Group M MiniTwit Report
DevOps Spring 2025 KSDSESM1KU

Iulia Maria Negrila, Max Bezemer, Olivia Burca, Tore Kjelds, Su Mei Gwen Ho
iune@itu.dk, mbez@itu.dk, olbu@itu.dk, tokj@itu.dk, suho@itu.dk

May 29, 2025

Contents

1 Introduction

This report documents the design, implementation, and evolution of our ITU-MiniTwit system throughout the DevOps course. Working as a team, we began with a basic Python-based "Twitter clone" and migrated it to a Go-based architecture, applying DevOps principles at every stage. Over the semester, we incrementally integrated tools and practices such as containerization, infrastructure as code, continuous integration and deployment, monitoring, logging, and scaling strategies. This report outlines the technical decisions we made, the tools and workflows we adopted, and the lessons learned while building a reliable, testable, and maintainable system.

2 System

2.1 Design & Architecture

Our Minitwit system evolved during the course into the current architecture shown in Fig. 1. The entire system is hosted on DigitalOcean, where a web interface communicates with a backend API; the API reads and writes to a PostgreSQL database where persistent data is managed. In addition both monitoring and logging are integrated, including volumes to store logs and metrics. Fig. 1 shows how the servers (Droplets) are set up; each runs eight containers when active as well as a Keepalived daemon for the failover mechanism. At all times one server is active, serving the Minitwit web application and accessing the database. To ensure availability floating IPs are used to point to the appropriate server during failures and updates. All features mentioned here are expanded upon in coming sections.

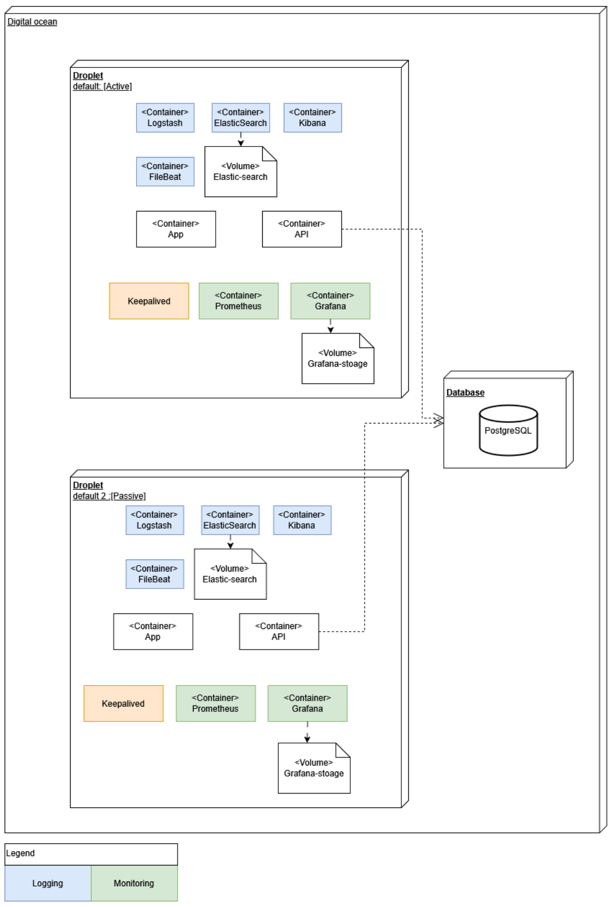

Figure 1: Deployment diagram

2.2 Dependencies & Tools

Development & Runtime

- **Go (Golang)** Primary language for the application.
- **GORM** ORM for Go used to interact with the database.
- **Gorilla Mux** HTTP router and URL matcher for Go, used to handle routing in the backend API.
- **Python** Used for testing, especially API and UI validation.
- **Docker & Docker Compose** Containerization of all services and orchestration for local and CI environments.

