layered security approach helped protect our services while ensuring the necessary communication between components within the infrastructure.

We created three security groups: services-sg, app-sg and database-sg. The inbound rules can be seen in the following images.

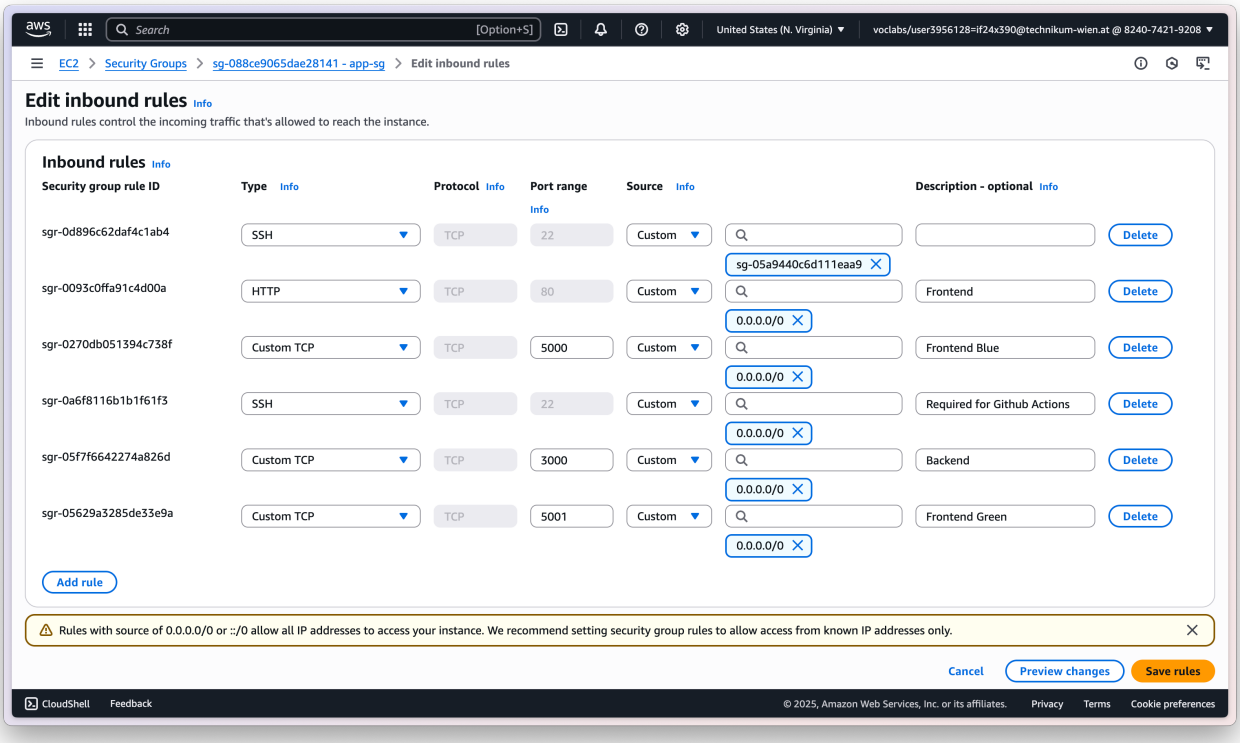
For the services-sg security group, we allow HTTP connections to the port 8080 for Jenkins and 9999 for SonarQube. Additionally, we allow SSH connection through the default port from a specific IP address, to prevent unauthorized access.



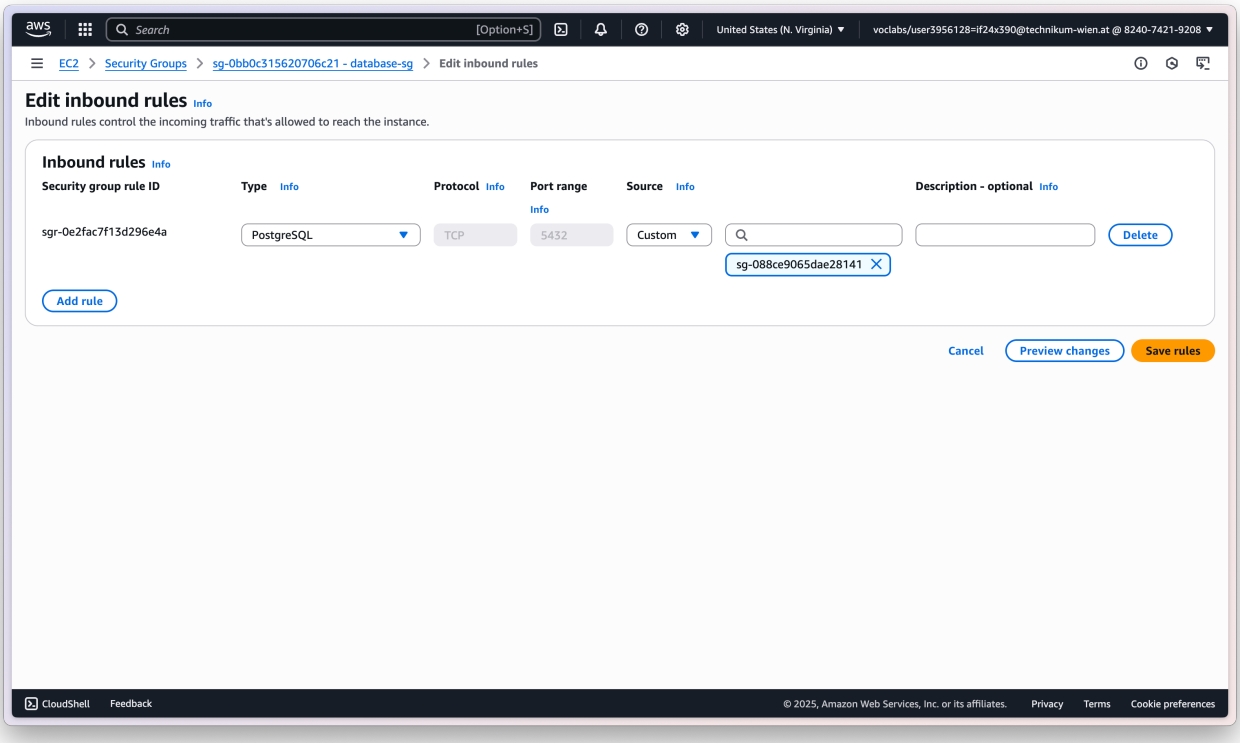
For the app-sg security group, we allow HTTP connections to the following ports:

- 80: deployed webapp
- 5000: blue deployment of the webapp. This is needed to run end-to-end and performance tests.
- 5001: green deployment of the webapp. This is needed to run end-to-end and performance tests.
- 3000: REST API access

Also, we needed to allow SSH access to anyone because of the CI/CD pipeline running in GitHub Actions. As this pipeline is run from GitHub servers, we cannot pinpoint the specific IP address that will be used to handle the deployment.



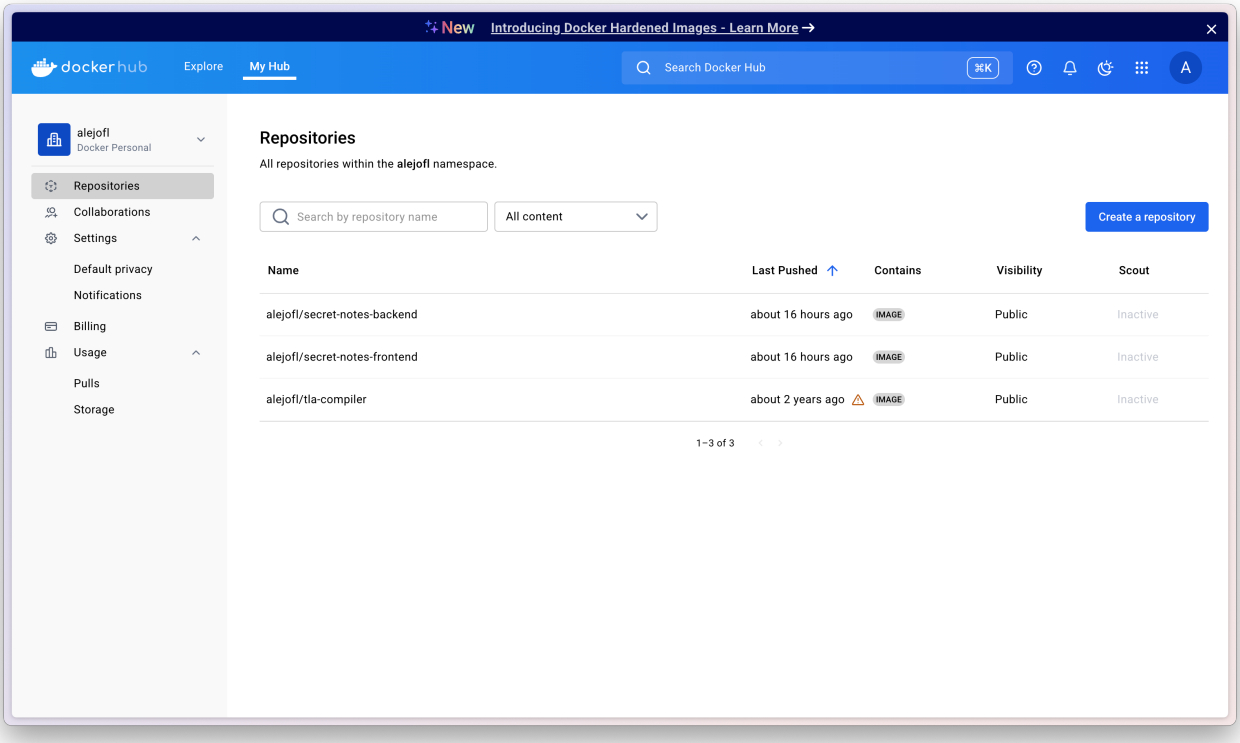
Lastly, the database-sg security group is much simpler. It just allows connections via the port 5432 to any instance that has the app-sg attached. This allows only the app EC2 instance to connect to the PostgreSQL database.



