IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance: Exam Project

Group C	
Mathias Gleitze Hoffmann	mglh@itu.dk
Martin Lupa Groppelli	mlup@itu.dk
Anne Sofie Rasmussen	soad@itu.dk
Cecilia Maria Fröhlich	cefr@itu.dk
Tomas Federico Maseda	tofm@itu.dk
Eleonora Borovic	eleb@itu.dk

We have received the source code for a miniature version of X (formerly known as Twitter) in Python, for us to rewrite using another programming language. From there we have been asked to containerize our application, create a database, set up infrastructure using code, add a CI/CD chain, add logging and monitoring, and add container orchestration to enable our application to run in a distributed manner.

KSDSESM1KU
Spring, 2024

Contents

1 Systems

1.1 Design and architecture

We have developed 'Csharp-Minitwit' using C# and the .NET 8.0 runtime, chosen for its strong, statically typed ecosystem and long-term support.[3] The application uses the ASP.NET Core framework[4], with razor pages for rendering HTML pages, and we are following a Model-View-Controller (MVC) pattern with dependency injection, for easier maintainability and testability. We use PostgreSQL with Entity Framework Core for database management (an ORM), making subsequent database alterations and migrations easy.

In 'Csharp-Minitwit', the main modules include Controllers, Models, Views and Services. They represent underlying files or directories. In Figure 1, the dependencies between the modules can be seen. The Controller module is the main component, heavily dependent on the other modules. If we decompose the controller module it can be seen that it consists of two underlying packages, corresponding to the two interactions there is with the system, through the apicontroller, fitted to the simulator, or as a client through the homecontroller. How these interactions are, is further elaborated in section 1.2.

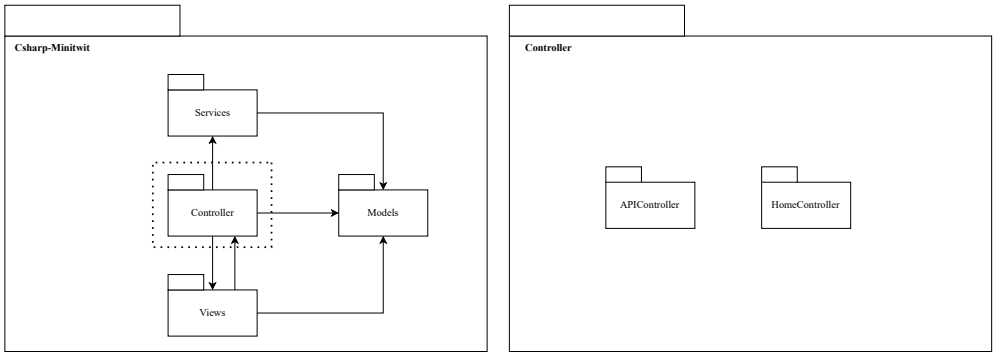

Figure 1: Simplified Csharp-Minitwit package view

Figure 2 shows the infrastructure in our system. It can be seen, how he application is hosted on DigitalOcean. We have a Docker Swarm to orchestrate containers and use Terraform for automating provisioning and configuration of the infrastructure.

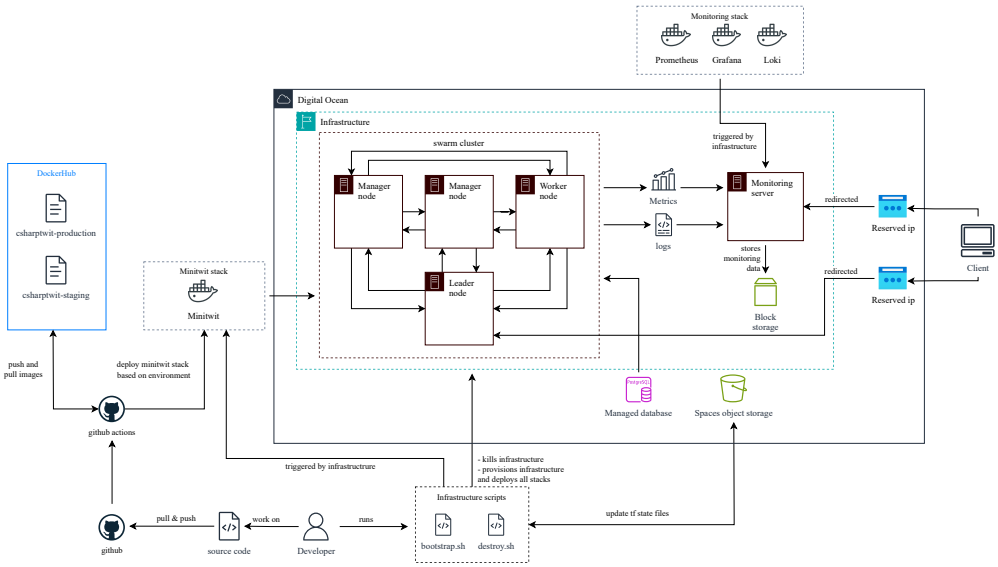

Figure 2: System Architecture Diagram

1.2 Interactions of subsystems

Incoming requests to the application are handled by either the `APIController` or the `HomeController`. Both provide similar endpoint logic but are meant to be used by the simulator and regular users respectively. A request's path will match a specific endpoint, prompting the application to perform a corresponding database operation. For instance a POST request to */login* will invoke the `GetByUsername` method from the `userRepository` and request the user information through an SQL SELECT query. The ORM will map the results back to C# objects, which the controller returns in the HTTP response to the user. Along with this Prometheus and Grafana collect metrics to visualize. Data from logging is aggregated through Loki and Serilog. Below are two sequence diagrams showing how a user and the simulator interacts with our subsystems.

