**Capstone Assignment Report:** Book Catalog with CI/CD Pipeline

Swan Saung Lwin

Diploma in DevOps (Summer 2025)

8th August 2025

<https://github.com/Swan276/books-catalog-api>

# Abstract

This project presents an end-to-end DevOps workflow for a Django REST API that provides CRUD operations for books catalog. I’ve setup CICD pipeline with Github Action on the main branch to test, check migrations, build docker image with semantic versioning, push image to github container registry, and update latest tag on values file, which will be watched by ArgoCD to deploy to Kubernetes cluster using Helm. Structured JSON logging with correlation id is centralized via Loki for traceability. This report explains the end to end workflow, how the docker image is being built, CICD pipeline, deployment and how observability is achieved. Challenges faced during the project and desired future improvements are also included in the report.

The instructions on how to setup and run the project is documented in the README.md file in the github repository.

# Contents

[Abstract 1](#_heading=h.ne5ek5kri66r)

[Contents 2](#_heading=h.b7lnugxui0pq)

[End to End Workflow Overview 3](#_heading=h.1iq4idpid639)

[Docker Image Build Process 3](#_heading=h.vzixdemklfrj)

[CI/CD Pipeline 4](#_heading=h.n1g54qm9zpvk)

[Github Commits & Pull Requests 4](#_heading=h.k82itz2w4wml)

[Helm Chart Deployment 5](#_heading=h.pnungqnklumj)

[Logging and Observability 6](#_heading=h.n7m2s9irng68)

[Lessons Learned & Challenges 7](#_heading=h.2ei3xqcb5oxc)

[Future Improvements 7](#_heading=h.9i8n7v7u41oh)

# End to End Workflow Overview

The Django project has CRUD REST endpoints for book-catalog api, which stores data on sqlite on development mode and postgresql on release mode. Logging middleware is set up so that the api can be observable. The application can be containerized and tested locally with postgresql using docker compose.

When a feature is ready and opens a pull request on the main branch, Github Actions run tests, apply migrations and check them so that issues can be caught before merging to main. On merge to main, the pipeline builds the Docker image, tags using semantic versioning, and then pushes it to the Github container registry. The pipeline then updates the latest tag to values file so that ArgoCD watching the changes, can sync and re-deploy the application on the Kubernetes cluster. As part of the release, a migration job runs against PostgreSQL database so that the latest migrations are applied. I can also easily rollback the deployment by reverting the image tag in the values file.

The deployment includes a replica-set with 3 pods, which will pull the image from the registry, a ClusterIP service, configmap and ingress to route external traffic at /api to the service.

For observability, logs emitted from the application are scraped and shipped to Loki so that they can be searched by endpoints or correlation ids.