Docker Hub

To deliver our application images, we used Docker Hub as the image registry, leveraging its integration with our CI/CD pipeline for seamless deployment. We used the default Docker Hub repository under the alejofl account and followed a clear naming convention for our images: {user}/{application}-{service}. This resulted in two main images: alejofl/secret-notes-backend and alejofl/secret-notes-frontend, corresponding to each part of our application.

For versioning, we adopted semantic versioning (semver) aligned with the project's development stages. During the CI process, the current version is automatically extracted from the package.json file of each workspace. Each image is then built and tagged with this version and with the latest tag to mark the most recent stable build. This setup ensures consistency between code and deployment versions, simplifies rollback procedures, and allows for easy identification and use of specific application states directly from Docker Hub.



FH University of Applied Sciences TECHNIKUM WIEN

CI Servers

Self-Hosted

For our self-hosted CI server, we chose Jenkins, which is running on the services EC2 instance. We set up a Multibranch Pipeline project that automatically detects branches and executes the appropriate pipeline based on the presence of a Jenkinsfile. This file is located in the services/jenkins directory of our repository and defines the steps for building, testing, analyzing, and deploying the application.

To trigger the pipeline automatically on code changes, we configured a webhook in GitHub that fires on each push event. This webhook notifies Jenkins, which then scans the repository for changes and runs the corresponding pipeline for the updated branch.

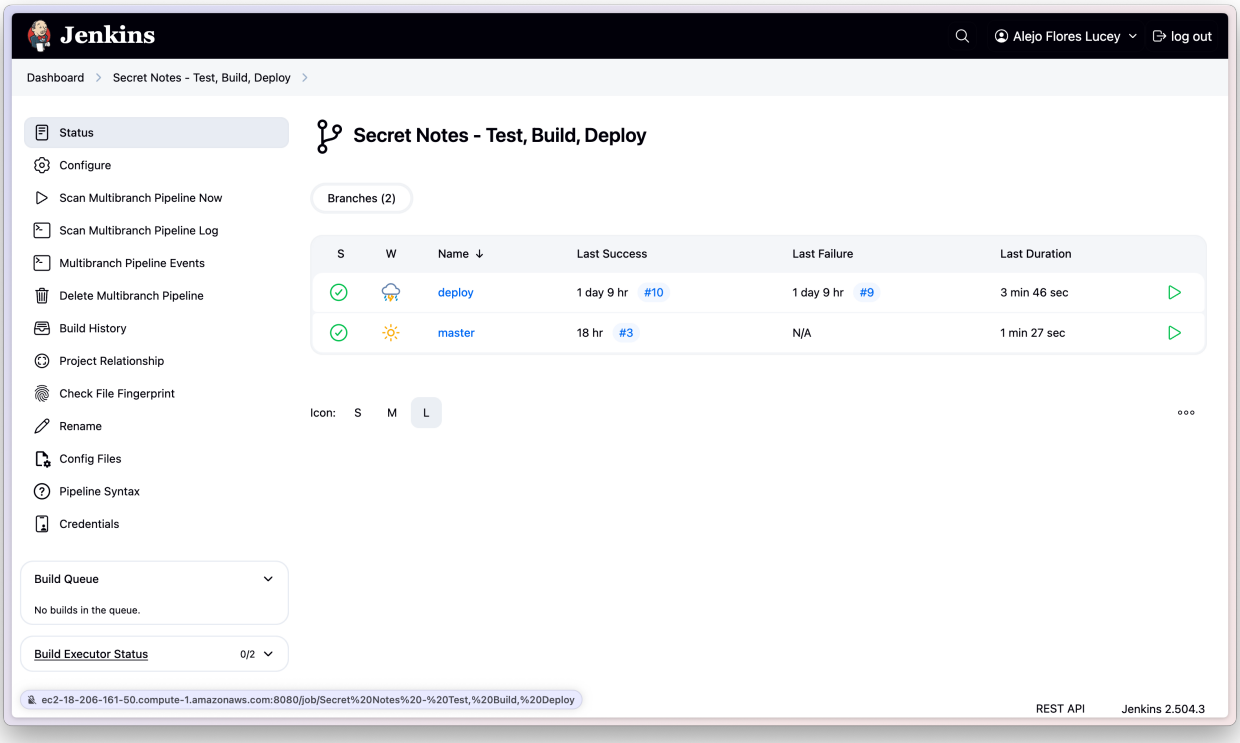
To fully configure the server and meet all assignment requirements, we installed and set up several essential Jenkins plugins, including:

- SonarQube Scanner: for static code analysis and quality checks
- Snyk Security: for dependency vulnerability scanning
- Docker Pipeline: to build and manage Docker images as part of the pipeline
- NodeJS: to manage Node.js environments during the build
- SSH Pipeline Steps: to enable remote command execution over SSH
- Multibranch Scan Webhook Trigger: to enable webhook-based triggering of multibranch projects

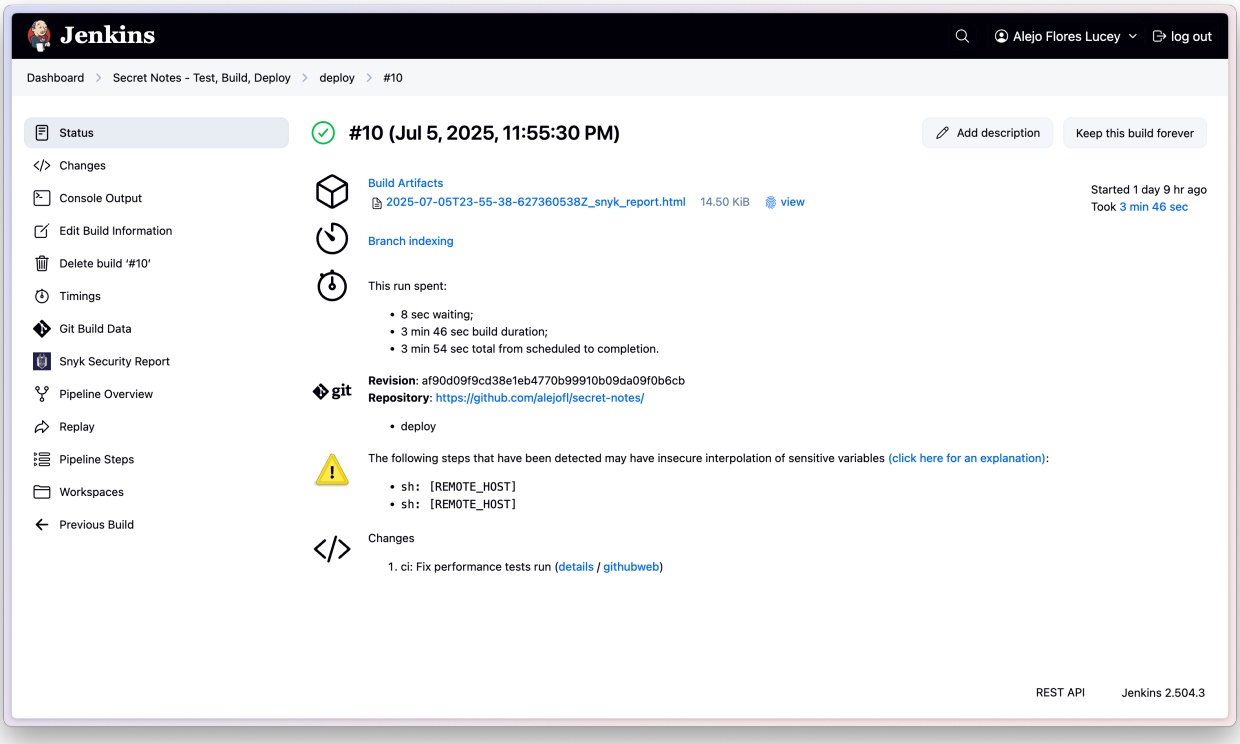
This setup provided a robust and automated CI/CD process, ensuring code quality, security, and deployment readiness throughout the development lifecycle.

One caveat of our implementation resides on the fact that we need to update several environment variables each time we start our AWS Academy infrastructure, due to its public IPs changing on every startup. This was taken into account during our implementation of the CI/CD pipeline, and we use the credentials feature of Jenkins to handle it.

In the following image we can see the Jenkins dashboard, with all of the information of our pipelines:

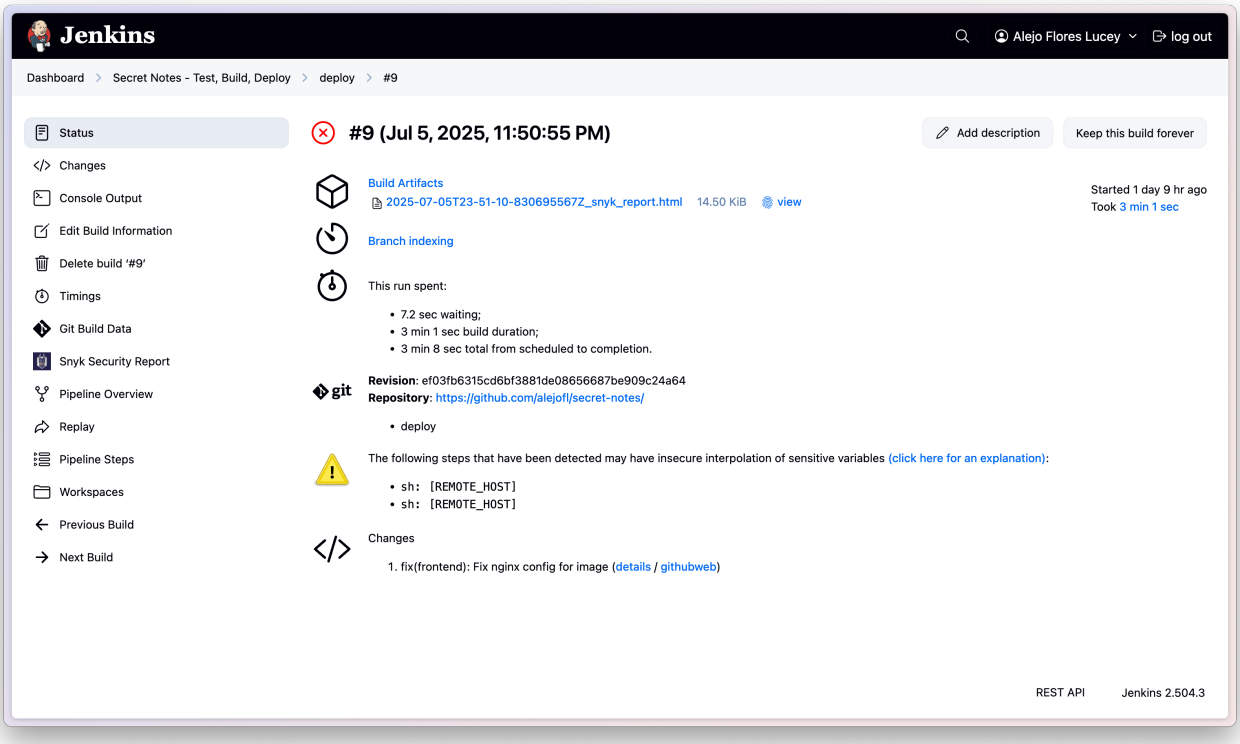


Then, in the next image, we can see a successful run of the deploy pipeline. After this success, a new version of the application can be accessed. A full log of the pipeline can be seen in our repository, under the docs directory.



