IT UNIVERSITY OF COPENHAGEN

Final Report: The Pentuple MiniTwit

Master of Science in Computer Science
IT-University of Copenhagen

Course Name: DevOps, Software Evolution and Software Maintenance
Course Code: KSDSESM1KU
Submission Date: July 7, 2025

Author	**Email**
Adam Hadou Temsamani	ahad@itu.dk
Christian Bank Lauridsen	chbl@itu.dk
Jacob Grum	jacg@itu.dk
Rasmus Ole Routh Herskind	rher@itu.dk
Thor Liam Møller Clausen	tcla@itu.dk

Word count: 2416
(excluding front page, table of contents, references, and appendix)

Contents

1 Introduction

This report covers our project of converting a `Python Flask` application called `MiniTwit` into a modern `.NET` application using the latest technologies. In summary, the application allows users to register and post messages, which are then displayed on a timeline for everyone to see.

The project was part of the course *DevOps, Software Evolution and Software Maintenance* course during spring 2025 at the IT University of Copenhagen. The report will cover how various tools, workflows and technologies where implemented and used to help setting up continuous integration (`CI`) and continuous deployment (`CD`) pipelines while keeping a focus on software qualities like maintainability.

2 System's Perspective

This section covers the main modules of the system with their respective technologies and dependencies, as well as how they were deployed.

2.1 Module view

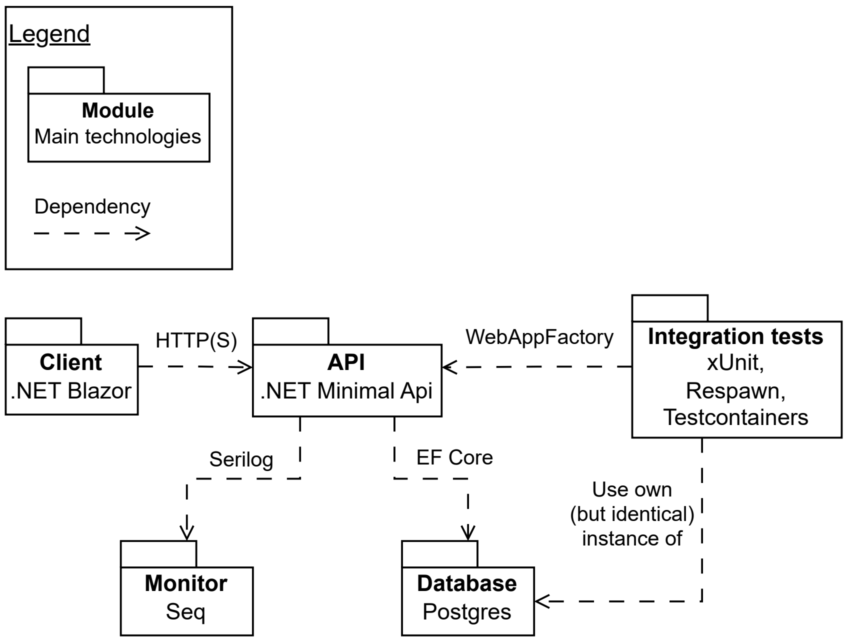

Figure 1: Diagram showing the modules that make up the system with their dependencies and main technologies.

Figure 1 provides an overview of how the main modules are connected, specifically:

- Users interact with the `Client` module
- The `Client` module sends requests to the `API` module which contains a `PostgreSQL` database connection
- The `Monitor` module reads logs written by the `API`
- The `Integration tests` module uses services from the `API` and its own database instance to test the `API`