In Figure 3 is a sequence diagram showing how a user successfully follow another user and gets a response from the system.

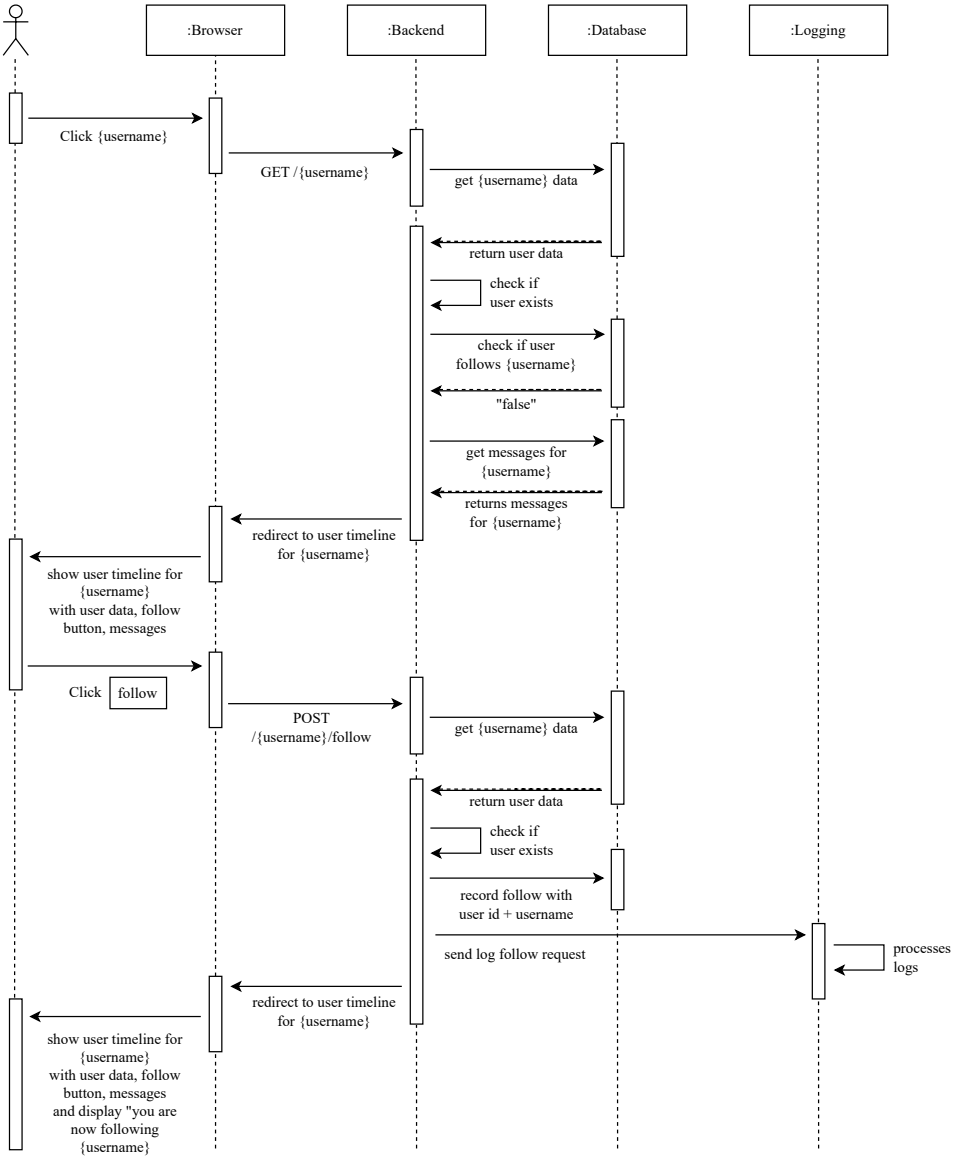

Figure 3: Sequence diagram of a successful follow scenario from user perspective.

In Figure 4 is a sequence diagram illustrating how a request from the simulator is handled in the system.