Lastly, in the next image, we can see a failed run of the deploy pipeline. In this case, the failure was caused by the performance tests run with Grafana K6. Because of this fail, the deploy was not

finished and the previous version of the application can still be accessed without problems. A full log of the pipeline can be seen in our repository, under the docs directory.



Cloud-Hosted

In addition to our self-hosted Jenkins setup, we implemented a cloud-hosted CI/CD pipeline using GitHub Actions to ensure continuous integration and automated delivery in a scalable, serverless environment. This pipeline is defined in a single workflow file and is triggered on pushes to the master or deploy branches.

The workflow is structured in multiple jobs which do the same as the self-hosted alternative. The jobs handle preparation, quality analysis, testing, building, delivery, and deployment. In the preparation stage, the version is extracted from the package.json file, allowing us to tag Docker images consistently with the application's current semver version.

We make use of several pre-built actions from the GitHub Marketplace, including:

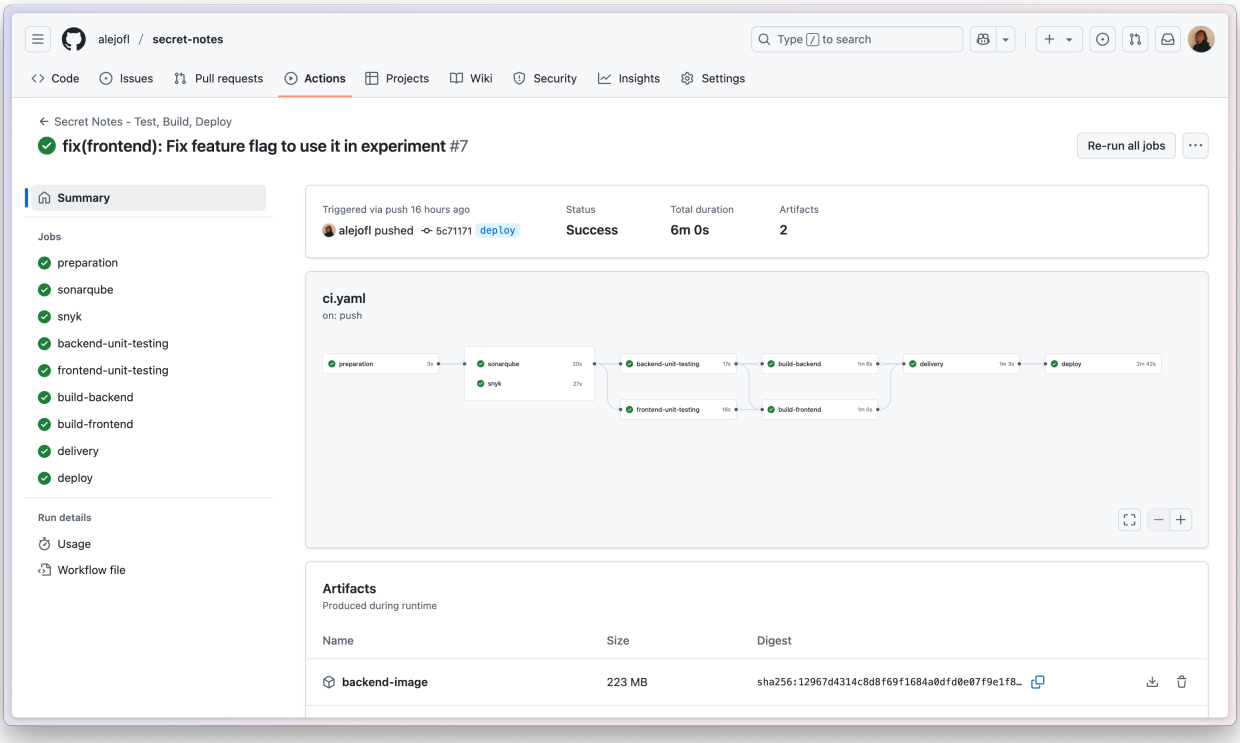
- actions/checkout to clone the repository,
- docker/build-push-action for building and pushing Docker images,
- sonarsource/sonarqube-scan-action for static code analysis,
- snyk/actions/node for security vulnerability scanning,
- and webfactory/ssh-agent for secure SSH access during deployment.

Unit tests for both the frontend and backend are executed using npm test, and successful builds are packaged into Docker images. These are uploaded as artifacts and then pushed to Docker Hub under the naming convention alejofl/secret-notes-{service}, tagged both with latest and the extracted version.

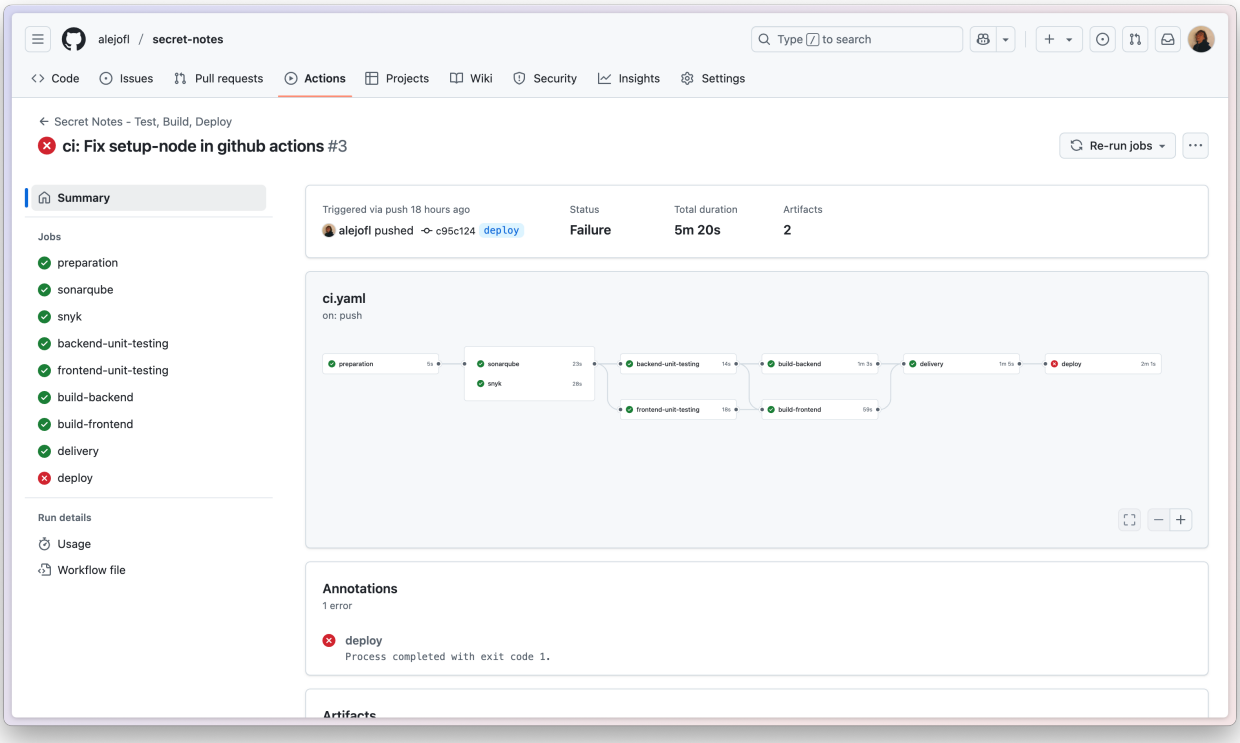
Finally, in the deploy job, a blue-green deployment strategy is performed via SSH to our remote host. This includes running end-to-end tests and performance tests on the new version before switching traffic over to it. By combining GitHub Actions with Docker and our deployment scripts, we achieved a robust and fully automated CI/CD pipeline hosted entirely in the cloud.

As well as with the Jenkins pipeline, we need to handle the change of IPs and DNS names every time we start our AWS Academy infrastructure. To do this in GitHub Actions, we use the repository secrets.

In the following image we can see a successful run of the GitHub actions pipeline:



And, as well as with Jenkins, here we can see a failed run of the pipeline. In this case, the pipeline failed because of the end-to-end tests. Because of this, the deploy was aborted and the faulty version of the app never reached production.



Complete logs of the runs can be found under the docs directory.

