IT UNIVERSITY OF COPENHAGEN

DevOps Report

IT University of Copenhagen

Course Name: DevOps: Software Evolution and Software Maintenance
Course Code: KSDSESM1KU
Submission Date: 30/05/2025
Group i - CloudMorphers

Authors	**Email**
Alexander Niclas Lerche	alnl@itu.dk
Cem Ergin	ceme@itu.dk
Lucian-Alexandru Aionicesei	laio@itu.dk
Philip Mørch	phimo@itu.dk

Table of contents

1 System Perspective

1.a Design and Architecture

Initially, when we took over MiniTwit, it was a Python-based application built with the Flask framework. We then began replacing it with a new implementation using ASP.NET Core MVC.

This involved a full rewrite of both the frontend and backend, as well as re-implementing all core functionality.

In the early stages, we used Vagrant to provision and configure DigitalOcean droplets, ensuring reproducible virtual machine setups for the database and web server. As the project matured, we transitioned fully to Docker containers because they offer a more lightweight and efficient virtualization approach, making it easier to manage deployments and integrate with CI/CD pipelines.

The Web Application

Our MiniTwit web application is built using ASP.NET Core with the Model-View-Controller (MVC) architecture, combining both frontend and backend logic. Razor Views, together with CSS, provide the UI, while the backend handles requests and data processing, and communication with the database. We use Entity Framework Core (EF Core) for object-relational mapping (ORM), allowing us to define the data model directly in C#. This makes our database design code-first, as the database is created and managed using migrations.

The Database

We chose PostgreSQL for our database, because it is an efficient and open-source database and a popular choice in the industry. Additionally, it has a provider for EF Core called Npgsql[1]. The database instance runs in a Docker container on its own DigitalOcean droplet. It persists data using a Docker volume, and is exposed on port 5432 on the internal network in-between our droplets (more about that in

[1] https://www.npgsql.org/

Section 2.d). A health check ensures readiness in case the database is unavailable, and a Watchtower container keeps the container image up-to-date.

1.b Dependencies (across all levels)

We generated a dependency graph with NDepend, a static analysis tool for .NET projects that helps visualize dependencies and measure quality: dependency-diagram[2].

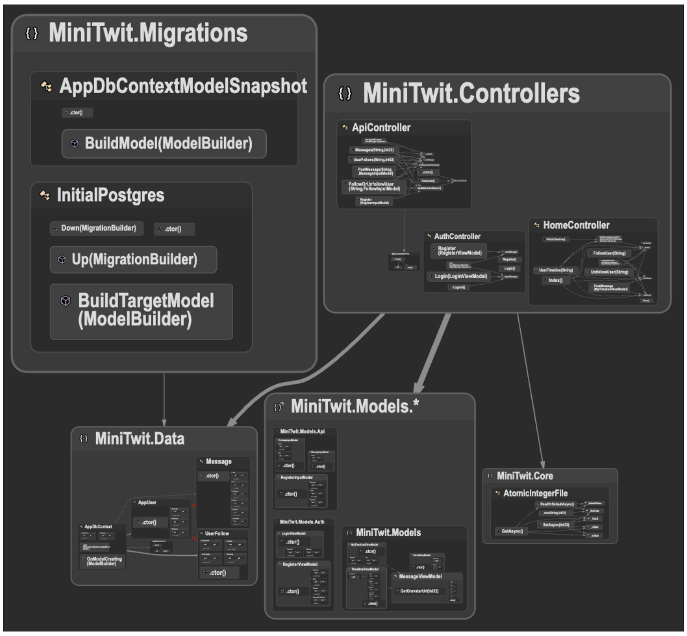


Figure 1: Dependency diagram of the MiniTwit system.

The above Figure 1 visualizes the internal structure of the codebase by showing how the namespaces depend on each other. The architecture is organized into distinct layers: Models, Controllers, Views, Data, Core, and Migrations.

Controllers handle HTTP requests and use Models to represent the business logic data. The Views layer contains the view for the user interface, which makes use of the data classes from Models to visualize the data (e.g. messages). The Data layer is responsible for data access (through the `AppDbContext` class) and contains the

[2] `http://67.207.72.20:8081/NDependReportFiles/ComponentDependenciesDiagram.html`

data model in the form of POCOs (Plain Old CLR Object). Although not explicitly shown in the diagram, communication with the database is done through EF Core.