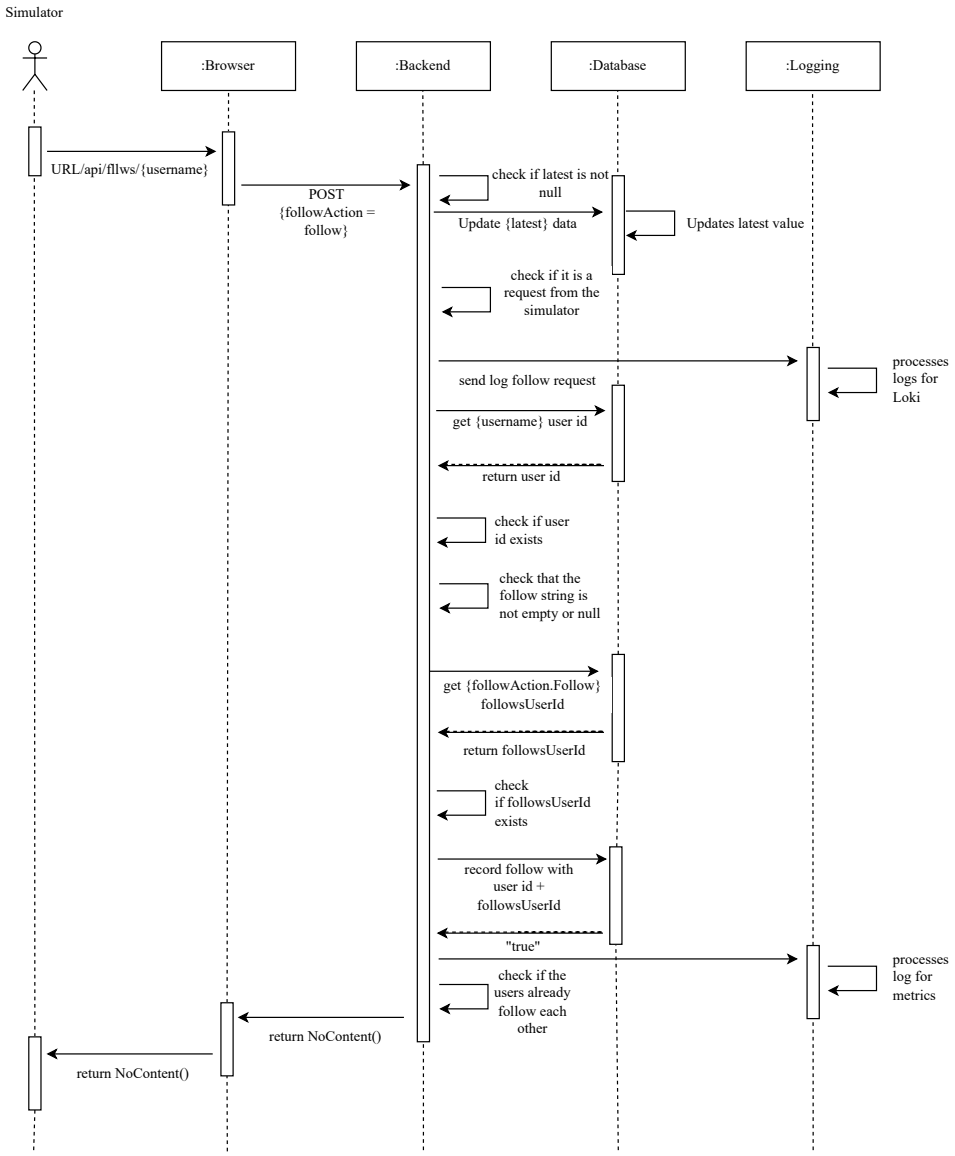

Figure 4: Sequence diagram of a successful follow scenario from the simulator over the API.

1.3 Dependencies and technologies

Application Dependencies

Dependencies are handled using dotnet's package manager, NuGet:

- Npgsql.EntityFrameworkCore.PostgreSQL
- OpenTelemetry.Exporter.Prometheus.AspNetCore (exports telemetry data to Prometheus)
- OpenTelemetry.Extensions.Hosting
- Microsoft.EntityFrameworkCore (ORM)
- Serilog (structured logging library)
- Swashbuckle.AspNetCore (auto-generated Swagger documentation)

Technologies used

A quick overview of the technologies used:

- **GitHub**: Code repository, versioning and project management.
- **Github Actions**: CI/CD service, deploying code.
- **SonarCloud**: quality analysis for code.
- **CodeClimate**: code quality analysis.
- **dotnet format**: automatic formatting of C# code.
- **HadoLint**: linting Docker file.
- **Digital Ocean**: hosting of virtual servers, databases and volumes.
- **Terraform**: provisioning and management of infrastructure on Digital Ocean.
- **Docker**: containerization of the application for deployment.
- **DockerHub**: for storing our docker Images.
- **DockerSwarm**: for clustering and orchestration.
- **Prometheus**: monitoring system and time series database that collects metrics.
- **Loki**: log aggregation system integrated with Grafana for viewing logs.
- **Grafana**: visualization of metrics and logs.
- **Vagrant**: initially used for provisioning.

Github

The application is versioned, tested, quality assured and deployed using GitHub. Our group has been following the Gitflow workflow to avoid merge conflicts, and ensure tractability. Gitflow also allows us to have a 'develop' and a 'main' branch, enabling us to easily set up a staging server, for integration testing.

Our GitHub repository is also used for project management, creating issues and documentation.

Deploying our application to our servers is done using GitHub Actions, creating Docker images and pushing them to the relevant environment. GitHub actions also runs unit tests, formats our source code using 'dotnet format' and analyzes it using SonarCloud and CodeClimate.