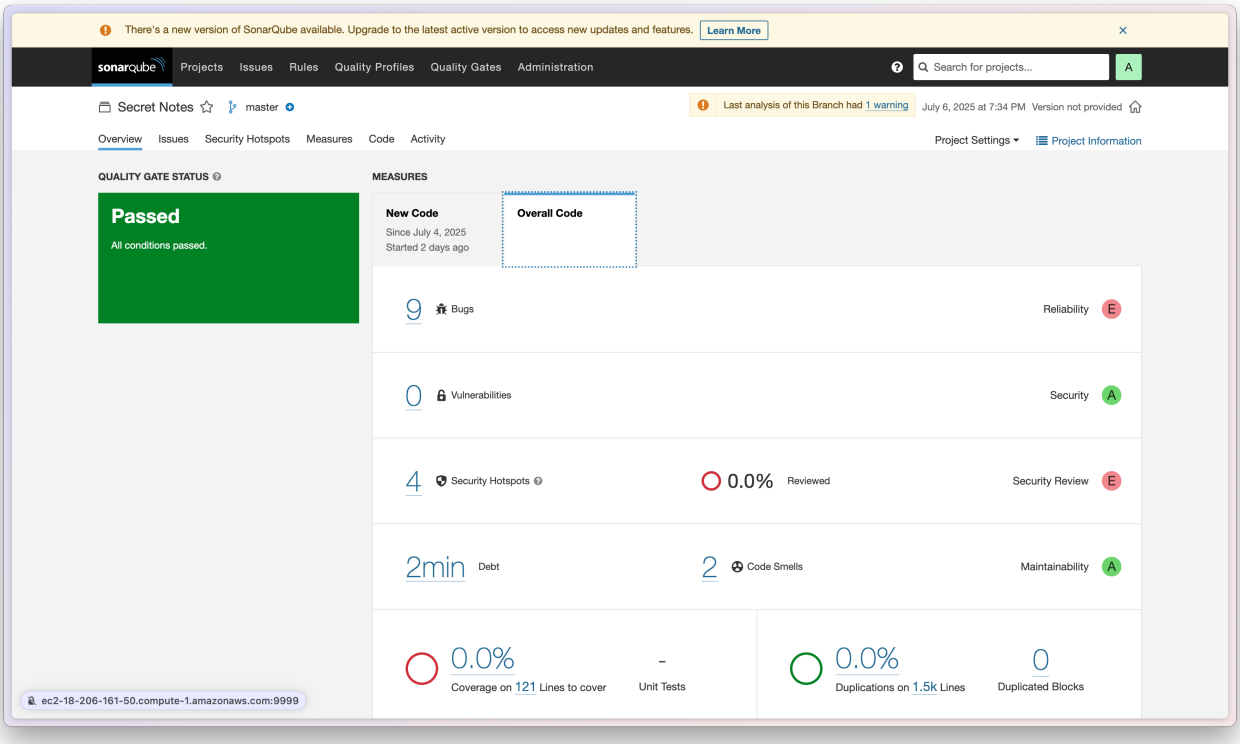
Code Quality Server

To ensure code quality and maintainability across our project, we integrated SonarQube as our static analysis tool. We use the Community Edition of SonarQube, which is deployed in its containerized version using Docker Compose on the services EC2 instance. This setup allows us to run the service locally within our infrastructure while keeping it isolated and easy to manage.

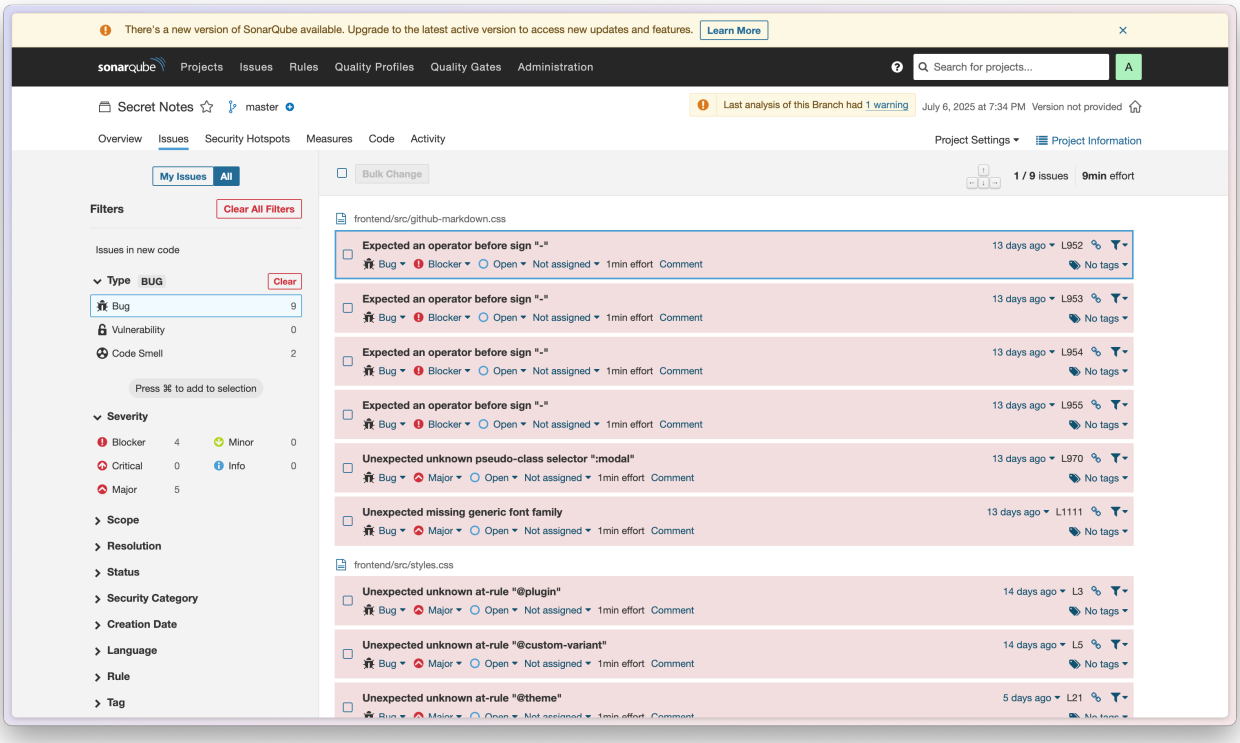
For local analysis during our Jenkins pipeline, we installed the SonarQube Scanner plugin, which connects directly to the running SonarQube service on the same instance. This enables Jenkins to automatically run code quality scans as part of the pipeline, ensuring that every change is evaluated for bugs, code smells, and other issues.

In the cloud-hosted pipeline with GitHub Actions, we use the official sonarsource/sonarqube-scan-action from the GitHub Marketplace. This action connects to our self-hosted SonarQube instance over the internet using credentials stored in GitHub Secrets. This way, both our self-hosted and cloud-based CI environments are aligned in terms of code quality enforcement, helping us maintain a consistent and reliable codebase throughout development.

In the following image, we can see a typical output of our analysis in SonarQube:



As we can see there, SonarQube is detecting some bugs. However, if we take a closer look, we can see that it is a false positive, as it is not understanding that some CSS files are filled with Tailwind syntax.


There's a new version of SonarQube available. Upgrade to the latest active version to access new updates and features.
Learn More
sonarqube
Projects
Issues
Rules
Quality Profiles
Quality Gates
Administration
Search for projects...
A
Secret Notes
master
Last analysis of this Branch had 1 warning
July 6, 2025 at 7:34 PM
Version not provided
Overview
Issues
Security Hotspots
Measures
Code
Activity
Project Settings
Project Information
My Issues
All
Bulk Change
1 / 9 issues
9min effort
Filters
Clear All Filters
Issues in new code
Type
BUG
Clear
Bug 9
Vulnerability 0
Code Smell 2
Press ⌘ to add to selection
Severity
Blocker 4
Minor 0
Critical 0
Info 0
Major 5
Scope
Resolution
Status
Security Category
Creation Date
Language
Rule
Tag
frontend/src/github-markdown.css
Expected an operator before sign "-"
13 days ago L952
Bug Blocker Open Not assigned 1min effort Comment
No tags
Expected an operator before sign "-"
13 days ago L953
Bug Blocker Open Not assigned 1min effort Comment
No tags
Expected an operator before sign "-"
13 days ago L954
Bug Blocker Open Not assigned 1min effort Comment
No tags
Expected an operator before sign "-"
13 days ago L955
Bug Blocker Open Not assigned 1min effort Comment
No tags
Unexpected unknown pseudo-class selector ":modal"
13 days ago L970
Bug Major Open Not assigned 1min effort Comment
No tags
Unexpected missing generic font family
13 days ago L1111
Bug Major Open Not assigned 1min effort Comment
No tags
frontend/src/styles.css
Unexpected unknown at-rule "@plugin"
14 days ago L3
Bug Major Open Not assigned 1min effort Comment
No tags
Unexpected unknown at-rule "@custom-variant"
14 days ago L5
Bug Major Open Not assigned 1min effort Comment
No tags
Unexpected unknown at-rule "@theme"
5 days ago L21

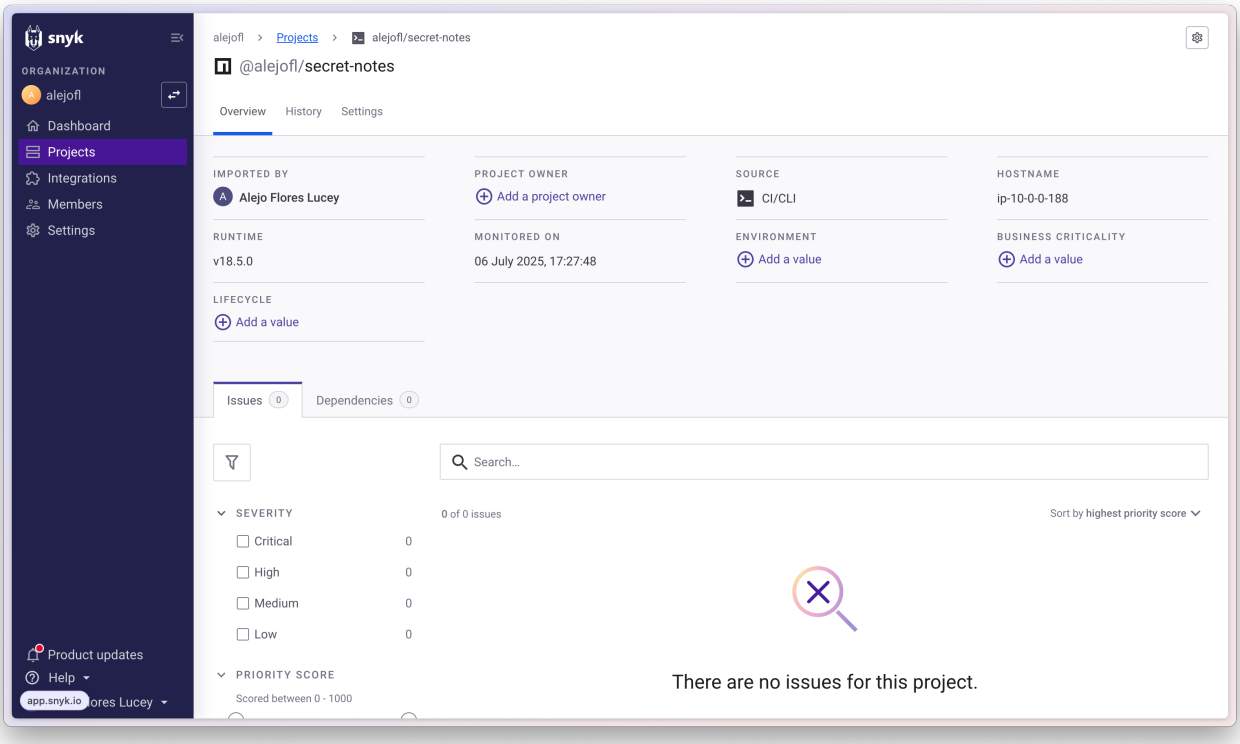
Security Scan Server

For security scanning, we integrated Snyk, a powerful tool that identifies vulnerabilities in application dependencies. Snyk connects to its cloud-hosted service using an API token, which we securely store and access through environment variables or CI secrets.