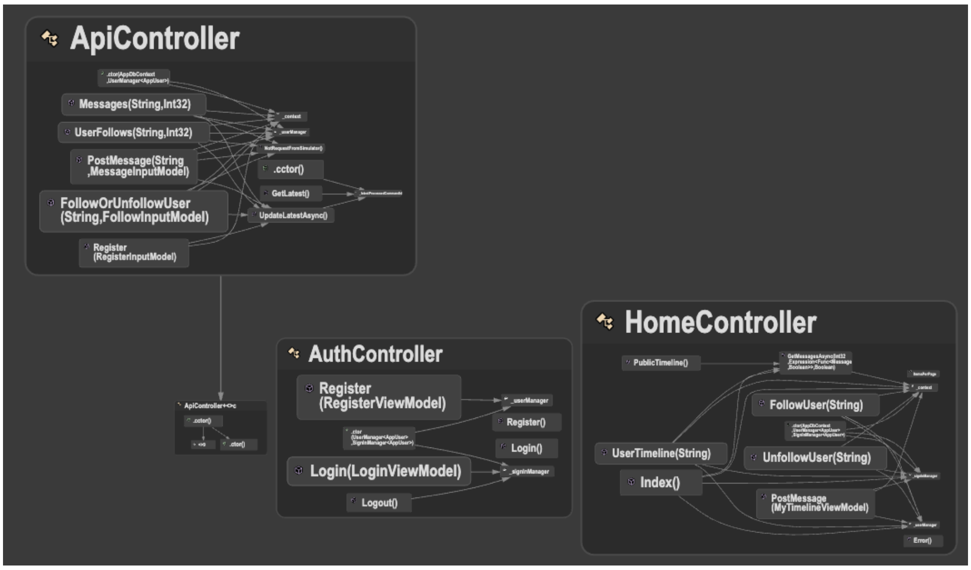

ApiController
.ctor(AppDbContext ,UserManager<AppUser>)
Messages(String,Int32)
UserFollows(String,Int32)
PostMessage(String ,MessageInputModel)
FollowOrUnfollowUser (String,FollowInputModel)
Register (RegisterInputModel)
_context
_userManager
NotRequestFromSimulator()
.cctor()
GetLatest()
UpdateLatestAsync()
ApiController+<>c
.cctor()
.ctor()
AuthController
Register (RegisterViewModel)
.ctor (UserManager<AppUser> ,SignInManager<AppUser>)
Login(LoginViewModel)
Logout()
_userManager
Register()
Login()
_signInManager
HomeController
PublicTimeline()
GetMessagesAsync(Int32 ,Expression<Func<Message ,Boolean>>,Boolean)
itemsPerPage
_context
FollowUser(String)
.ctor(AppDbContext ,UserManager<AppUser> ,SignInManager<AppUser>)
UserTimeline(String)
UnfollowUser(String)
Index()
PostMessage (MyTimelineViewModel)
_signInManager
_userManager
Error()

Figure 2: Dependency diagram of the MiniTwit controllers.

The Controllers layer (Figure 2) is further divided into specific controllers based on functionality: `ApiController` handles API requests, `AuthController` manages authentication and user management, and `HomeController` serves the main web pages. This separation helps organize concerns and simplifies maintenance by clearly delineating different parts of the application interface.

1.c Important Interactions of Subsystems

The system handles two key interaction flows, captured in the provided UML sequence diagrams shown below.