Infrastructure

- **Terraform** Infrastructure as Code to provision and manage DigitalOcean droplets, floating IPs, and DigitalOcean Spaces.
- **DigitalOcean** Cloud provider hosting the production infrastructure, including our high availability setup.
- **Vagrant** Used initially to provision reproducible local development environments.
- **Keepalived** Provides failover between active and standby droplets using floating IP reassignment.
- **Docker Hub** - Used to store the Docker images.

CI/CD & Automation

- **GitHub Actions** Automates testing, building, deploying, and releasing workflows.
- **gofmt** Enforces Go code formatting standards.
- **golangci-lint** Static analysis aggregator for Go code quality.
- **dockerfilelint** Linter for validating Dockerfiles.
- **GitHub CLI** Used for automated GitHub release creation.
- **SonarQube** Used to automatically analyze code quality.

Testing & Validation

- **Go test** For backend unit and integration tests.
- **pytest** Runs API-level tests written in Python.
- **Selenium (with Firefox + Geckodriver)** Powers UI and end-to-end testing in a headless browser.
- **unittest (Python)** Traditional functional tests.
- **SQLite3** Lightweight database used in testing environments to verify correctness.

Monitoring & Logging

- **Prometheus** Collects metrics from application containers, especially health and performance indicators.
- **Grafana** Provides dashboards and visualizations of Prometheus metrics at `localhost:3000`.
- **Elasticsearch** Central log storage and indexing engine for collecting application logs.
- **Logstash** Processes and forwards logs from containers into Elasticsearch.
- **Filebeat** Lightweight log shipper running on containers to collect logs.

- **Kibana** UI for querying and visualizing logs from Elasticsearch (at `localhost:5601`).
- **Keepalived** Also monitors app health to trigger failover, contributing indirectly to availability observability.

2.3 Subsystems Interactions

We chose two common scenarios to illustrate a dynamic view of our system. First the sequence diagram below, Fig. 2, shows the flow of events triggered by a user when they attempt to send a new message. It starts with the User pressing submit, having filled out the HTML form which makes the app issue a POST request to the API. Upon receiving the request an initial logging event records the attempt. The API queries the PostgreSQL database to retrieve the User's ID which it then uses to create a new message entry. Once the message is successfully saved in the database, confirmation that the message was stored without error is logged and two separate counters used for system monitoring are updated. Finally, the API returns a 204 No Content HTTP status code to the Simulator to signal that the message was properly processed.

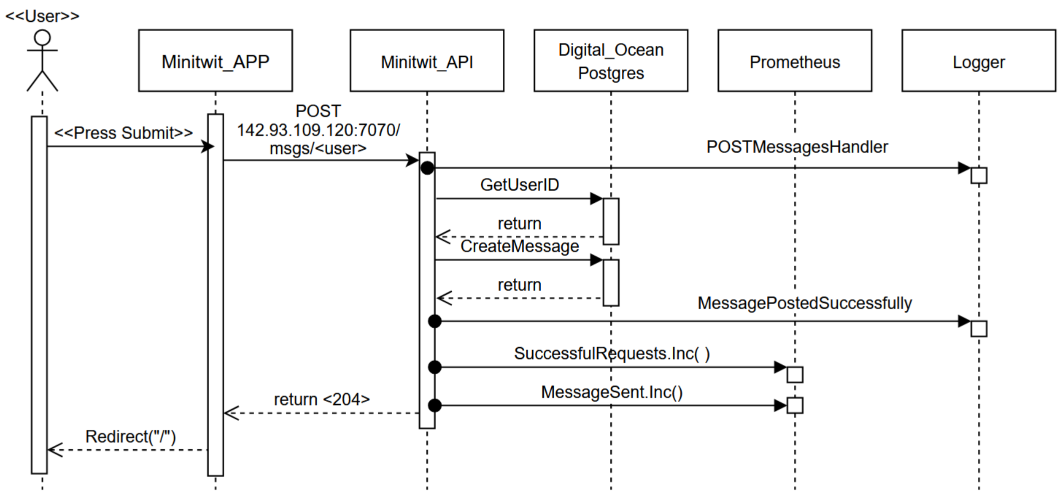

Figure 2: Sequence diagram of a successful sent message from the user's perspective

The next sequence diagram, Fig. 3, shows a common interaction between the Simulator and our system this semester. When an error prevents a User from being created, the Simulator may later attempt to send a message through a **POST** request at the "*/msgs/«User»*" endpoint. This is unsuccessful since the User does not exist and therefore cannot post.