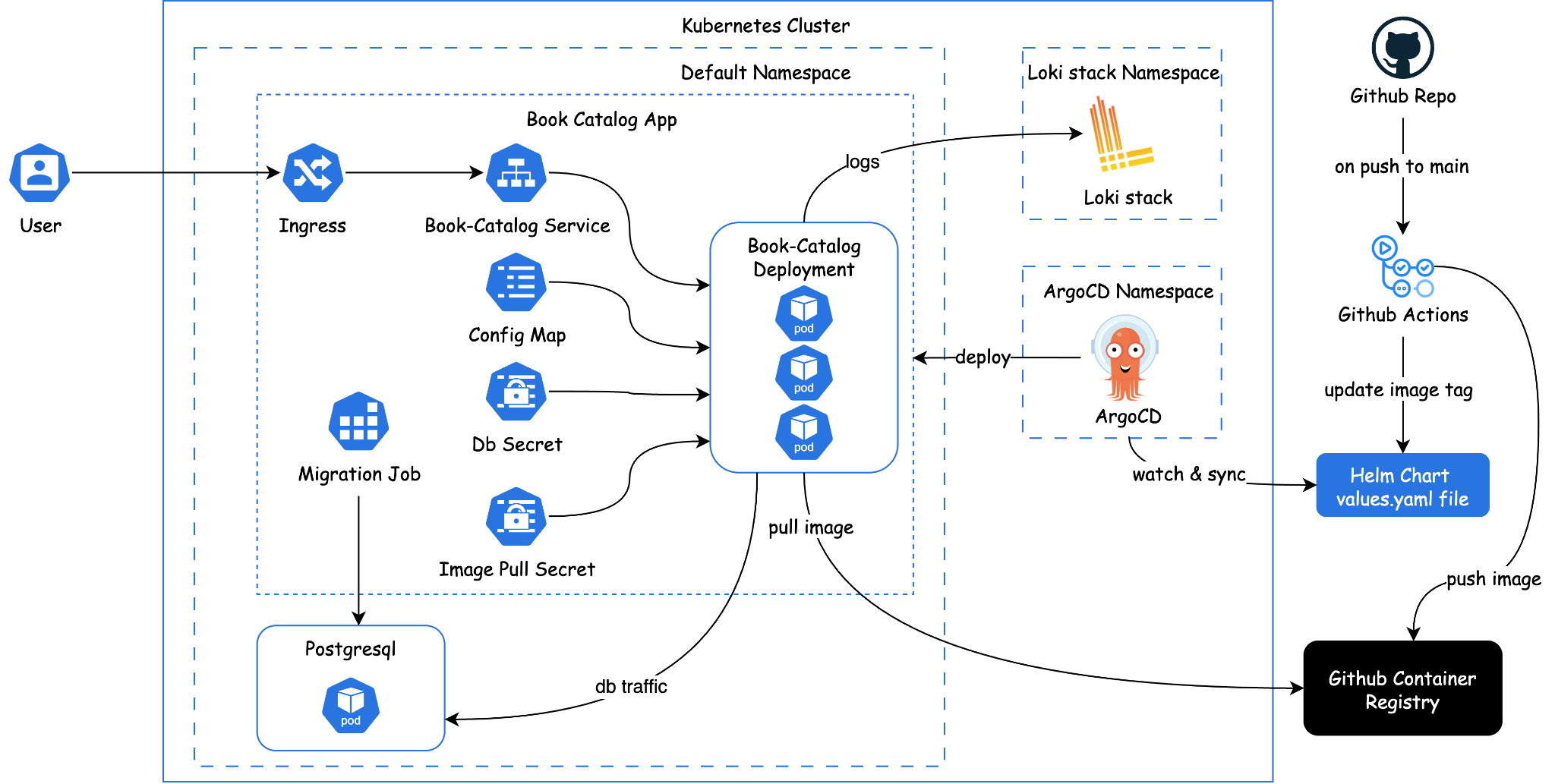


Figure. End to end workflow

# Docker Image Build Process

Docker image is built using Python 3.12-slim to match my virtual environment. My Dockerfile follows the following procedure:

* Copy requirements first to leverage the Docker layer cache
* Install python dependencies together with netcat to later be used
* And then copy the app code.
* Finally, the entrypoint script is executed, which includes instructions to wait for the Postgres database port to be up using netcat, run database migrations, and start the server.

Netcat is used to listen for the database port to be up because the app can start before the database is ready, which can cause failures. I also created a .dockerignore to avoid including unnecessary files to the build context, speeding up builds and reducing image size.

# CI/CD Pipeline

CI/CD pipeline is implemented using GitHub Actions.

On pull request to main:

* Run unit tests with pytest.
* Run migrations and check for missing migrations to prevent potential issues.

On push/merge to main:

* Repeat all previous checks to be safe.
* After all the checks are passed, create a semantic release version number from conventional commits. Also record all the commits into the CHANGELOG.md file to see which changes are included for each release.
* Build the Docker image and tag it with the previously created semantic release version and push it to github container registry.
* Update the image version in Helm chart values to point to the latest image tag and commit that change.

This design keeps Git as the single source of truth for what should be running. Argo CD watches the image version values file and syncs the cluster to deploy the latest version.