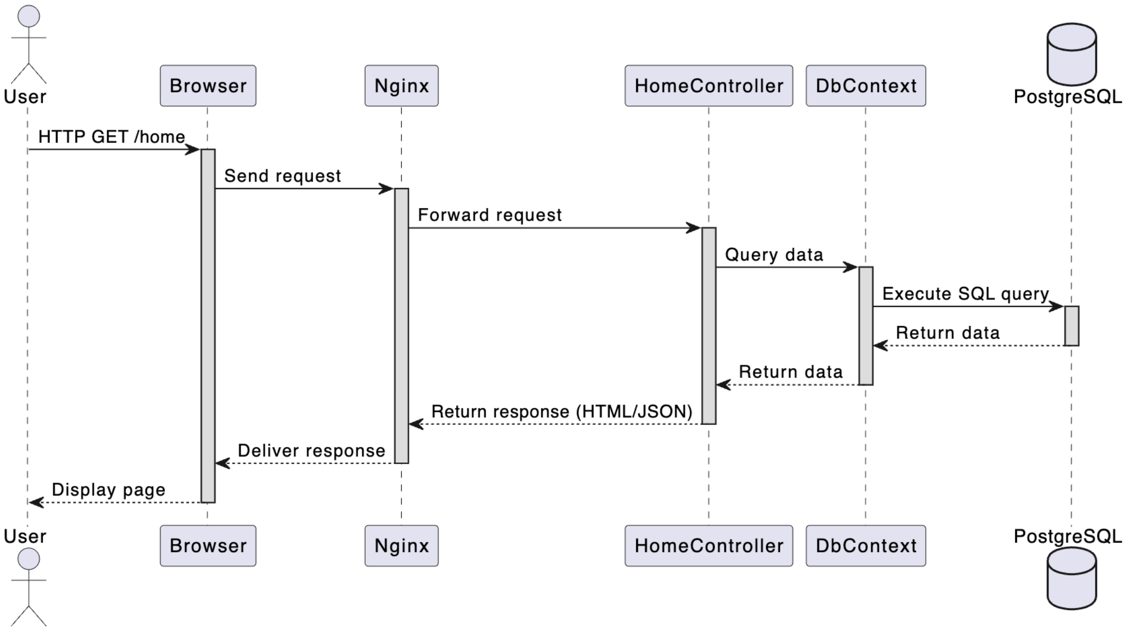


Figure 3: User interaction sequence diagram.

In Figure 3, a user sends a request from the browser (e.g., accessing the home page or registering). This request is routed through Nginx, which forwards it to the `HomeController` or `AuthController` of the ASP.NET Core application. The controller uses the EF Core `AppDbContext` to query the PostgreSQL database. Once the database returns the result, the controller prepares the response, which is passed back through Nginx to the browser, ultimately presenting the result to the user.

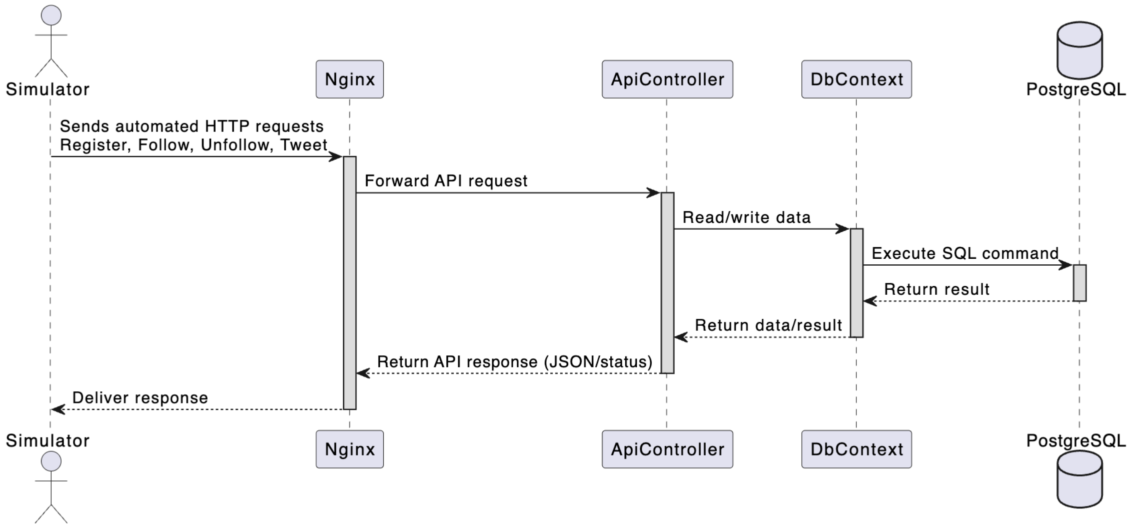


Figure 4: Simulator interaction sequence diagram.

In Figure 4, the simulator sends API requests to the system. These requests also pass through Nginx but are handled by the `ApiController`. The controller also uses `AppDbContext` to query the PostgreSQL database. The system processes the requests from the simulator and returns JSON responses.