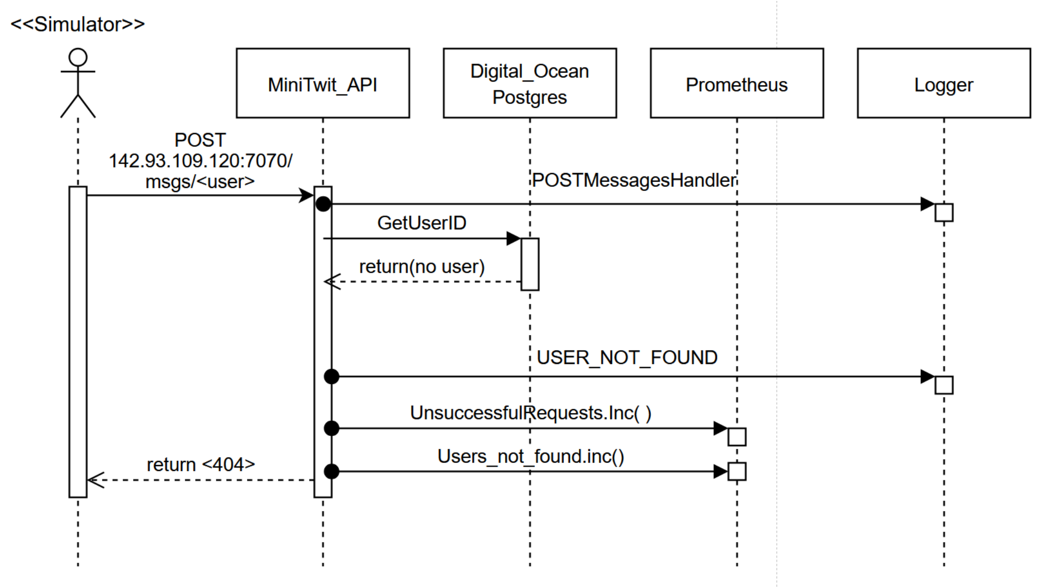

Figure 3: Sequence diagram of the simulator trying to post a message as a non-existent user.

2.4 Current State

Weve been using SonarCloud since March 28 to monitor code quality. The Quality Gate passes, but deeper analysis highlights several areas for improvement:

- **Code Duplication:** Currently at 4.4%, exceeding the 3% recommended threshold. New code remains within limits, but duplication is mainly due to repeated literals and logic across components.
- **Maintainability:** 55 open code smells, many flagged as high severity. These are mostly related to repeated strings that should be refactored into constants.
- **Reliability:** 10 minor issues, no blockers.
- **Security:** No active vulnerabilities, but 21 security hotspots, mostly due to hardcoded credentials in test files (e.g., `pwd = "super_safe"`). These dont impact production but should be reviewed.

Summarising the above, the overall maintainability and reliability ratings sit at A, while the security rating is E. Technical debt is estimated at 6.5 hours. The codebase is in stable condition, but is in need of some maintenance to improve long-term quality.

3 Process

3.1 Continuous Integration and Deployment (CI/CD)

To automate building, testing and releasing our ITU-MiniTwit system, we implemented a CI/CD pipeline using GitHub Actions. We chose GitHub Actions due to its native integration with our GitHub-hosted repository, ease of configuration, and support for open-source projects. This allowed us to set up an effective pipeline without maintaining external CI infrastructure, which aligned well with the size and scope of our team project.

Pipeline Overview

Our pipeline is triggered on every push to the `main` branch and consists of three main stages: **build**, **test**, and **deploy**.