In both our Jenkins pipeline and GitHub Actions workflow, we run Snyk scans during the early stages of the pipeline to ensure that any known vulnerabilities are detected before proceeding to build and deployment. By leveraging the snyk/actions/node GitHub Action and the Snyk Security plugin for Jenkins, we automatically check our frontend and backend Node.js dependencies for security risks.

This setup helps us enforce security best practices and maintain a safer application by preventing vulnerable packages from being promoted to production.

As we can see in the following image, there are no issues found on our project:



To fix vulnerabilities identified by Snyk, if we had any, the first step would be to review the detailed report provided by the scan, which highlights the affected packages, severity levels, and recommended remediation steps. In many cases, vulnerabilities can be resolved by simply upgrading to a more secure version of the affected dependency, often suggested directly by Snyk.

Feature Toggle & A/B Test Server

To implement feature toggling and A/B testing in our application, we integrated PostHog, a robust open-source analytics and experimentation platform. PostHog allows us to dynamically manage features in production without requiring redeployments, making it an ideal tool for iterating on user experience and interface changes. We use the PostHog React SDK, which provides built-in support for feature flags and experiments directly in the frontend codebase.

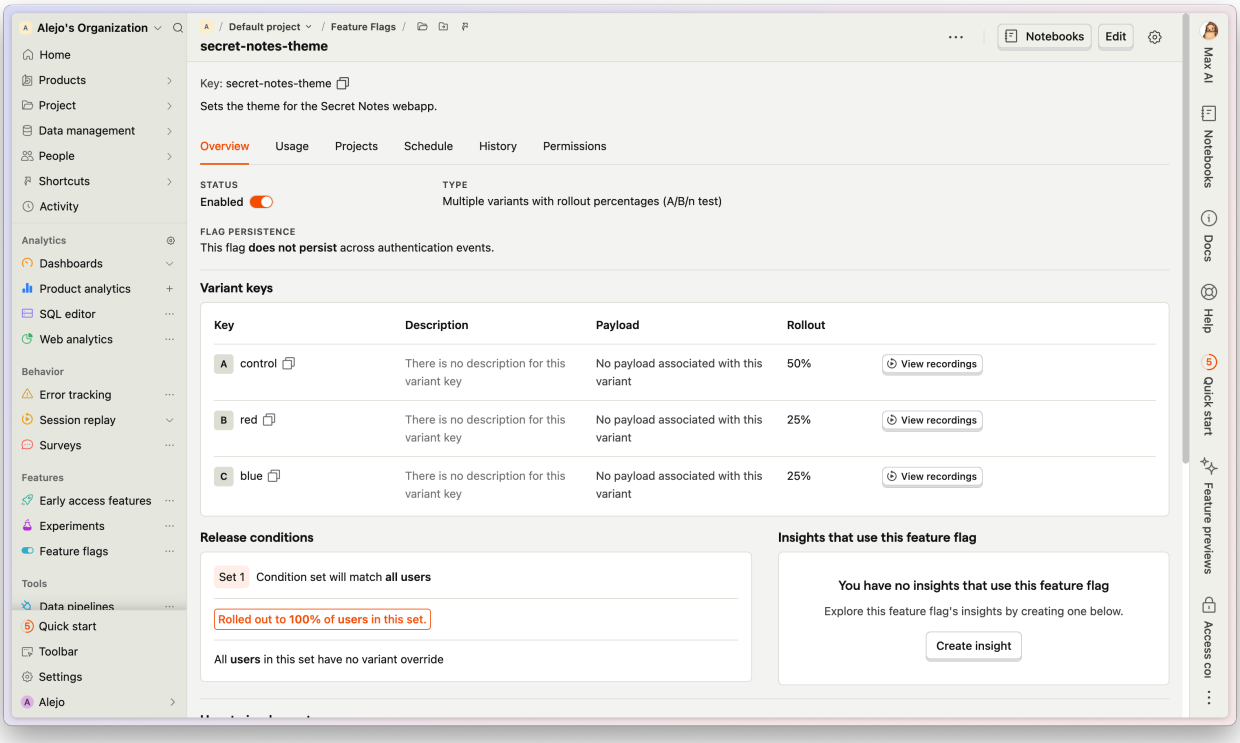
In our application, we use PostHog to control the theme selection feature through a feature flag. We created three visual variants of the interface: the default black theme, a red theme, and a blue theme. PostHog's experimentation feature is then used to define an A/B test, where users are automatically and randomly assigned to one of the three themes. This enables us to observe how different user segments interact with the application under each visual setting.

All user interactions and behaviors, such as session time, navigation paths, or feature usage, are tracked through PostHog’s analytics dashboard. This data allows us to compare engagement metrics across theme variants, providing valuable insights into user preferences and the potential impact of design decisions.

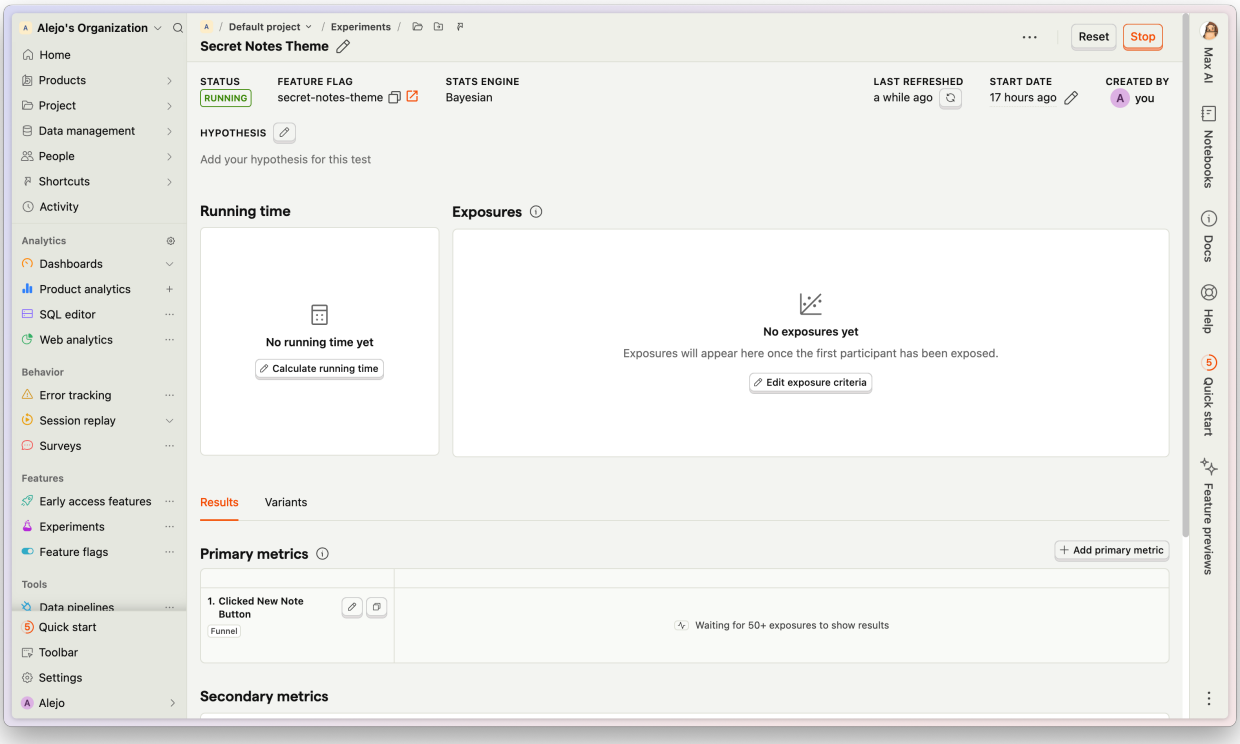
By using PostHog in this way, we not only enable gradual rollouts and safe experimentation, but also empower the team to make data-driven product decisions. This setup ensures that new features and changes can be tested in production with real users, helping us improve the application incrementally and with confidence.

In our case, we tracked the experiment success with the clicks to the “New Note…” button as the main metric.