GitHub was chosen because all members of the group was familiar with it. Furthermore, we chose GitHub actions for the CI/CD service, because it works well when our code repositories are already on GitHub.

DigitalOcean

We have chosen DigitalOcean as our cloud provider, running our application on provisioned virtual machines that we manage our selves. DigitalOcean has been chosen for the sole reason that they offered $200 in credits for students — enough for the entire project.

Database

As mentioned, the application uses a PostgreSQL database. We have opted to go with a managed database, for simplicity and to delegate reliability issues to DigitalOcean. A single vertically scaled database will be able to handle many concurrent users. If we needed to scale up substantially, we would likely have to switch to a specialized distributed database system like Kafka, which the real X uses[5].

Terraform

For provisioning we use Terraform, which has been chosen specifically for it being declarative. The Terraform scripts are run using the 'bootstrap.sh' script, which takes a single parameter; 'production' or 'staging'. This in turn provisions our entire system (except the database) and pushes the corresponding docker image to the Docker Swarm. Terraform integrates well with Docker Swarm, has great scalability and automation capabilities. All of this makes it more suitable for our project as it evolved, compared to vagrant, that we had used under the development of the application.

Docker and Docker Swarm

The application is containerized with Docker and deployed using Docker Swarm, set up with 1 leader node, 2 manager nodes, and 1 worker node. Since we initially containerized our system through Docker, it was the natural choice to use a Docker Swarm for distributing our system. We initially chose Docker because it was presented in class, moreover it is a popular and well documented tool for containerizing systems.

Monitoring and logging

We use Prometheus for monitoring and DigitalOcean for infrastructure monitoring, it was introduced in exercises, and has been kept since. We use Loki for log aggregation, and Grafana for data visualization. Loki is inspired by Prometheus and chosen for that same reason[2].

1.4 Current state of the systems

The application is currently shutdown due to the limited amount of credits at DigitalOcean. Before this was done the application, had just been deployed with Terraform, implementing the functionality of the docker-swarm cluster. The application was fully functional on the last days of the simulator running, after this deployment.

We have implemented different tools that can track the quality of our code including CodeClimate and SonarCloud.

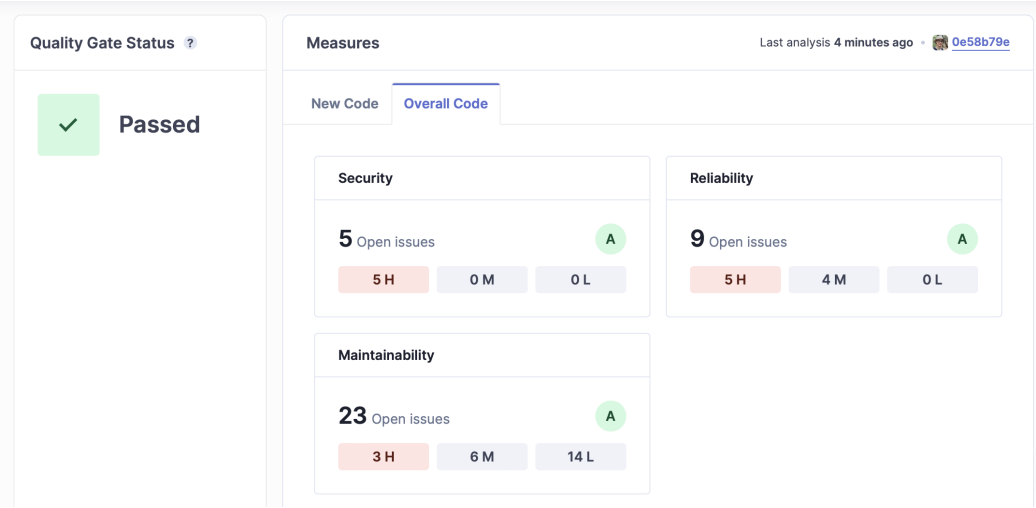

Figure 5: Screenshot of the latest SonarCloud Quality Gate

Figure 5 shows a screenshot of the latest quality gate state in SonarCloud. It displays, that there remains some open issues. All of them have rating A, meaning that it has the lowest score when it comes to improving the software. [1] When it comes to technical debt, we currently have 5%, as seen in Figure 6.