1. **Build Stage**
   - The pipeline checks out the code and logs into Docker Hub using secrets stored securely in GitHub.
   - It then builds two Docker images:
     - `devoops-app` (the frontend)
     - `devoops-api` (the backend)
   - These images are tagged and pushed to Docker Hub, using cache layers for efficiency.

2. **Test Stage**
   - After successful builds, the pipeline spins up a full test stack using `docker-compose_test_local.yml` which runs containerized tests.
   - This includes integration, API, UI, and end-to-end tests. The pipeline will fail fast if any test fails, stopping the deployment stage to ensure only verified code is released.

3. **Deploy Stage**
   - This stage involves deploying to our remote infrastructure on DigitalOcean using a `blue-green deployment strategy` based on two floating IPs. The full upgrade logic and IP switching process is described separately in the Upgrade Strategy section.
   - A step using `terraform init` is also executed to set up the infrastructure. While this ensures the environment is up-to-date, we acknowledge that re-running `terraform init` and `apply` on every push may be excessive and could be optimized to only run when infrastructure-related files change.

Static Analysis & Quality Gates

We integrated multiple static analysis tools as quality gates into a pipeline that runs when a pull request is created, to catch code issues early and enforce consistent formatting and linting:

- `gofmt` checks Go code formatting.
- `golangci-lint` performs static analysis on Go files.
- `dockerfilelint` validates all Dockerfiles.

Release Automation

We automated our release process using a GitHub Actions workflow that triggers on tags matching `v*`, creating versioned GitHub Releases with auto-generated notes. This ensures consistent, traceable production releases aligned with our deployment flow.

3.2 Monitoring

We use Prometheus and Grafana to monitor our systems health. Metrics are instrumented with the Prometheus Go client library, registered via the default Prometheus registry, and exposed through the *metrics* endpoint for Prometheus to scrape.

For visualization we use Grafana, that uses prometheus as a data source and enables us to create and customize panels that visualizes the different metrics.

In relation to the James Turnbull monitoring maturity model[1], ours would be considered reactive as we track specific errors in relation to specific endpoints and overall availability.

We do not do any front-end monitoring, as the course was primarily focused on the simulations interaction with our API.

3.2.1 Metrics monitored

We defined the following endpoints. Some, like *Successful requests* track multiple endpoints and aggregate them in single time line for easier viewing and a reduction in overall panels in Grafana. The full dashboard can be seen in figure 5

- **Uptime**
- **Successful requests (Tracks multiple endpoints)**
- **Unsuccessful requests (Tracks multiple endpoints)**
- **Follows**
- **Unfollows**
- **Successful messages sent**
- **Users not found**