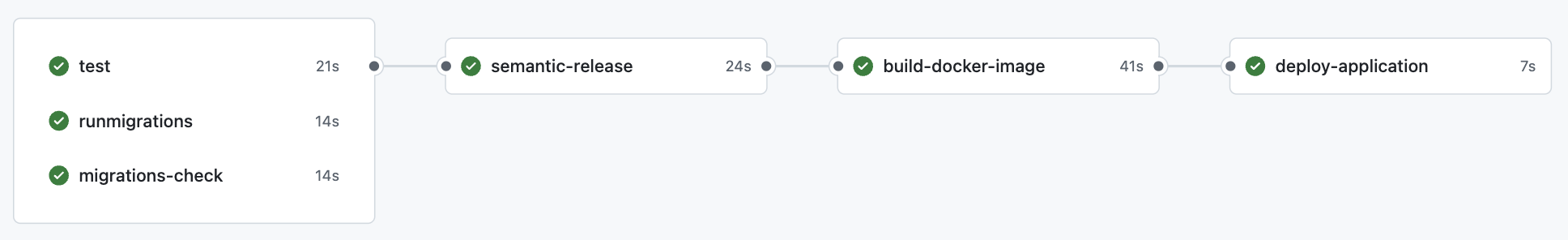


Figure. Github Actions on merge to main

# Github Commits & Pull Requests

Commit messages are written using conventional commit semantics so that each change can be understood easily upon review.

It is also important to make a pull request on the important branches such as main, develop, when working with a team or even alone, because we can have an extra chance to review the code so that any potential issues will be spotted manually or by the github action. When the code push is directly pushed, these checks are bypassed and potential issues may occur.

That’s why it is also important to set up branch protection rules on critical branches. Branch protection rules can protect specific branches from directly pushing or merging without passing the checks.

# Helm Chart Deployment

I packaged the application as a Helm chart that contains resources such as Deployment, Service, Ingress, a ConfigMap, and a Job that runs database migrations.

* **Deployment:** Deploying 3 pods in a replica set with container exposed on port 8000. Image is pulled from the github container registry using image pull secret, ghcr-token, created as kubernetes secret resource. Each container is given a resource of 100m to 500m cpu and 128Mi to 256Mi memory. All the environment variables are loaded from ConfigMap while database password is loaded from postgres secret created as a kubernetes secret resource.
* **Service:** ClusterIP Service that exposes the app internally on port 80 and forwards to the container’s port.
* **Ingress**: route external traffic containing prefix /api to the Service on port 80.
* **Config Map:** contains non-sensitive environment variables.
* **Migration Job:** pre-install/pre-upgrade job to run database migration. It ensures the database schema is up to date before new pods serve traffic.

Postgres database is installed using helm from values file containing database name, username and password. Both the application and the database are deployed in the default namespace, while ArgoCD and Loki stack are deployed in their own namespace.