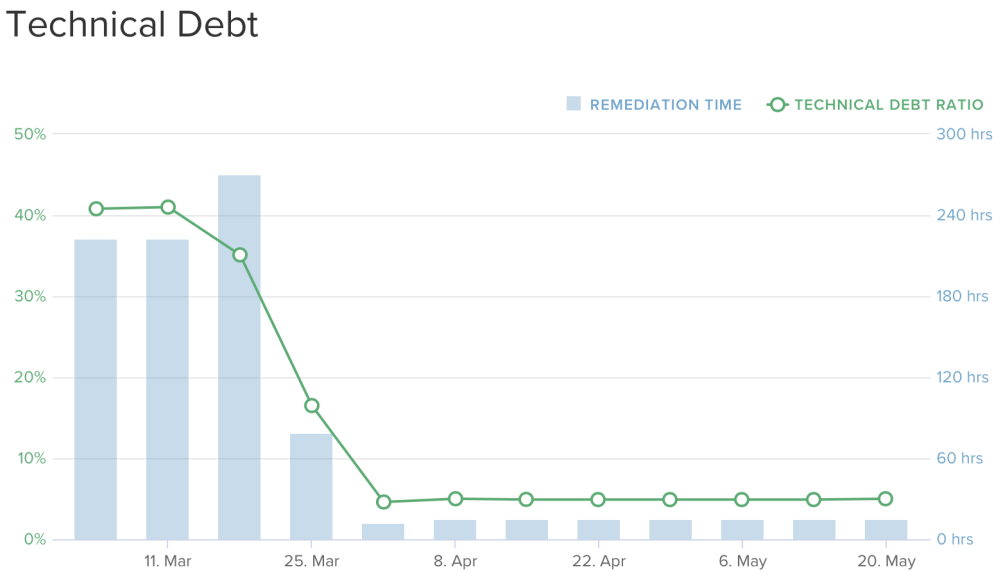

Figure 6: Screenshot of the Technical Debt in our system, from CodeClimate

2 Processes

2.1 Tools and stages included in CI/CD pipelines

Our CI/CD pipelines are implemented using Github Actions. Our workflows are splitted in the following:

quality.yml

Triggered on pull request actions (opened, reopened, edited, synchronize, and ready for review). The main purpose is to run quality checks over the changes existing on the pull request branch. Figure 7 illustrates the stages of the quality workflow.

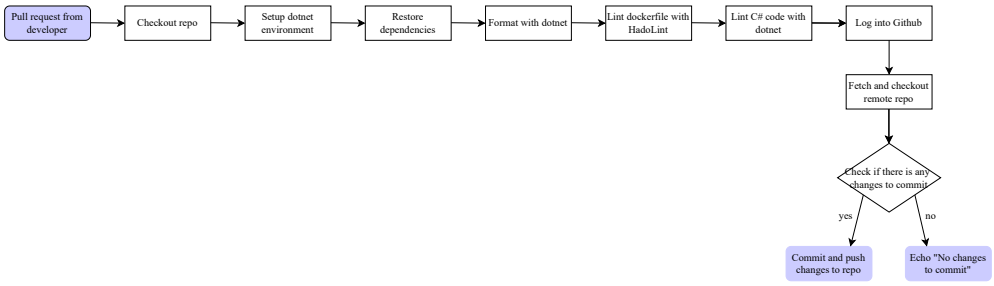

Figure 7: Flowchart illustrating the stages of the quality workflow

Tools used in quality.yml:

- .NET SDK (for running dotnet commands)
- dotnet format (for automatic formatting)
- Hadolint (for Dockerfile linting)
- SonarQube (for code quality and security analysis)

release.yml

Triggered on pushes to `main` branch. It uses a base `release.config.js` file at the root of the repository and it is meant to automate the generation of releases and its changelogs through the use of conventional commit messages. Figure 8 illustrates the stages of the release workflow.

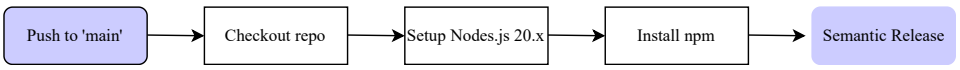

Figure 8: Flowchart illustrating the stages of the release workflow

Tools used in release.yml:

- Node.js (for running npm and semantic-release)
- Semantic Release (for versioning and changelog management)

deploy.yml

Triggered on pushes to `develop` and `main`. It internally checks the targeted environment (`staging` or `prod`) retrieving the corresponding secrets, pushing the application image to Dockerhub and deploying to Digital Ocean. Figure 9 illustrates the stages of the deploy workflow.

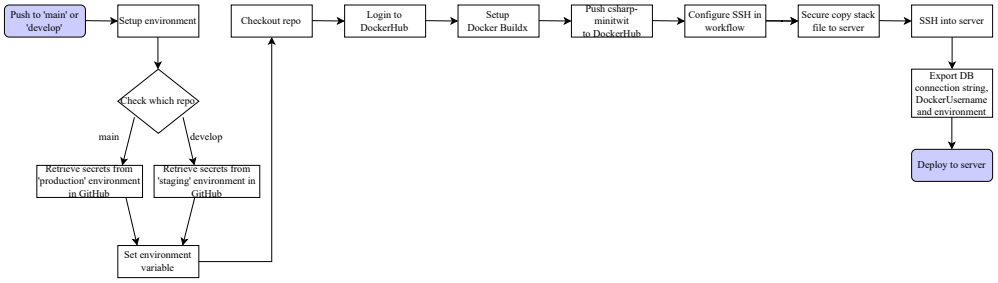

Figure 9: Flowchart illustrating the stages of the deploy workflow

Tools used in deploy.yml:

- Docker (login, build, push actions)
- SSH (for deploying to servers)

2.2 Monitoring

Monitoring is conducted using a self-hosted instance of Grafana on a DigitalOcean Droplet. This instance receives metrics data from Prometheus servers attached to the application droplet. When the Csharp-Minitwit application processes a request that involves a metric method, the OpenTelemetry.Exporter.Prometheus.AspNetCore package collects and exposes these metrics on a */metrics* endpoint. A Prometheus server scrapes this endpoint at regular intervals. Prometheus is then used as a data source for Grafana, allowing us to visualize this information on a dashboard.