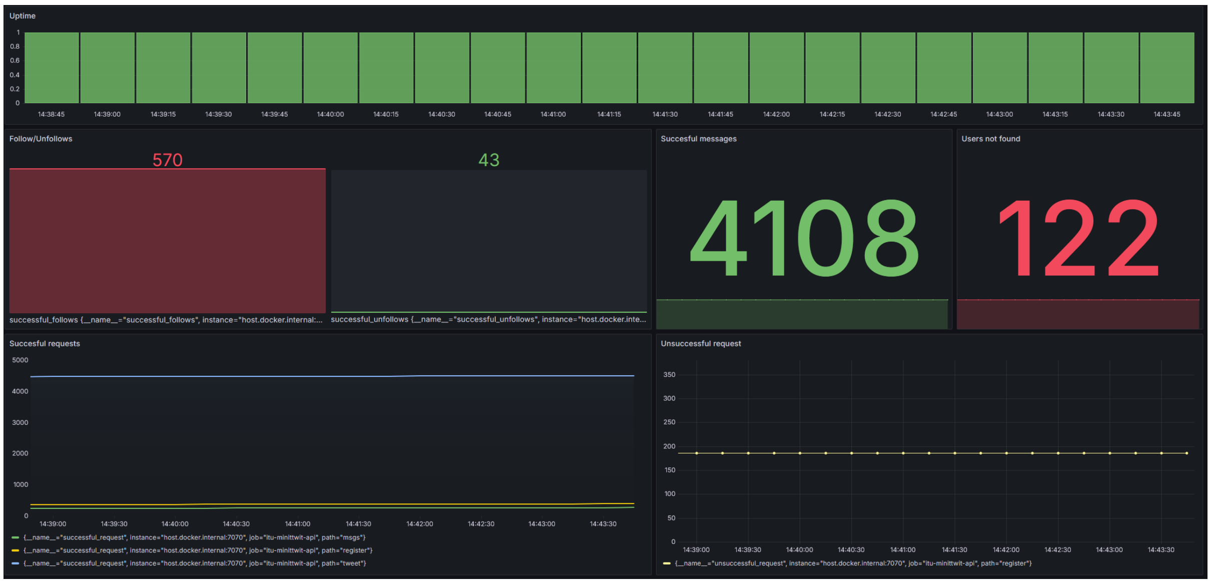
Uptime
Follow/Unfollows
570
43
successful_follows {__name__="successful_follows", instance="host.docker.internal:...
successful_unfollows {__name__="successful_unfollows", instance="host.docker.inte...
Succesful messages
4108
Users not found
122
Succesful requests
{__name__="successful_request", instance="host.docker.internal:7070", job="itu-minittwit-api", path="msgs"}
{__name__="successful_request", instance="host.docker.internal:7070", job="itu-minittwit-api", path="register"}
{__name__="successful_request", instance="host.docker.internal:7070", job="itu-minittwit-api", path="tweet"}
Unsuccessful request
{__name__="unsuccessful_request", instance="host.docker.internal:7070", job="itu-minittwit-api", path="register"}

Figure 4: Screenshot of the Grafana dashboard showing API monitoring metrics

3.3 Logging

For getting the right logs out of our Go application, we used a package called Logrus, which gave us full control over what we wanted to log. By setting the logging level (currently WarnLevel) and using function-level flags we avoided unnecessary verbosity. Since the application handles considerable traffic we deliberately avoided logging every event. Most functions only log when they exceed a 2-second execution threshold to help us focus on performance bottlenecks. Were also mindful of not logging any sensitive information such as user credentials, tokens or personal data, to ensure compliance with security best practices. However, we dont yet have a formal log rotation or retention strategy in place, which is something we aim to address as the system scales.

To aggregate logs, we use the ELK Stack. Filebeat collects logs from the Docker containers, and sends these on to Logstash where they are parsed and enriched. Then they are sent to Elasticsearch, which indexes them for efficient querying. Finally, we use Kibana to visualize and analyze the logs through dashboards and search interfaces. This centralized log aggregation gives us access to real-time insight into our distributed application.