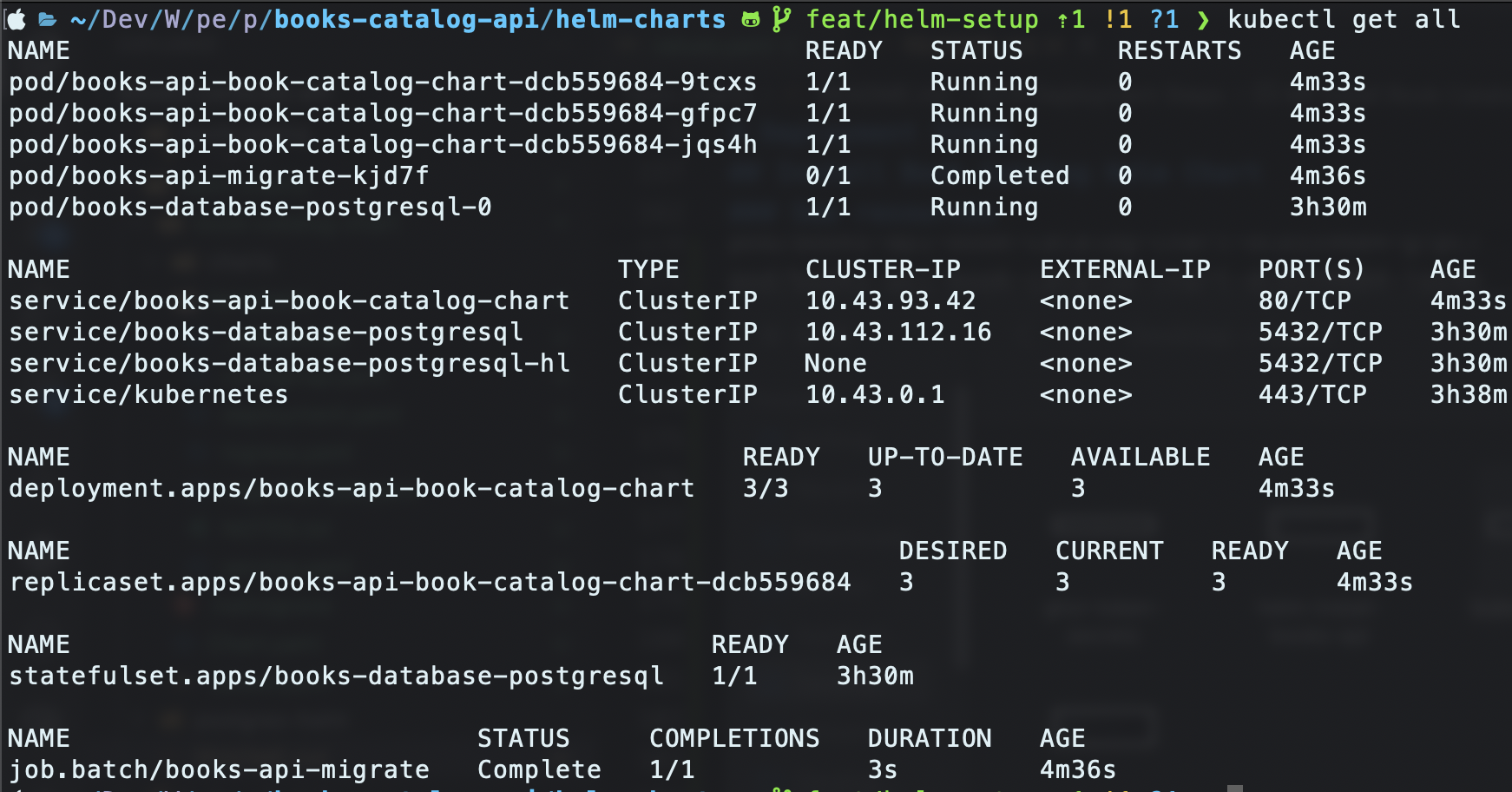


Figure. Kubernetes resources in default namespace

ArgoCD is continuously watching the image tag in values.yaml file in the github repository and a change in the value will kick off the deployment. When the tag changes, ArgoCD syncs the cluster, executes the migration job and rolls out the updated pods. If I need to roll back, I can just revert the image tag in values.yaml to a previous version, and ArgoCD redeploy the cluster back to that previous release.