Some of the data is also collected by querying the database on some specific indicators, such as:

- Messages registered (application usage)
- Users registered (conversion)
- Follower registrations (users interaction level)

Application monitoring

The data is then used together with the build in packages to display, how the application is performing. We have chosen to monitor the following, as seen in Figure 10.

- Rate of HTTP requests received per endpoint
- Total number of requests (last 24hs)
- Total count of errors per status code (last 24hs).
- Top 10 unhandled exception endpoints.
- Top 10 Requested endpoints (API).

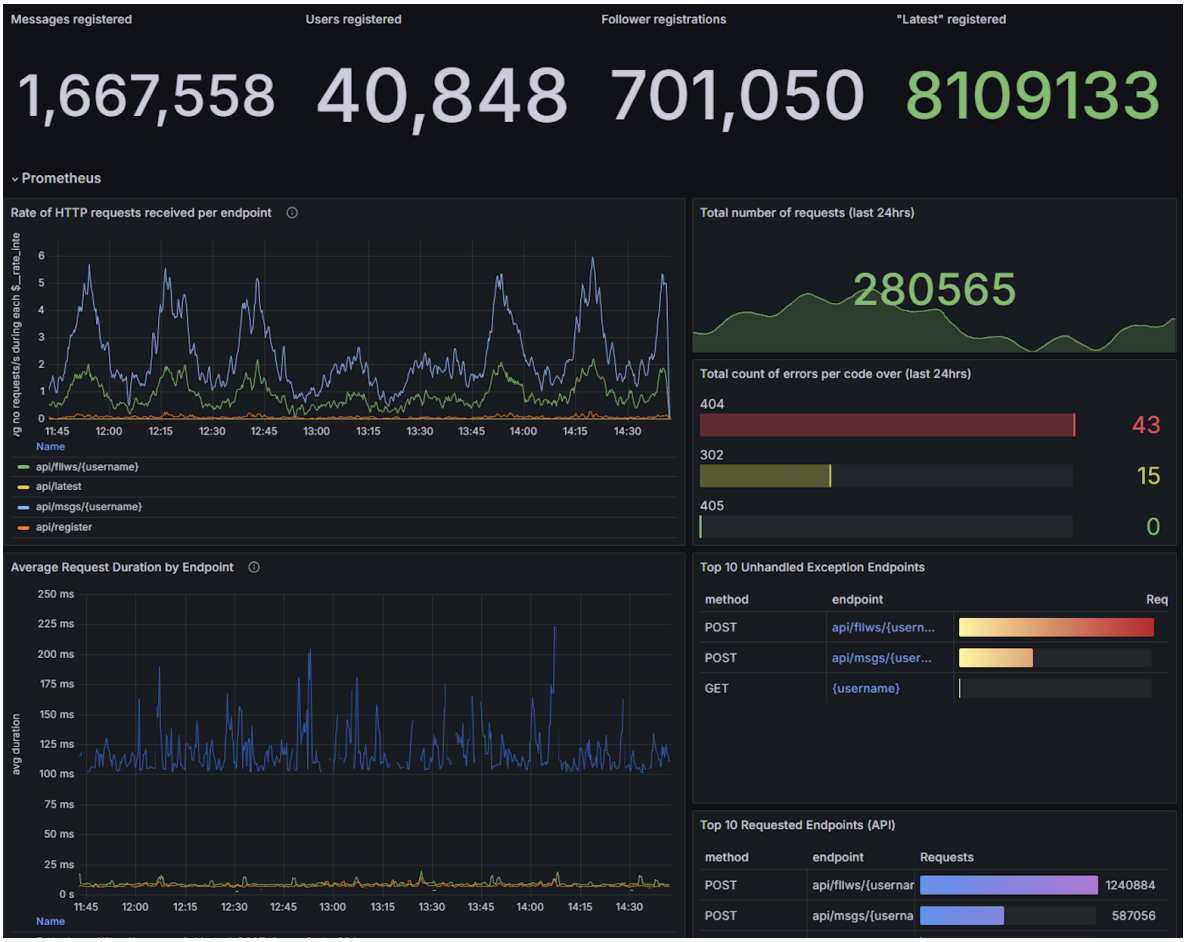

Figure 10: Snapshot of Grafana Monitoring Dashboard

Infrastructure monitoring

Even though we haven't set this monitoring ourselves, Digital Ocean provides some out of the box monitoring for its droplets, such as CPU usage, memory usage, DISK I/O, Disk Usage, Bandwith, etc.

2.3 Logging

Logging is configured through a log aggregation system called Loki, which acts as a Grafana datasource. These logs are pushed directly to Loki by using Serilog, a logging library for .NET applications. Serilog configures a C# API that will receive the Csharp-Minitwit application logs and send them in a POST request to Loki's server. The structured representation of these Logs makes it easier to extract information from them at a later point. Loki can then be configured as a data source in Grafana to process and store these logs to be queried from a Dashboard.