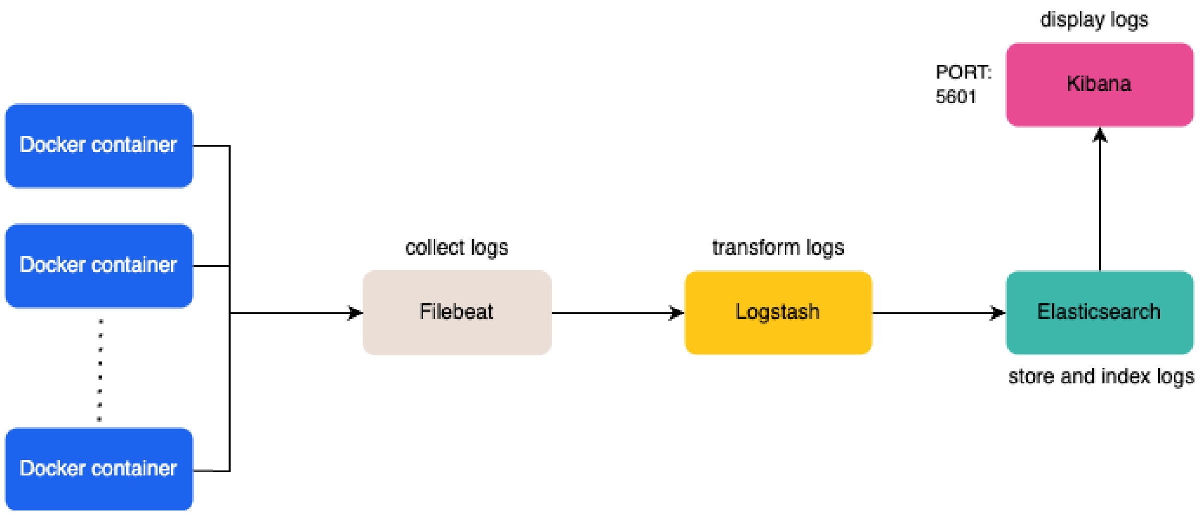

Figure 5: ELK stack used for logging

3.4 Security Assessment

Our security assessment revealed threats across various parts of our system. Assets at risk are listed below, along with risk scenarios and how to mitigate them:

- **Web Application** - An attacker performs SQL injection to extract sensitive user data. This is already mitigated by using ORM methods. A further step would be to automate back-ups of the database.
- **Database** - An attacker scraps Github for exposed ports, holding our database for ransom. Again, we should back up the database regularly. Configure firewalls for all droplets and update Docker Compose files.
- **API** - An attacker overwhelms the network, causing it to go down. We could avoid this by using Distributed Denial-of-Service (DDOS) protection tools which detect artificial traffic.
- **Github Repository** - Attacker hacks member account and steals Github secrets, exposing all other assets. In general, vulnerable passwords should be strengthened and two-factor authentication (2FA) set up.
- **Server** - An intruder gains remote access to the server via SSH, compromising the service. This is partially mitigated as we use GitHub secrets and avoid accidental exposure, however we should ensure all members have 2FA.
- **Logs** - If important events (internal or attack) are not logged properly they cannot be traced or followed up. We do already log a lot of errors however there is room for optimisation.

- **Monitoring** - Suspicious activity occurs however the team is not notified resulting in long response times. Ideally, our Grafana dashboard is populated with crucial metrics, e.g. number of requests per sec, and Grafana alerts are activated.
- **Data in Transit** - Internal communications are intercepted by malicious actors, exposing sensitive information. A Transport Layer Security (TLS) is needed to authenticate our servers and encrypt data in transit.
- **Source Code** - Dependencies get outdated; updates may be incompatible and affect functionality of app. These bugs can be exploited by attackers. We could use tools such as Dependabot to keep our systems up-to-date.

The following risk matrix reflects which threats are most important to focus on. Due to time constraints the exposed ports have been addressed using a Docker Compose Linter, and . Ideally, we would also integrate Dependabot, optimise monitoring metrics, use TLS, configure firewalls and automate back-ups of our database as soon as possible.

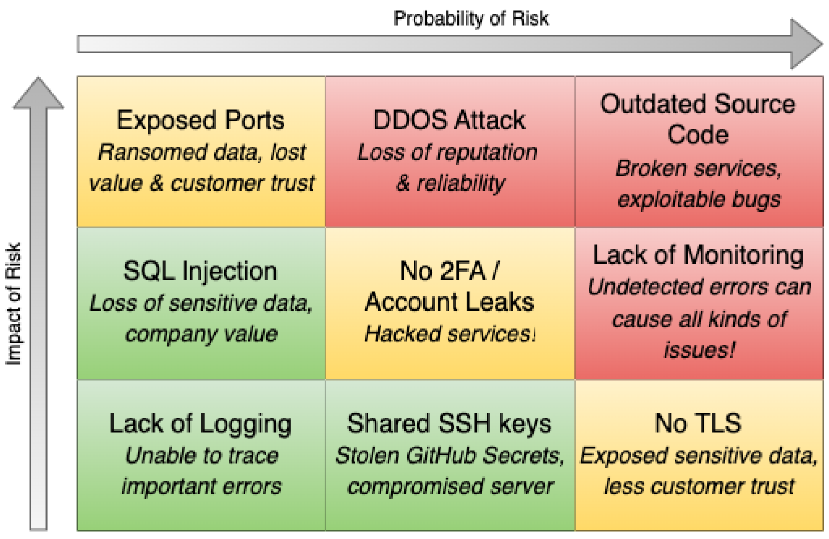
Probability of Risk
Impact of Risk
Exposed Ports
Ransomed data, lost value & customer trust
DDOS Attack
Loss of reputation & reliability
Outdated Source Code
Broken services, exploitable bugs
SQL Injection
Loss of sensitive data, company value
No 2FA / Account Leaks
Hacked services!
Lack of Monitoring
Undetected errors can cause all kinds of issues!
Lack of Logging
Unable to trace important errors
Shared SSH keys
Stolen GitHub Secrets, compromised server
No TLS
Exposed sensitive data, less customer trust