1.d Current State

Our ASP.NET Core application is built and deployed through a CI/CD pipeline using GitHub Actions and Docker, automating building, publishing, and containerization. However, static code quality checks are currently non-existent. For future development, we could add a workflow for running compilations and tests. An overview of the deployed system can be seen in Figure 5.

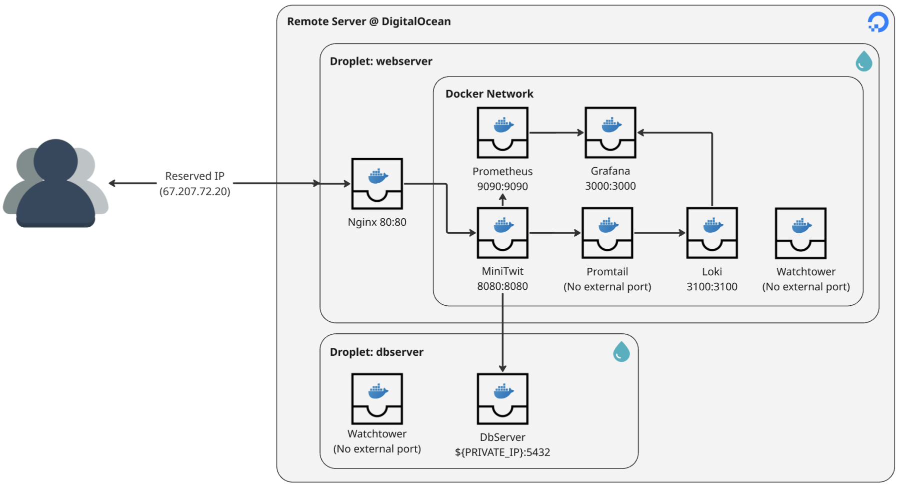
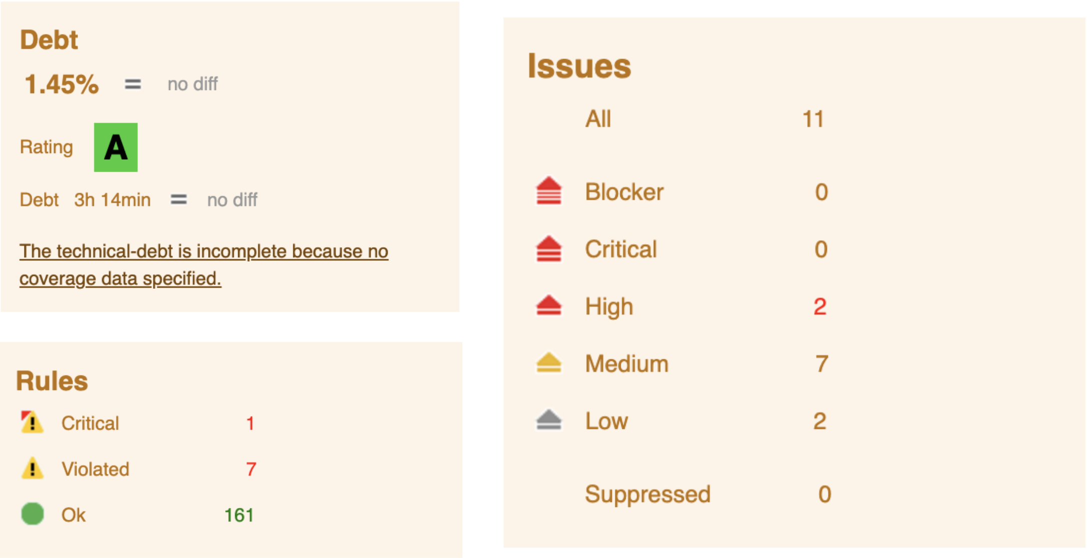


Figure 5: Overview of the current system.

To assess the quality of the codebase, we ran a local static analysis using NDepend, which produced detailed dependency diagrams and metrics on system health. Although not integrated into the CI/CD workflow, the HTML report is hosted on the server for team review: NDependReport.



Debt

1.45% = no diff

Rating A

Debt 3h 14min = no diff

The technical-debt is incomplete because no coverage data specified.