Below, we present the log events throughout our application:

- Message posting
- User logging / Failure logging / Invalid username or password
- User logout
- User registering / Failure registering
- API requests that are not from the simulator
- API registering / Failure registering
- API message posting from users / Failure or incorrect usage of endpoint
- API retrieval of messages for a specific user / Failure in the retrieval
- API retrieval of followers for a specific user / Failure in the retrieval
- API follow/unfollow requests / Failure in the execution

Loki uses a streamlined approach for log aggregation by indexing labels and metadata from each message, then compressing the log message itself. This allows for efficient querying in Grafana based on labels, time ranges, and patterns. For our Csharp-Minitwit application, logs are stored in chunks on a DigitalOcean volume attached to the Grafana droplet.

2.4 Security assessment

Risk identification

We have the following assets:

- Web application
- Database
- CI/CD
- Image repository
- Container orchestration
- Secrets

Threat sources

Web application: Runs on HTTP, making it vulnerable to data leaks and replay attacks. Messages are rendered in HTML, it is possible that this somehow allows for cross-site scripting (XSS).

Database: Usernames and emails are stored in clear-text; passwords are salted and hashed, to defend against rainbow-table attacks. User IDs increment sequentially, which is not best practice. To connect to the database, we use the same user and password for both our staging and production environments.

CI/CD: GitHub actions has access to all secrets. Images are publicly stored on DockerHub. On our staging server, we don't use tokens for logging into DockerHub, instead the credentials are stored in the environment, in clear text. Our codebase is public on GitHub.

Other: We have multiple open ports on our servers, potentially also for the logging endpoint.

Additionally we are using multiple dependencies, which possesses a risk as well, as there could be vulnerabilities in connection to those.

Risk scenarios

Scenario	Probability	Impact	Risk	Strategy / Action
An adversary listens to the non-encrypted data sent on our web application. This is possible because we use HTTP and not HTTPS	5	5	25	Get an SSL-certificate and redirect all traffic to HTTPS
An adversary finds an open port on one of our nodes and gains access to our system	3	5	15	Run Nmap, close unused ports
An adversary constructs a message that escapes HTML and runs code when rendered (XSS)	3	5	15	Sanitize user input
An adversary finds our Docker images online	3	1	3	Make image repository private
An adversary hijacks the cookie of a user	1	2	2	Unavoidable

Unfortunately, we have not had time to implement the above actions. However, we have during the course made some hardening of our system e.g. made a secure connection to the DB and used GitHub secrets to store enviroment variables.

2.5 Scaling and upgrades

We monitor our system to determine when scaling is needed. To scale, we adjust the number of nodes in the Terraform file ‘minitwit_swarm_cluster.tf‘ and run the ‘bootstrap.sh‘ script. This process is quick due to the state being stored in DigitalOcean Spaces Object Storage.

Currently, we lack an automated upgrade process. Based on our experience

upgrading from Python 2 to Python 3, we know upgrades are complex and require careful management of dependencies.

2.6 AI-assistance

AI tools have been part of our development process, with ChatGPT mainly used for debugging and advice. However, due to the unique nature of our application, ChatGPT's resources were limited. We found the documentation for each dependency more thorough and effective for addressing our specific implementation needs.

3 Lessons learned

3.1 Evolution and Refactoring

Changing Infrastructure

One notable incident occurred when we mistakenly configured a workflow to delete the database if it already existed. This resulted in the complete loss of our database and we did not have any backup. We learned the importance of not having destructive operations in our workflows and ensuring that workflows are thoroughly reviewed and tested before deployment and moreover having backups of the database.

Slow Endpoint - Update Indexing

We also encountered performance issues with the /public endpoint of our application, which was notably slow. The root cause was the lack of proper indexing in our database, which had not been configured to align with our application's logic. By adding appropriate indices, we significantly improved the response time of this endpoint.

Issue Link:Look into adding indices to our database #141

3.2 Operation

Bug Fixing through Monitoring

The follow/unfollow endpoint was returning a 200(OK) status code instead of 204(NO CONTENT) as required by the simulator API, resulting in cumulative errors. It was possible to detect, given the information provided by the http://206.81.24.116/status.html status dashboard.

Issue Link:Follow/Unfollow endpoint bug

Performance Improvements through Monitoring

By analyzing metrics, specifically "average request duration by endpoint," and research, we successfully reduced latency across our endpoints. The primary cause of the high latency was the geographical separation between our Database (hosted in the NYC region) and our application server (in the FRA region). We resolved this by co-locating our database and application server in the same region. The improvement is evident in Figure 11:

Issue Link:DB location error

Figure 11: Significant reduction of latency metrics measured in Grafana

3.3 Maintenance

Grafana and Loki Memory Issues

We faced disk space overloading on our Grafana server, where Loki was hosted. At first we tried to resolve this by deleting log files every 12 hours using a cronjob. However, this method was flawed as it deleted files still in use by running processes, resulting in a memory leak. We identified files deleted but still held open by processes using ‘sudo lsof +L1‘. Terminating these processes with ‘sudo kill -HUP [PID]‘ released the disk space (see Figure 12), restored server functionality, and allowed us to regain access to Grafana and SSH. The correct approach was to reconfigure the server to store logs on the attached volume and transfer the existing logs from the disk to this volume from the beginning.