Figure 6: A risk matrix which takes into account current state of system and mitigations (if any).

3.5 Scaling and Upgrade Strategies

3.5.1 High Availability via Hot-Standby Setup

To enhance the availability and resilience of our application, we implemented a **hot-standby server setup** instead of using a container orchestration system like Docker Swarm. In this architecture:

- Only one droplet is active at a time, handling all traffic.
- A second droplet is in standby mode, kept in sync and ready to take over instantly if the active one fails.
- A DigitalOcean floating IP is used to route traffic to the active server and is automatically reassigned during failover.

Implementation Steps

We followed these steps to implement the hot-standby architecture:

- Created a snapshot of the initial (active) droplet and used it to spawn the standby server.
- Assigned a floating IP to the active droplet.
- Installed and configured Keepalived on both droplets:

  - `check-api.sh` monitors the health of the application.
  - `master.sh` reassigns the floating IP upon failure detection.
  - `keepalived.conf` and `keepalived.service` manage and start the Keepalived daemon.
- Added necessary firewall rules and database access permissions on DigitalOcean.

Why We Chose Hot-Standby over Docker Swarm

- **Simplicity**: Swarm introduces added complexity and overhead in managing a cluster, which was not justified for our relatively simple application architecture.
- **Predictable failover behavior**: With hot-standby, only one server is live at a time, making debugging and maintenance easier.
- **More control over failover**: The Keepalived setup gives us full control over the logic that triggers failover.

While our hot-standby setup improves availability, it does not horizontally scale the API, meaning only one server handles traffic at a time. However, we opted for vertical scaling by upsizing both droplets and the database to better handle increased load, which we found sufficient for our current scope. If the system was under higher demand, we could have implemented horizontal scaling by running multiple active API replicas behind a load balancer or using a Docker Swarm cluster with service replicas and built-in load distribution.

3.5.2 Blue-Green Deployment Strategy

To handle zero-downtime updates and safe rollbacks, we implemented a Blue-Green Deployment Strategy, building on the infrastructure we already had for scaling.
We assigned a second floating IP to the standby (passive) droplet. This enabled our CI/CD pipeline to:

1. Deploy the new version of the application to the passive server (Blue).
2. Swap the floating IPs using the DigitalOcean API after a successful deployment, making Blue the new active server (Green).
3. Redeploy to the now-passive (previously active) server to bring both environments in sync.

This strategy ensures that:

- Deployments are atomic and safe: if the Blue deployment fails, Green remains untouched and continues serving traffic.
- Traffic is only routed to stable, verified servers, avoiding mid-deploy errors.
- Rollbacks are simple: just reverse the IP swap.

Why We Chose Blue-Green over Rolling Updates

- **Minimal complexity**: Our infrastructure is not container-orchestrated, so rolling updates would be harder to coordinate reliably.
- **Clear separation** between active and staging environments.
- **Natural fit with hot-standby**: Using floating IPs made Blue-Green deployment easy to implement with our existing failover setup.