Rules

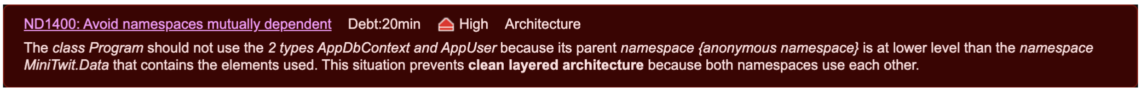
⚠	Critical	1
⚠	Violated	7
●	Ok	161

Issues

	All	11
	Blocker	0
	Critical	0
	High	2
	Medium	7
	Low	2
	Suppressed	0

Figure 6: Overview of NDepend Static Analysis Results

The analysis in Figure 6 indicates low technical debt and overall good architectural structure, though some rule violations (Figure 7) and issues were identified that warrant attention. Notably, the report lacks coverage data because we do not have any tests, which we should have prioritized.



ND1400: Avoid namespaces mutually dependent Debt:20min High Architecture

The *class Program* should not use the *2 types AppDbContext and AppUser* because its parent *namespace {anonymous namespace}* is at lower level than the *namespace MiniTwit.Data* that contains the elements used. This situation prevents **clean layered architecture** because both namespaces use each other.

Figure 7: Example - high priority rule violation

Integrating static analysis and test coverage into the CI/CD workflow will be an important next step to enforce quality gates and prevent regressions.

1.e Justifications for Technology Choices

The choice of ASP.NET Core and C# was partly motivated by the fact that one of the group members is very familiar with .NET and C#, but also because the rest of the group wanted to learn and get familiar with the technology, as it is a popular choice within the industry.

With regards to the particular choice of architecture within the application, we began exploring the use of Razor Pages. However, as we have a lot of shared functionality, e.g. different timelines that all look identical, but where the difference is that it is either scoped to a user, the logged in user (my timeline) or the public timeline, we wanted a more general solution that supported this unified view of a timeline. So, we landed on the use of traditional ASP.NET Core MVC. That way we could have one view for timelines and different controller actions in the backend for the three different kinds of timelines.

We chose to use ORM from the start in the form of EF Core. The main reason for this is because EF Core is tightly integrated into the library we chose for user management called ASP.NET Core Identity[3]. The library offers a superclass called `IdentityDbContext`, which already implements the database model for users, which we inherit from in our own `AppDbContext` class. Additionally, the two classes from

[3] `https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-9.0&tabs=visual-studio`

the identity library we use for authentication, `SignInManager` and `UserManager`, make use of this database context class. So, using SQL directly from the beginning would have been an unnecessary challenge with no benefit, even though it would have been nice to have our software evolve by making the switch to an ORM throughout the development process.

2 Process Perspective

2.a CI/CD Pipeline

Our ASP.NET Core web application is built and deployed using a CI/CD pipeline that integrates GitHub Actions, Docker Hub, and DigitalOcean. When code is pushed to the main branch, the workflow compiles the code, builds a Docker image of the compiled application, and pushes it to Docker Hub. In addition, two separate workflows handle compiling the project report and one that creates GitHub releases (see Figure 8).