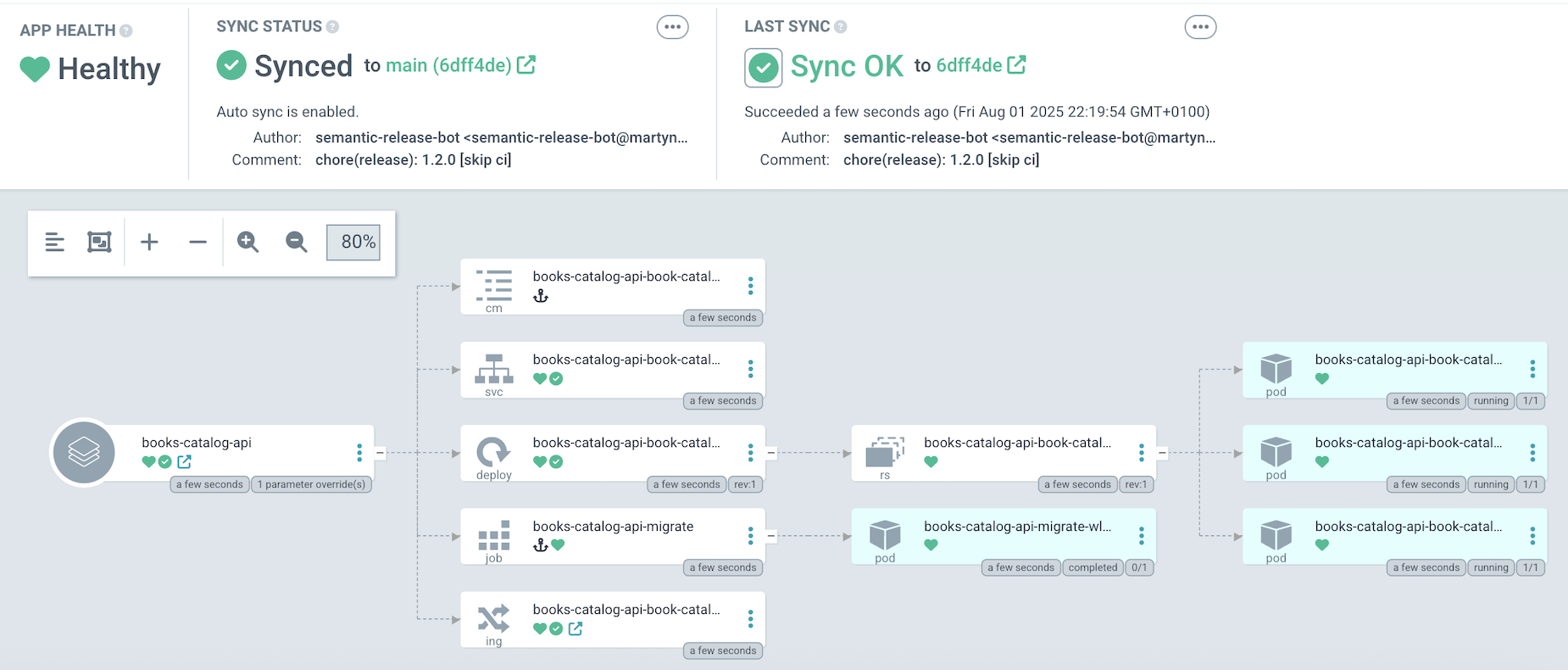


Figure. ArgoCD setup for book-catalog application

# Logging and Observability

For observability, I have added two custom Django middlewares.

1. **Correlation ID Middleware:** For each API request, it generates a unique id and attaches it to the response header. This makes it easy to trace each request and correlate with the error log if it exists.
2. **Request/Response Logging Middleware:** Logs the request time, method, path, status code, latency, request payload and response. These logs are formatted as JSON so that they can be easily formatted.

I also logged the unexpected errors in views as internal errors. The error log includes error message, exception type, traceback, and correlation id. With Loki and Grafana, I can quickly see the background of the api responses having 500 server errors by querying with correlation id.