4 Reflections

4.1 Lessons in Evolution & Refactoring

- Migrating from SQLite to PostgreSQL introduced several compatibility and configuration challenges, particularly around data types, secure connectivity, and credential management, all of which required adjustments in both infrastructure setup and application code.
- Investigating tasks thoroughly before starting them is crucial. For example, introducing GORM to our application was a long and tedious process which should have been better explored beforehand. In general, a complete overview over the entire system would have informed some crucial decisions and saved us pain/time/both - we now understand why time is well spent on designing a system before building it.
- Refactoring should be broken down into tidbits and tackled with care. Also, Github best practices should not be taken for granted as common knowledge. A large refactoring of the code base was committed and pushed one night, and following day a direct merge into the main branch happened accidentally. It was time-consuming and stressful to fix, but we managed!

4.2 Lessons in Operation

- After migrating to PostgreSQL, our API suddenly stopped connecting to the database. We discovered this was due to too many open connections each request was opening a new connection. The solution was to refactor the API to use a single persistent database connection instead of reconnecting per request.
- Storage limitations on the remote server highlighted the need to avoid storing data inside containers and to use external volumes for persistence. To solve these issues, we also scaled vertically by increasing the size of both the droplet and the database instance.
- Although the group was quite good at checking the status of our system manually (very not-DevOps like), occasionally the system had been down for an extended period of time without anyone noticing. It turned out that we were not monitoring the most useful metrics nor setting alerts for them; at the same time, we found our logging stack challenging to work with. We learned that familiarity with both and quick implementation of missing metrics can go a long way.

4.3 Lessons in Maintenance

- We also had to update Dockerfiles and Docker Compose paths when switching to pulling images from Docker Hub. SSH command chaining required extra care due to syntax differences, and .bash profile needed to be sourced manually to ensure environment variables were available during remote setup and deployment.
- We learned that budget constraints can introduce challenges to maintaining the system as demand grows, but it can also scope or simplify decision-making when so many solutions (e.g. CPU type, RAM size, etc) are available.

4.4 On Using Large Language Models

The team is comprised of people with a wide range of experience; some preferred using LLMs better than others. Those who used LLMs mainly used Gemini and ChatGPT to help us bridge knowledge gaps in the various areas of the course which were new to some if not all of us. This was especially helpful due to the high pace of the course and conflicting schedules, allowing us to maintain momentum and collaborate better. In addition, Gemini was useful as a steady hand during Git merges gone wrong and doomloop rebasing. That being said, we also experienced the negative effect LLMs can have on the development process. In particular, separate individuals using LLMs heavily to assist in the ORM transition lend to more time spent on debugging and merging of code. This process exposed how important it is to understand fundamentals of the code base before relying on LLMs, especially when it comes to testing functions during web development. It is clear sometimes LLMs are not suitable replacements for knowledge sharing within the team.

We used LLMs primarily to help us understand unfamiliar tooling, clarify architectural patterns, and resolve language-specific issues, particularly as we transitioned from Python to Go. Given the steep learning curve associated with Gos syntax, type system, and concurrency model, LLMs were quite useful in bridging knowledge gaps. This support allowed us to maintain momentum with the weekly exercises and contributed to better collaboration, even among those with limited prior programming experience. That being said, we also experienced that LLMs can have a negative effect on the development process. In particular, after relying on an LLM to write part of our code for the ORM transition, we ended up spending much more time on debugging this code and aligning it with its constant than time saved by the using the LLM in the first place.

4.5 On Ways of Working

Compared to previous projects, our team adopted a clear DevOps style by automating the entire buildtestdeploy cycle through CI/CD pipelines, managing infrastructure with code, working with Github Issues and monitoring the system in production. We focused on fast feedback through automated testing and logging, which helped us catch and resolve issues early. This approach made our workflow more efficient, reliable and aligned with real-world DevOps practices.

Given that most aspects of this project were new to us, we worked fairly smoothly as a team. We shared decision-making well, did our best to split tasks fairly, and managed expectation well according to ability and time available. The main challenge was plugging knowledge gaps in a fast-moving project while juggling other coursework. Time constraints and shifting priorities are the enemy of DevOps - it is ideal that team members are not spread over too many projects, and that there is dedicated space for knowledge sharing and hand-offs.

A Appendix