Issue Link: Link for wiki issue week 14

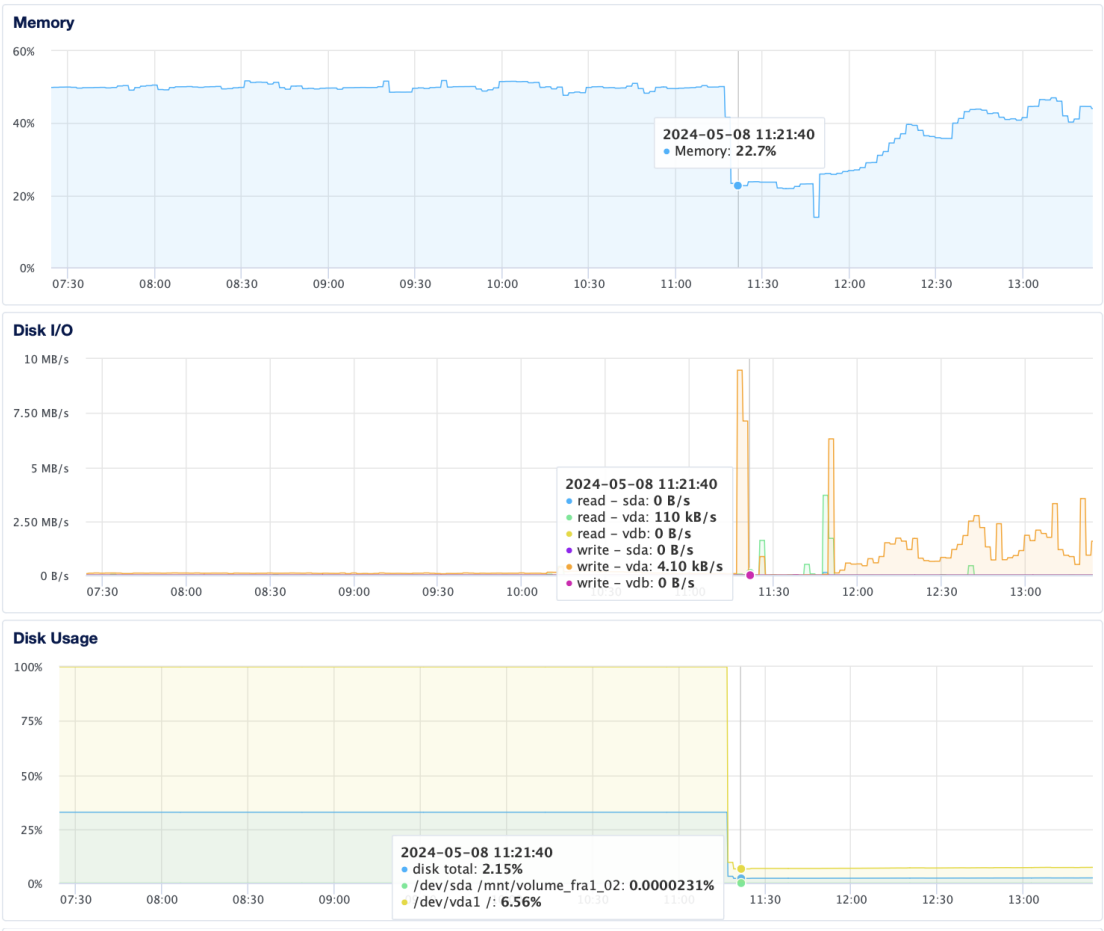

Figure 12: Cleaning up the disk after memory exhaustion

3.4 DevOps Practices Reflection

Our approach to this project was ”DevOps”-oriented compared to previous development projects. Key differences and practices included:

- *Continuous Monitoring and Feedback:* We implemented Grafana and Prometheus, which allowed us to identify and resolve performance issues and bugs.
- *Infrastructure as Code:* We used Terraform for infrastructure management, enabling consistent and repeatable deployments.
- *Collaboration and Communication:* The use of GitHub for issue tracking and commit referencing improved team collaboration.
- *Automated Deployment:* Our CI/CD pipeline, built using GitHub Actions, reducing the risk of human error and speeding up the release process.

These DevOps practices enhanced our ability to maintain and evolve the CSharp-MiniTwit system efficiently and reliably.

References

[1] CodeClimate. *Quality Glossary*. 2024. URL: https://docs.codeclimate.com/docs/code-climate-glossar (visited on 05/21/2024).

[2] Grafana. *Loki repository*. 2024. URL: https://github.com/grafana/loki (visited on 05/19/2024).

[3] Microsoft. *.Net Core support*. 2024. URL: https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core (visited on 05/17/2024).

[4] Microsoft. *Introduction to ASP.NET Core*. 2023. URL: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0 (visited on 05/11/2024).

[5] X. *Processing billions of events in real time at twitter*. 2024. URL: https://blog.x.com/engineering/en_us/topics/infrastructure/2021/processing-billions-of-events-in-real-time-at-twitter- (visited on 05/17/2024).