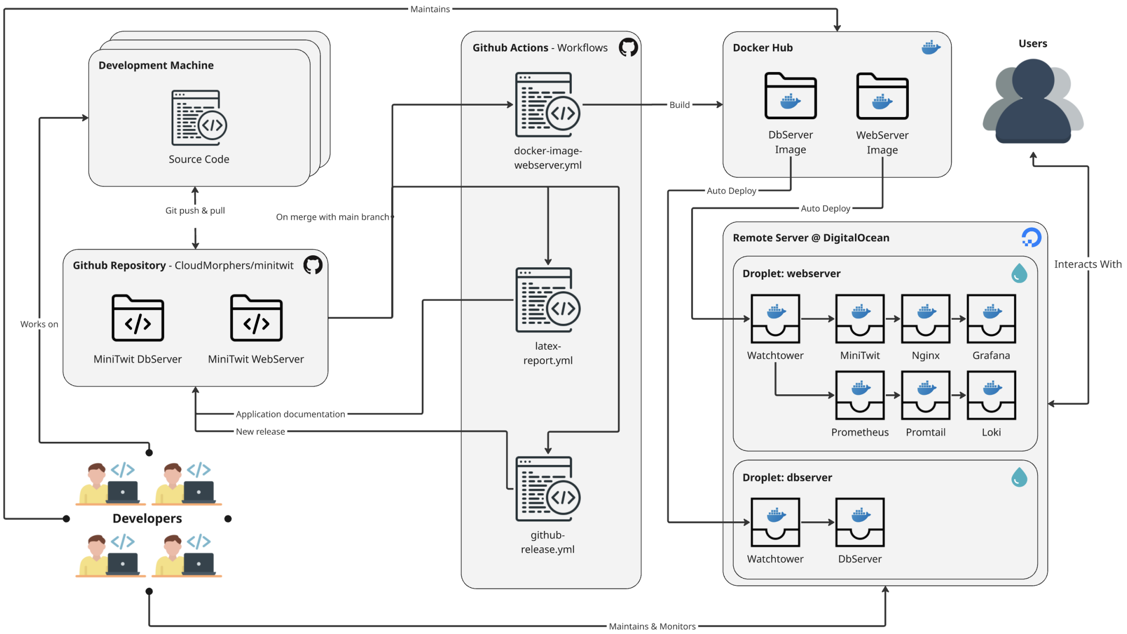


Figure 8: CI/CD Pipeline

On our DigitalOcean droplets, we use a set of Docker containers, including MiniTwit, Nginx, Grafana, Prometheus, Promtail, Loki, and Watchtower, which are launched and managed via docker-compose. Once running, Watchtower monitors for updated images and automatically deploys the latest versions of the containers specified in the compose file.

We chose GitHub Actions for our CI/CD pipeline because it integrates seamlessly with our GitHub repository. Compared to alternatives like GitLab CI or Jenkins, GitHub Actions was the most accessible option for our team, allowing us to set up workflows quickly.

While the CI/CD pipeline is largely automated, the initialization and configuration of the containers via docker-compose, environment variables and config files still require manual SSH access to the server. Furthermore, automated testing has not yet been fully integrated into the pipeline and remains an area for future improvement.

2.b Monitoring

We monitor our system using Prometheus and Grafana. The prometheus-net.AspNetCore package exposes application metrics at the `/metrics` endpoint, which Prometheus scrapes every 5 seconds to provide real-time insights into system behavior and performance.

In Grafana[4] we visualize metrics like the rate of incoming HTTP requests, helping us track the volume of API calls and their return code, in order to detect anomalies such as traffic spikes or increases in 404 errors for timely issue identification. The Grafana dashboard can be accessed with password: `Q9meuDXeCMi5G2QosWiR` and username: `admin`.

2.c Logging

In our Grafana dashboard, we log key HTTP request and response details, including request paths, status codes, and execution times. Additionally, we inject custom logs

[4] `http://67.207.72.20:3000/d/feh7hq21y4g00a/api-endpoints?orgId=1&from=now-20d&to=now&timezone=browser`

inside specific controller methods (e.g. in `ApiController`) to capture important actions like follow/unfollow operations, along with relevant parameters.

We use Serilog to handle logging, writing logs both to the console and to local log files under `/var/log/minitwit/api/` with daily rolling. To aggregate and centralize these logs, we use Promtail to collect logs from the file system and forward them to Loki, which stores them and makes them queryable in Grafana. This setup allows us to visualize and filter logs directly within Grafana dashboards, making it easy to monitor HTTP activity, detect errors, and trace specific application events across the system.

2.d Security

In terms of security, we have set up a firewall that blocks any incoming traffic on ports other than SSH (22) and ports that our Docker containers are set up to expose. We have a droplet for the website and a droplet for the PostgreSQL database. The database only needs to be accessed by the web application, and thus it is unnecessary to expose it to the internet. Therefore, we opted to only have the database exposed on the internal network between our droplets, also called the Virtual Private Cloud (VPC) network. This ensures that the database container will only use the private IP in the VPC network of the droplet.

Another form of security hardening we have done is to use Nginx as a reverse proxy to the web application. Although the Kestrel web server of ASP.NET Core is capable of being directly exposed to the internet, it is generally considered good practice to have a reverse proxy in front. This is because Nginx is very widely used and thus can be considered more secure to have as the public-facing application (because one can assume it is more field-tested). Furthermore, it conceals the fact that we are using ASP.NET Core, as the responses being sent out to users contain Nginx headers instead.