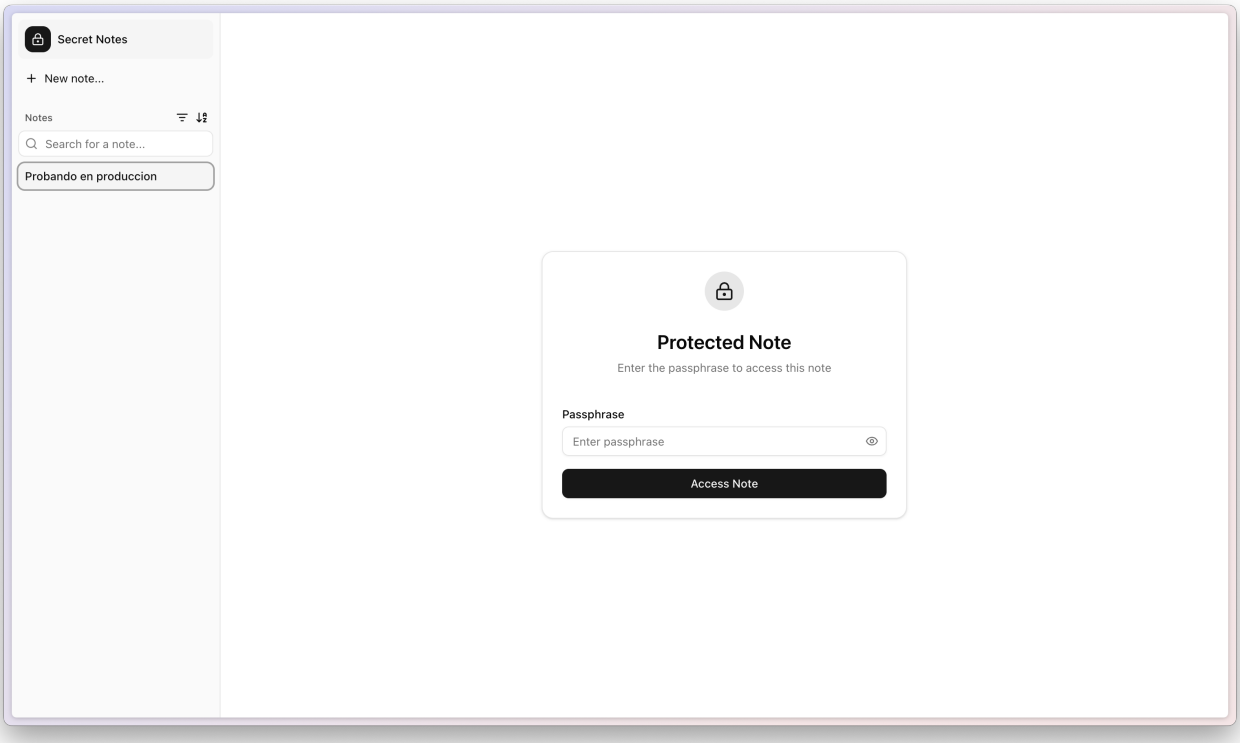
In the following image, we can see the configuration for the feature flag:

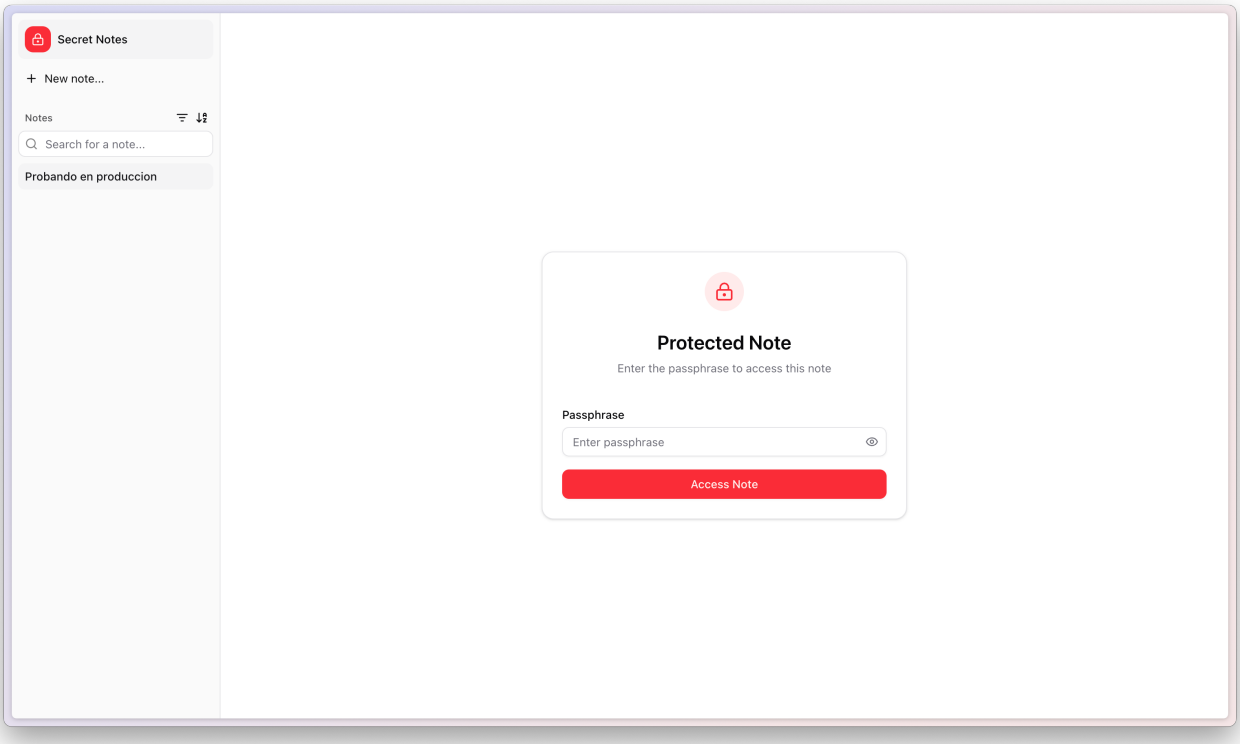


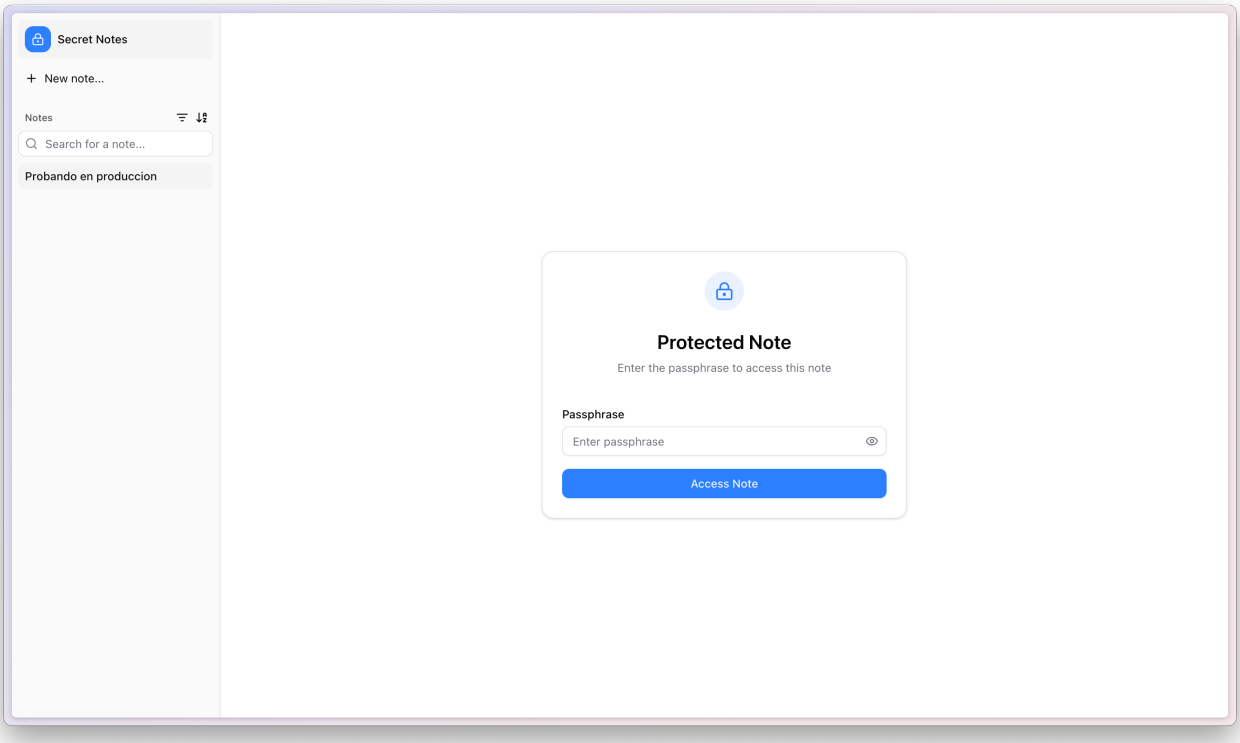
Also, in the next image, we can see how the A/B test is configured. As we do not have real users, we do not see any results yet, but we can see that everything is configured to work once we have enough data:



Lasly, in the following three screenshots, we can see the application with the variants defined:


Secret Notes
New note...
Notes
Search for a note...
Probando en produccion
Protected Note
Enter the passphrase to access this note
Passphrase
Enter passphrase
Access Note


Secret Notes
New note...
Notes
Search for a note...
Probando en produccion
Protected Note
Enter the passphrase to access this note
Passphrase
Enter passphrase
Access Note


Secret Notes
New note...
Notes
Search for a note...
Probando en produccion
Protected Note
Enter the passphrase to access this note
Passphrase
Enter passphrase
Access Note

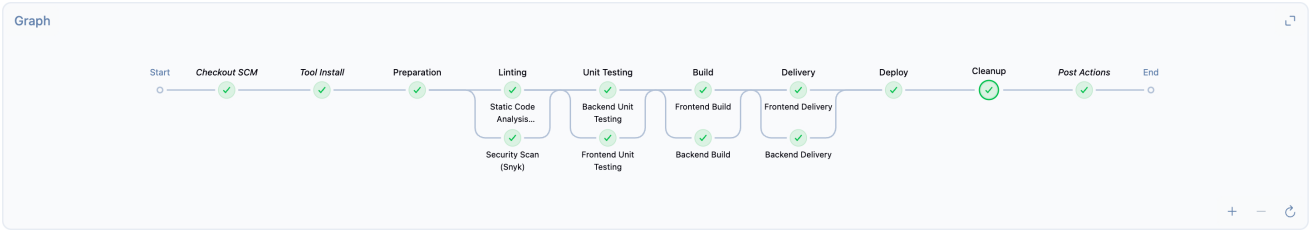
Pipeline Stages

Our CI/CD process is designed with a series of well-defined stages. This process is implemented in both Jenkins and GitHub Actions, with slight variations in how each platform handles certain steps. We divided the pipeline in seven stages:

