CI-CD Pipeline:

The CI/CD pipeline is fairly straightforward. First, I log in to my private DockerHub account using GitHub secrets. After that, I build the Docker image and push it to my Docker Hub repository.

The deployment process is implemented, but is commented out . what I did:

- SSH into the server where the application will be hosted.
- Fetch the latest Docker Compose file either from a GitHub repository or a secret manager like HashiCorp Vault.
- Update the Docker image reference in the Compose file with the newly pushed version.
- Run `docker compose up -d` to restart the service with the updated image.

I also recommend using hosted runners for the CI pipeline. They offer great flexibility and have the ability to cache Docker image layers, which significantly improves build times during repeated deployments.

Promtail should either be included in the same Docker Compose setup or be running separately in the background on the same host to make sure logs are correctly forwarded to Loki.

Docker image fixing and optimization :

We are using third-party Python packages like **Redis** and **FastAPI** in the application. Since I'm relying on a `requirements.txt` file in the Dockerfile, I first generate it using:
pip freeze > devops/requirements.txt
In the requirements I saw that the list is missing Uvicorn so I installed it with pip and updated the requirements file.

I also changed the base image from `python:3.9` to `python:3.9.23-alpine3.22`. This reduced the image size significantly—from **around 1GB down to 75MB**.

Performance Optimization :

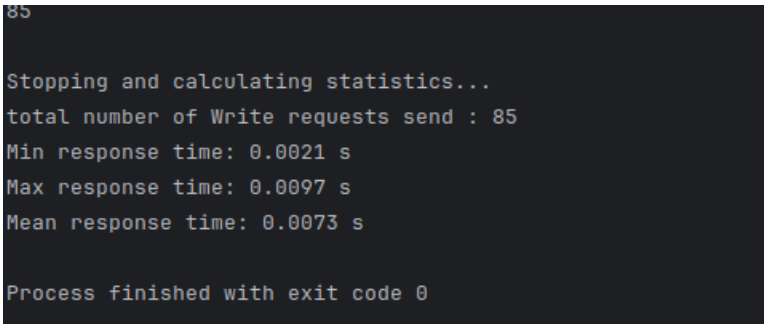
Here are a few things I kept in mind for performance optimization:

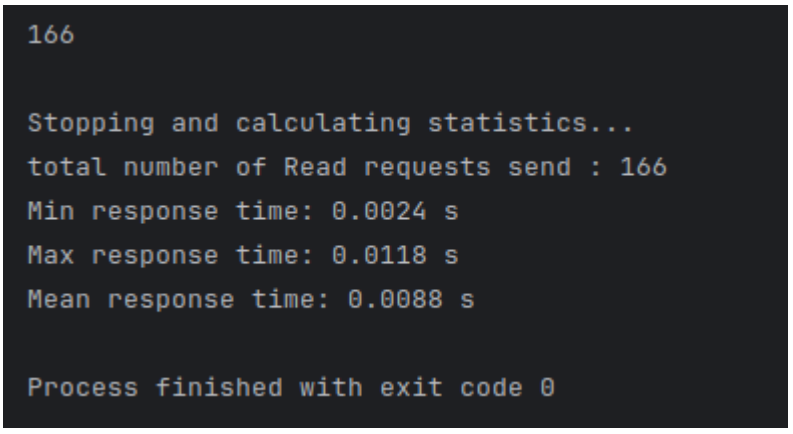
1. **All I/O operations**, including Redis calls, should be **asynchronous**. This helps keep the main thread free to serve new incoming connections.

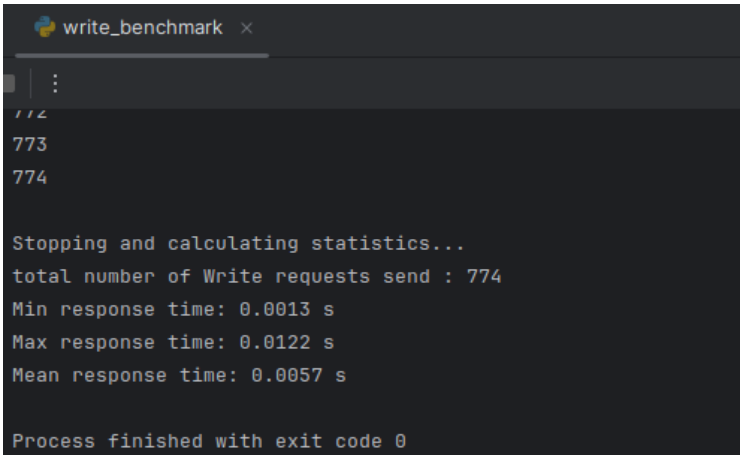
2. The **number of Uvicorn workers** can be adjusted depending on the expected load.

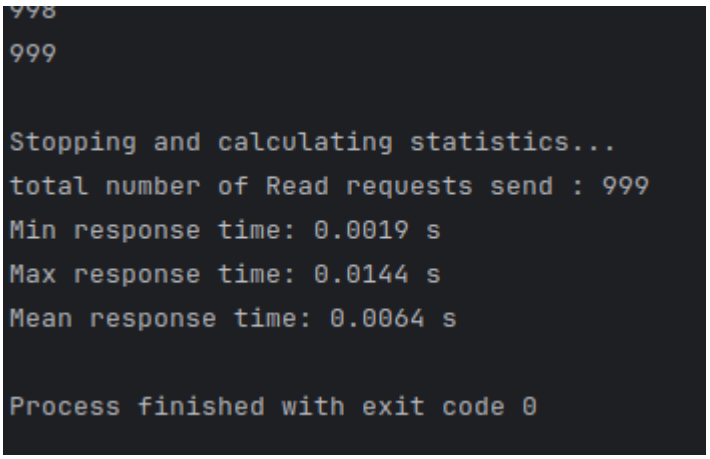
3. Since this is a sample app and not resource-heavy, vertical or horizontal scaling (like adding load balancers) can be considered for a real production deployment.

To conduct more thorough testing, I created two benchmarks — one for write operations and one for read operations — each running at 100 requests per second against localhost. Both benchmarks measure the mean response time. However, since the results at 10 RPS and 100 RPS were virtually identical, I haven't been able to identify a performance issue to begin optimizing or debugging.









Logging and Monitoring

Since the deployment environment wasn't specified, I recommend using the **Loki + Promtail** combination. It's easy to set up for both Docker and Kubernetes, and it offers good performance right out of the box.

My suggestion is to set up a **dedicated server or VM** to run Loki. Then, all other VMs can forward logs to that Loki instance using Promtail.

It's important to regularly **back up the configuration and data** for these services since they're interdependent. Setting up Loki was simple—I just set the retention policy to 90 days and left other settings as default. Loki also supports using **S3** and **external NoSQL databases**, but for now, local storage works fine as long as it's backed up.

Promtail needs to run on the **same host as the application**. With the provided config file, it will send all container logs to Loki.

To visualize logs, I use **Grafana**. It's set up with default credentials, but those will be changed on first admin login. I add Loki as a data source in Grafana and point it to the Loki instance. This lets me **monitor and query container logs**, set up basic alerts, and even write more complex queries if needed (though that may require additional configuration).