As a final note, we did not have time to set up Transport Layer Security (TLS) or HTTPS, so all traffic going to and from clients and the web server is unencrypted. This is an issue, because if there is an adversary listening to the traffic in-between the client and the server, they will be able to record the data contained within the

HTTP requests/responses, or even replace the contents to perform a Man In The Middle (MITM) attack.

To secure our website, we should have set up TLS by getting a domain name and requesting a free certificate from Let's Encrypt[5]. This can simply be done inside our Nginx reverse proxy, by setting it up to listen on port 443 and use the certificate. Alternatively, if we were to remove Nginx, the ASP.NET Core application can also be configured to use HTTPS.

2.e Scaling and Upgrades

We did not manage to implement horizontal scaling in our deployment. But, we planned to use an architecture using Docker Swarm, coordinating traffic across multiple instances of our application. Our design involved deploying containers for Nginx and MiniTwit, which can be replicated using Docker Swarm. The intended setup includes 2 to 3 manager nodes responsible for orchestrating the cluster, each connected to 3 worker nodes where the MiniTwit and Nginx containers would run. This approach would allow us to scale services horizontally by increasing the number of replicas of worker nodes, which would also reduce or prevent downtime.

2.f Use of AI Assistants

Throughout the development of our system, we made use of OpenAI's ChatGPT and GitHub Copilot to support our learning and implementation. These AI tools were particularly helpful for explaining unfamiliar concepts, such as ASP.NET Core architecture, EF Core behavior, and Docker configuration. We found that LLMs were most effective when used on isolated sections of code, while they struggled with broader context. This was especially true when dealing with interactions across multiple components, such as Docker container networking. Additionally, the AI sometimes generated outdated or deprecated code examples, which required us to cross-check with official documentation.

[5] `https://letsencrypt.org/`

3 Reflections

3.a Biggest Challenges

Database Migration

Midway through the project, we migrated from SQLite to PostgreSQL. To do this, we first tried to do a database dump using `sqlite3 App.db .dump > dump.sql` and then executing the resulting SQL file into our PostgreSQL database on the Droplet using `psql minitwit_db < dump.sql`. This did not work because there were statements in the SQL file that are not supported in PostgreSQL. So instead, we wrote some code in the main function of the web application, which would manually retrieve the data from the SQLite database using the ORM and insert it into the ORM for the PostgreSQL database. This proved to be super easy to do and worked without issue. We did a test on a local machine first to ensure it would work, to have as little downtime as possible (we only had around 5 minutes of downtime during the migration). After the migration we had to delete all the SQLite database migrations and create a new one for the PostgreSQL database.

Local Development - Docker Containers

During the implementation of Docker containers for our MiniTwit application, we gradually transitioned to using containers for local testing to ensure compatibility with monitoring tools like Prometheus and Grafana. However, we encountered frequent issues, as the main application could not start without a connected database. Much of the development happened in parallel, and one group member resolved the issue by setting up a local database and configuring the necessary environment variables. Unfortunately, this solution was initially treated as a local fix and was not immediately shared with the rest of the team, resulting in lost development time. This experience taught us the importance of clear communication and proactively sharing problems and solutions within the team.

Promtail Read Overload

Toward the end of the simulation, we noticed that part of our monitoring setup had stopped working, specifically, the Loki container was no longer running. We restarted the containers, and initially, everything appeared to function as expected. Unfortunately, shortly after, the MiniTwit application crashed, and we lost the ability to SSH into the remote server and had to restart the droplet.

The system slowdown resembled a resource exhaustion issue. After some research, we discovered similar cases where Promtail caused performance degradation due to the absence of a read limit on log ingestion. This matched our situation, as the system had accumulated a large volume of logs over time, and Promtail was attempting to process them all at once, overloading the server in the process. This taught us the importance of setting resource limits policies to prevent monitoring tools like Promtail from overloading the system.

3.b Reflections

Compared to previous projects, we adopted a more DevOps-oriented approach by integrating CI/CD pipelines, automated deployments, containerization, and infrastructure management with Vagrant. Our team collaborated using GitHub by creating issues, git branches, and making pull requests that required peer reviews before merging. Using GitHub issues made us think ahead and plan our development. Despite the challenges of learning ASP.NET Core and Docker, it paid off by giving us valuable insights into new technologies.