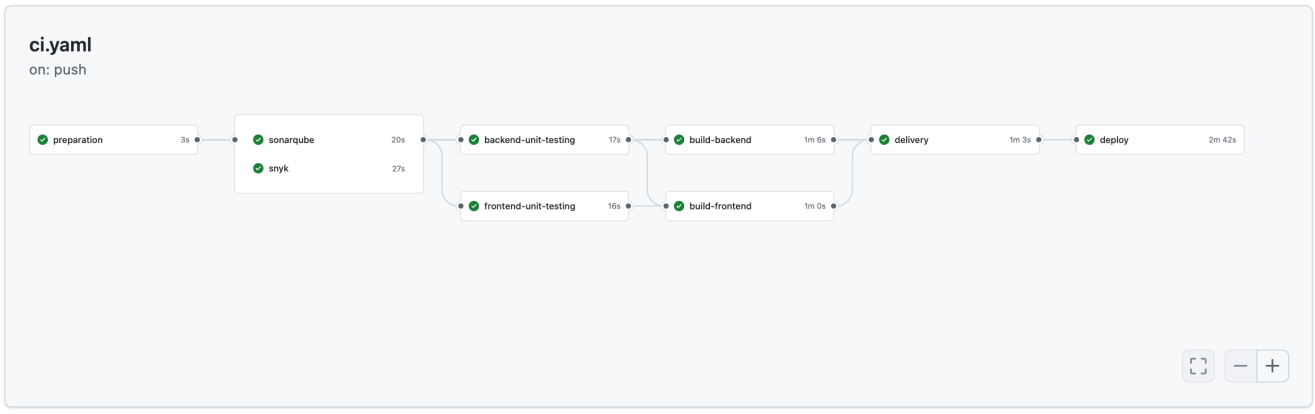
- Preparation
- Linting
- Unit Testing
- Build
- Delivery
- Deploy
- Cleanup

Depending on the server (self-hosted or cloud-hosted), there are more stages, that run inside these stages defined here. For example, the build stage can be divided into the backend build and the frontend build.

In the following image, we can see a diagram generated by Jenkins with the stages:



Likewise, in the following image, we can see another diagram generated by GitHub Actions with the stages:



Below is a breakdown of each stage and the differences between the two systems.

Preparation

The pipeline begins by checking out the latest version of the code from the repository and determining the application's version. This version is extracted from the package.json file and used to tag Docker images throughout the build and delivery process.

In Jenkins, the code is checked out using checkout scm, and a shell script is used to extract the version, which is then stored in a global pipeline variable.

In GitHub Actions, the same version is extracted and stored using job outputs, making it accessible to other dependent jobs.

While both approaches achieve the same goal, Jenkins relies on internal variables, whereas GitHub Actions uses a more declarative and modular output-sharing mechanism.

Linting

This stage focuses on code quality and security. We run two tools in parallel: SonarQube for static code analysis, and Snyk for identifying security vulnerabilities.

With Jenkins, SonarQube is executed via the installed scanner and environment setup. Snyk is run through a dedicated plugin using an API token stored in Jenkins credentials.

In GitHub Actions, the SonarQube scan is performed using the official sonarsource/sonarqube-scan-action, and Snyk runs via the snyk/actions/node action. Both tools connect to their respective remote services using securely stored GitHub Secrets.

The key distinction is how the integrations are managed. Jenkins leverages locally installed tools and plugins, while GitHub Actions uses hosted integrations through reusable actions.

Unit Testing

At this stage, we verify that the application logic behaves as expected by running unit tests for both the frontend and backend components.

Jenkins installs dependencies and runs tests in two parallel steps using shell commands for each part of the codebase.

GitHub Actions also performs this in parallel jobs, using setup-node to prepare the environment and executing test scripts from the respective directories.

The processes are functionally similar in both systems, although GitHub's use of discrete jobs improves separation and caching flexibility.

Build

The build stage compiles the application into deployable Docker images. This includes injecting environment-specific build variables such as API URLs and analytics keys.

In Jenkins, the docker.build command is used within pipeline scripts to create the frontend and backend images directly.

GitHub Actions uses the docker/build-push-action from Docker to build the images. These images are then exported as tar files and uploaded as artifacts for later stages.

Jenkins builds and retains Docker images in memory, while GitHub Actions uses a file-based handoff, which is better suited to stateless, cloud-hosted runners.

Delivery

In this stage, the Docker images built in the previous step are pushed to Docker Hub.

Jenkins handles Docker login and pushes images directly using the Docker registry credentials configured in the Jenkins credential store.

GitHub Actions downloads the previously built images, logs into Docker Hub using GitHub Secrets, and pushes both the latest and version-tagged images.

Both setups follow a similar logic, but GitHub Actions relies on previously uploaded artifacts to perform the push.

Deploy

The deployment process uses a blue-green deployment strategy, ensuring minimal downtime and safe rollbacks. The deployment is performed on a remote EC2 server.

Jenkins connects to the remote server via SSH, triggers a deployment script (start-blue-green.sh), runs end-to-end and performance tests using the exposed test environment, and then switches the live deployment using switch-blue-green.sh.

GitHub Actions performs the same steps using the ssh-agent and direct ssh commands. The tests are run against the same test endpoint, with URLs constructed dynamically based on the deployment output.

While the deployment logic is shared, the difference lies in credential handling and orchestration. Jenkins uses internal credential binding, whereas GitHub Actions uses injected secrets and a sequence of well-defined steps.

Cleanup

Once the deployment is complete, we clean up unused Docker images to free up disk space on the build agents.

In Jenkins, this is done using the docker image prune command within a post-deploy step.

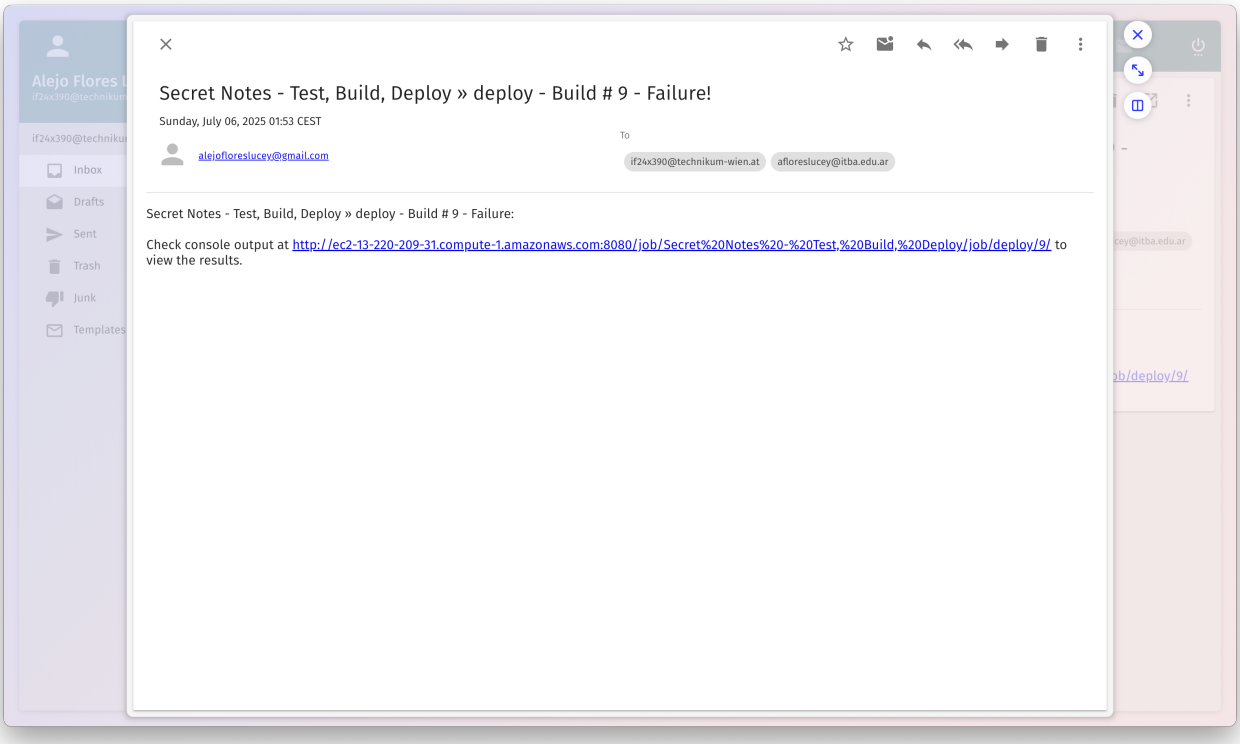
GitHub Actions does not currently perform this cleanup as the build environment is ephemeral and reset on every run.

This final stage ensures the Jenkins environment remains lean and avoids buildup from multiple deployments.

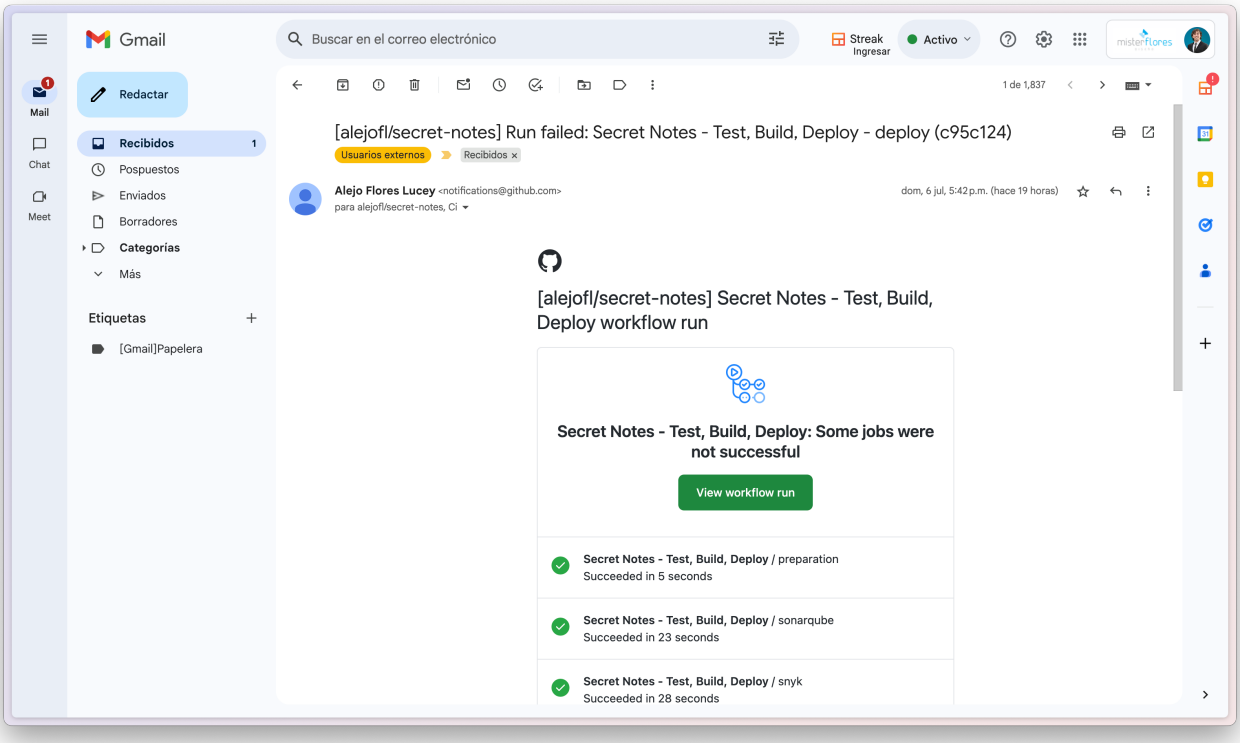
In summary, both Jenkins and GitHub Actions support a robust and automated CI/CD process with the same logical stages. Jenkins provides tighter control through its scripting capabilities and centralized environment, while GitHub Actions offers modularity, ease of use, and tighter integration with GitHub's ecosystem. The implementation across both systems ensures consistency, quality, and reliability in our software delivery pipeline.