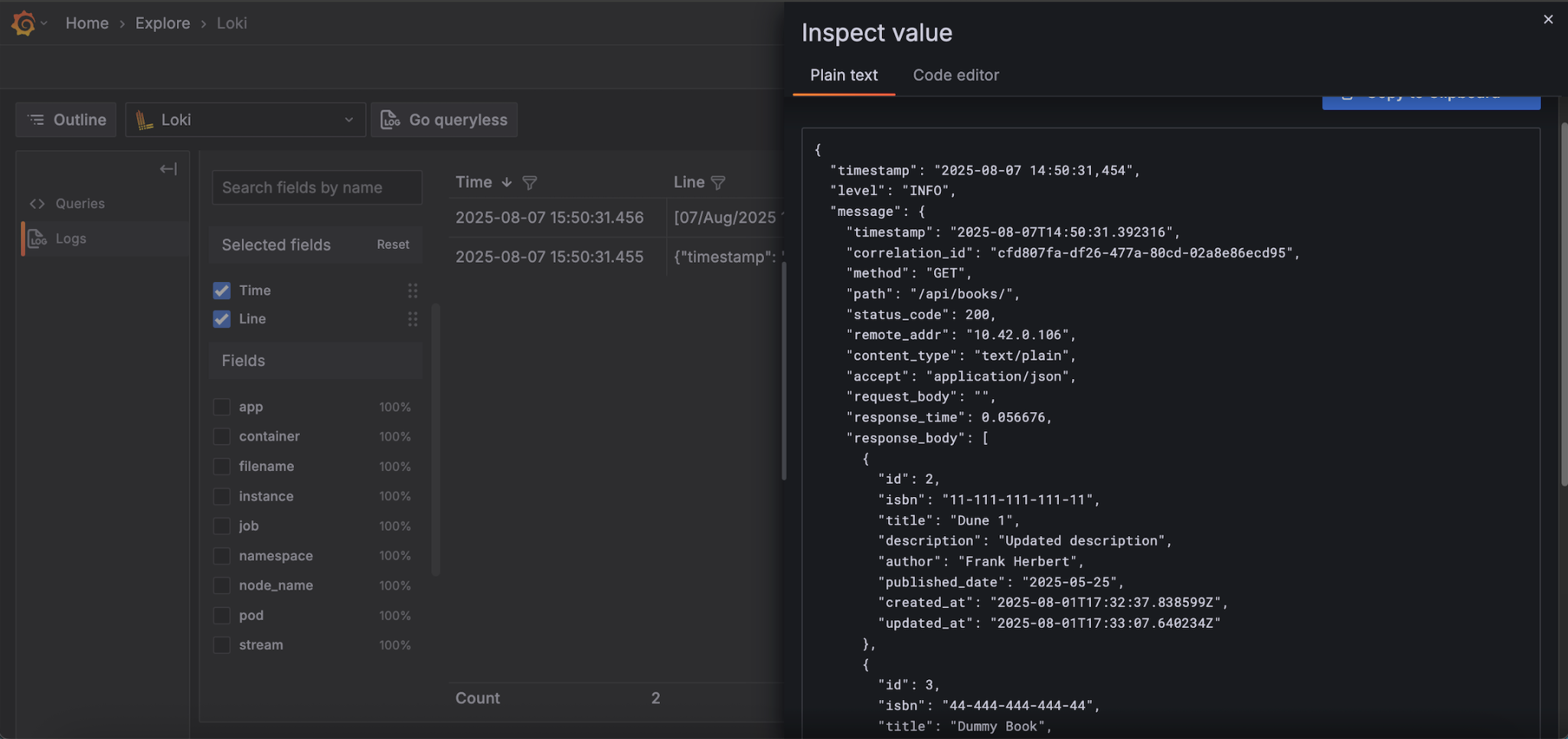


Figure. Querying logs in Grafana

Loki stack is deployed via Helm with Loki, Promtail and Grafana enabled.

**Promtail:** scrapes log files from nodes and pushes them to Loki.

**Loki:** log aggregation system that indexes metadata of the log rather than content.

**Grafana:** a visualization platform that connects to Loki as a datasource so you can query and visualize logs.

# Lessons Learned & Challenges

* Structured logs with correlation IDs were the biggest improvement for debugging.
* .dockerignore file helped ignore unwanted folders such as helm-charts, media files, etc.
* Have trouble with setting up branch protection as it is conflicting with committing changelog directly to the main branch.
* Also learned that you need to have the application name in ArgoCD same as the release name in helm, otherwise it creates new resources.

# Future Improvements

* Set up Github branch protection so that unverified changes cannot be pushed into the main branch.
* Set up CICD on the develop branch, together with a separate env values file so that the new feature images can be tested in the dev environment before deploying to the prod environment.
* Expand observability with Prometheus metrics.
* Deploy to cloud services like AWS or Azure and integrate with their offered services.