If any of the pipelines failed, then the developer should be notified. We handle this using emails.

For Jenkins, the specific steps to send de email must be configured in a post-actions stage. In the following image, we can see the email received:



With GitHub Actions, on the other hand, the email is sent automatically, without any configuration needed on our side. In the following image, we can see the email received:



Blue/Green Deployment

To achieve zero-downtime deployments and enable quick rollbacks in case of failure, we use a Blue/Green Deployment strategy for the frontend application. This strategy ensures that at any given moment, only one version of the application (either Blue or Green) is actively serving users through a shared NGINX reverse proxy. The inactive version is updated and tested in isolation, and only if all checks pass, it replaces the live version.

Our setup includes:

- Two frontend services: frontend_blue and frontend_green
- One reverse proxy service: proxy, running NGINX
- One backend service: backend

All services are managed by a shared docker compose stack (compose.app.yaml) and DNS resolution between services is handled internally by Docker Compose using service names.

To actively make a deploy with blue/green, first we run the start-blue-green.sh script. This script initiates the deployment by determining which frontend service is currently active, then deploying the updated version to the inactive one. Here's how it works:

1. Service Tracking:
    a. The currently active service (either frontend_blue or frontend_green) is tracked via a file named running_service.
    b. If the file does not exist (e.g., first deploy), it defaults to using frontend_blue as the inactive one.
2. Backend Update:
    a. The backend service is always updated before deploying the new frontend version.
    b. It is pulled, stopped, and relaunched to ensure the latest code is running.
3. Deploy Inactive Frontend:
    a. The script pulls and starts the inactive frontend service using Docker Compose.
    b. This service runs on a different internal port (5000 for Blue, 5001 for Green).
    c. The script outputs the port of the newly started (inactive) service so it can be used for end-to-end and performance testing.

This design ensures that no updates touch the currently live frontend, maintaining uninterrupted user access.

After start-blue-green completes, automated end-to-end and performance tests are executed against the newly deployed (inactive) frontend instance. These tests validate that the updated version behaves correctly and performs within acceptable limits. The inactive instance is accessible via its known internal port. If the tests pass, the system proceeds to switch traffic. If the tests fail, no changes are made to the live deployment.

Once the new version has been validated, the switch-blue-green.sh script performs the following:

1. Determine the Inactive (now tested) Service:
    a. Based on the running_service file, it determines which frontend service is currently inactive.
2. Switch Proxy Configuration:
    a. The NGINX proxy configuration is updated by copying the corresponding config file (e.g., frontend_blue.conf) to proxy.conf.

   b. A hot reload of NGINX is triggered inside the proxy container using nginx -s reload, ensuring that traffic is now routed to the new version without downtime.
3. Update State:
   a. The script updates running_service to reflect that the newly activated service is now live.
4. Clean Up:
   a. The previously active (now outdated) frontend container is stopped and removed.
   b. Docker images are pruned with docker image prune -a -f to free up disk space and remove unused images.

This Blue/Green deployment strategy with Docker Compose provides a reliable and rollback-safe mechanism to update frontend applications. It isolates the new version in a separate container (blue or green), keeping the current one intact until all tests pass; the transition of live traffic is handled gracefully through a centralized NGINX proxy and the deployment is automated with clear separation of deployment, validation, and switching phases.

Features and Refined User Stories

User Story 1: Create Secret Note

As a user, I want to create a secret note so that my text is stored securely with encryption.

Detailed Acceptance Criteria:

- The user must provide a title (minimum 1 character)
- The user must provide note content/text (minimum 1 character)
- The user must provide a passphrase (minimum 8 characters)
- The note content is encrypted using AES-256-GCM with PBKDF2 key derivation (100,000 iterations)
- Each note gets unique salt and IV for maximum security
- The system displays validation errors for missing or invalid fields
- UI shows a success confirmation after successful creation
- The user is redirected to the notes list after successful creation
- The note appears in the notes list with its title and creation date
- The passphrase is never stored or transmitted in plain text

Implementation Details:

- Frontend: React form with validation using React Hook Form + Zod schema
- Backend: Fastify POST endpoint /notes with request validation
- Database: Prisma ORM with PostgreSQL storing encrypted content only
- Security: Crypto service using Node.js crypto module with symmetric encryption
- UI: Modern form with password visibility toggle and security hints

User Story 2: Read Secret Note

As a user, I want to retrieve a note so that I can view its content when I have the correct passphrase.

Detailed Acceptance Criteria:

- The user must provide the exact passphrase used during encryption
- The system successfully decrypts and displays the note content if the passphrase is correct
- The system shows an error message if the passphrase is incorrect
- The note content is rendered with full Markdown support including GitHub Flavored Markdown
- The system displays the note title, creation date, and formatted content
- The passphrase input field has a visibility toggle for user convenience
- The system shows a loading state while decrypting the note
- The decrypted content is never cached or stored in plain text

Implementation Details:

- Frontend: Protected route with passphrase form and Markdown rendering
- Backend: GET endpoint /notes/:id with passphrase query parameter
- Decryption: Server-side decryption using the same crypto service
- UI: Protection card component with form validation and error handling
- Markdown: react-markdown with remark-gfm for GitHub flavored markdown support

User Story 3: Manage Feature Toggles

As a DevOps engineer, I want to toggle specific UI features so I can run A/B tests and control feature rollouts.

Detailed Acceptance Criteria:

- PostHog integration allows enabling/disabling UI features remotely
- A/B testing splits users into distinct groups with different experiences
- Feature flags can be toggled without code deployment
- Analytics track user interactions with different feature variants
- Feature toggles work across different environments (development, production)
- The system gracefully handles missing or failed feature flag requests

Implementation Details:

- Frontend: PostHog SDK integration with feature flag checking
- Configuration: Environment variables for PostHog key and host
- Analytics: User interaction tracking for A/B test metrics
- Fallback: Default behavior when feature flags are unavailable

User Story 4: Browse and Search Notes

As a user, I want to browse all my notes and search/filter them so I can quickly find specific notes.

Detailed Acceptance Criteria:

- The system displays a list of all notes with titles and creation dates
- Notes can be sorted by title or creation date in ascending/descending order
- The system shows a search/filter interface for finding notes by title
- Each note in the list shows its title, creation date, and access button
- The list is responsive and works on mobile devices
- Loading states are shown while fetching notes
- The system handles network errors gracefully

Implementation Details:

- Frontend: TanStack Router with query parameters for sorting
- Backend: GET /notes endpoint with orderBy and ascending parameters
- UI: Responsive grid layout with sorting controls
- State Management: TanStack Query for server state caching

- Search: Client-side filtering by title with debounced input

User Story 5: Handle Errors and Edge Cases

As a user, I want the application to handle errors gracefully so I have a good experience even when things go wrong.

Detailed Acceptance Criteria:

- Network errors show user-friendly error messages
- Invalid note IDs redirect to a custom 404 page
- The 404 page has "Go Home" and "Go Back" navigation options
- Form validation errors are displayed inline with helpful messages
- API timeouts are handled with retry options
- The system shows loading states during operations
- Error boundaries prevent the entire app from crashing

Implementation Details:

- Frontend: Error boundaries with React and custom error pages
- Backend: Proper HTTP status codes and error messages
- UI: Toast notifications for transient errors
- Routing: TanStack Router with notFoundComponent handling
- API: Axios interceptors for global error handling

User Story 6: Responsive Design and Accessibility

As a user with different devices and accessibility needs, I want the application to work well on all screen sizes and with assistive technologies.

Detailed Acceptance Criteria:

- The application works on desktop, tablet, and mobile devices
- All interactive elements are keyboard accessible
- The application supports screen readers with proper ARIA labels
- Touch targets are appropriately sized for mobile devices
- Forms have proper labels and error announcements
- The application supports system theme preferences (light/dark)

Implementation Details:

- Frontend: Responsive design with Tailwind CSS
- UI: ShadCN components with built-in accessibility features
- Testing: Playwright tests for mobile and desktop viewports
- Styling: CSS custom properties for theme support
- Accessibility: ARIA labels and semantic HTML elements

User Story 8: Security and Privacy

As a user, I want my notes to be secure and private so I can trust the application with sensitive information.

Detailed Acceptance Criteria:

- Passphrases are never stored or logged on the server
- Each note uses unique salt and IV for encryption
- Input validation prevents injection attacks
- Regular security scans are performed on dependencies
- The application provides security recommendations to users

Implementation Details:

- Backend: Fastify with security headers and CORS configuration
- Encryption: AES-256-GCM with PBKDF2 key derivation
- Database: Only encrypted content stored, no plaintext
- Security: Snyk scanning for vulnerabilities in CI/CD
- Validation: Zod schemas for input validation