2.2 Deployment view

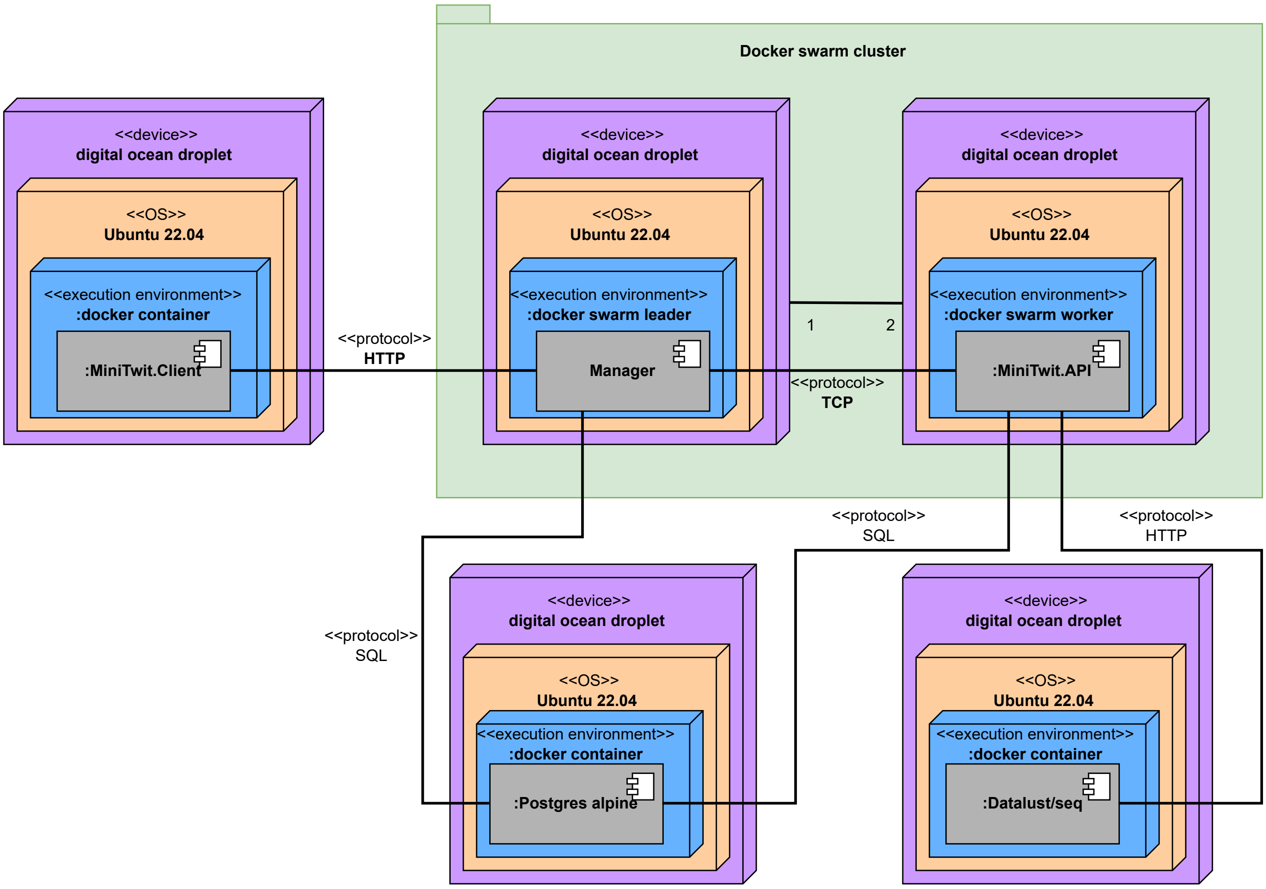

Figure 2: Deployment diagram showing how the modules were deployed.

We use `Digital Ocean` to deploy the modules to `droplets` running `Ubuntu 22.04` as illustrated in Figure 2.

2.3 `API`

The `API` functions as the backend for our `MiniTwit` application. It is deployed as three separate units to form a `Docker Swarm`. One is the `leader/manager` giving tasks to the two others, who act as `workers`. Thus, the `leader` functions as a `load balancer` using the `workers` as servers. We decided that the `leader` should not be a `worker`, because we wanted it to function optimally as a `load balancer`. We were not sure how it would be affected by also acting as a `worker`. Instead, we can use it to run other side tasks, like `database migrations`.

The API project is implemented in .NET using Minimal API[6]. The database communication is done using Entity Framework Core. It logs using Serilog[1], which is configured to write to Seq[4].

2.4 Client

The Client is the frontend for the MiniTwit application implemented as a Blazor WebAssembly app. It is responsible for sending requests and receiving responses from the API module.

2.5 Monitor

Our Monitor uses the tool Seq[4]. It displays custom graphs based on log information. We have both developer-relevant graphs such as API response times and errors, as well as business-relevant graphs such as number of newly registered users and messages posted.

2.6 Database

The database is a PostgreSQL[8] database. It contains information about registered users, posted messages, and followers.

2.7 Sequence diagram

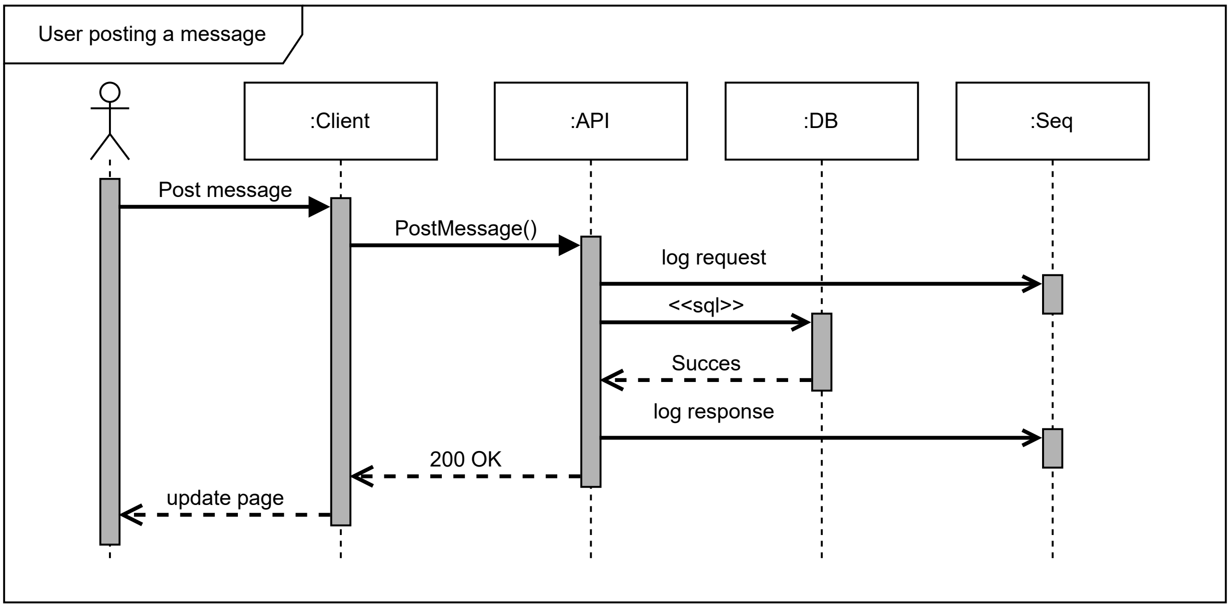

Figure 3: Sequence diagram showing how the system acts upon a user successfully posting a message.

In summary, users post messages from the `Client`. The `Client` sends requests to the `API`, which logs the request information, saves the messages to the database, logs the response information, and sends responses back to the `Client`. Logs are used for monitoring in `Seq`. The `Client` then updates the `UI` for the users. This is illustrated in Figure 3.

3 Process' Perspective

3.1 CI/CD pipeline using GitHub actions

Our overall workflow is shown in Figure 4.

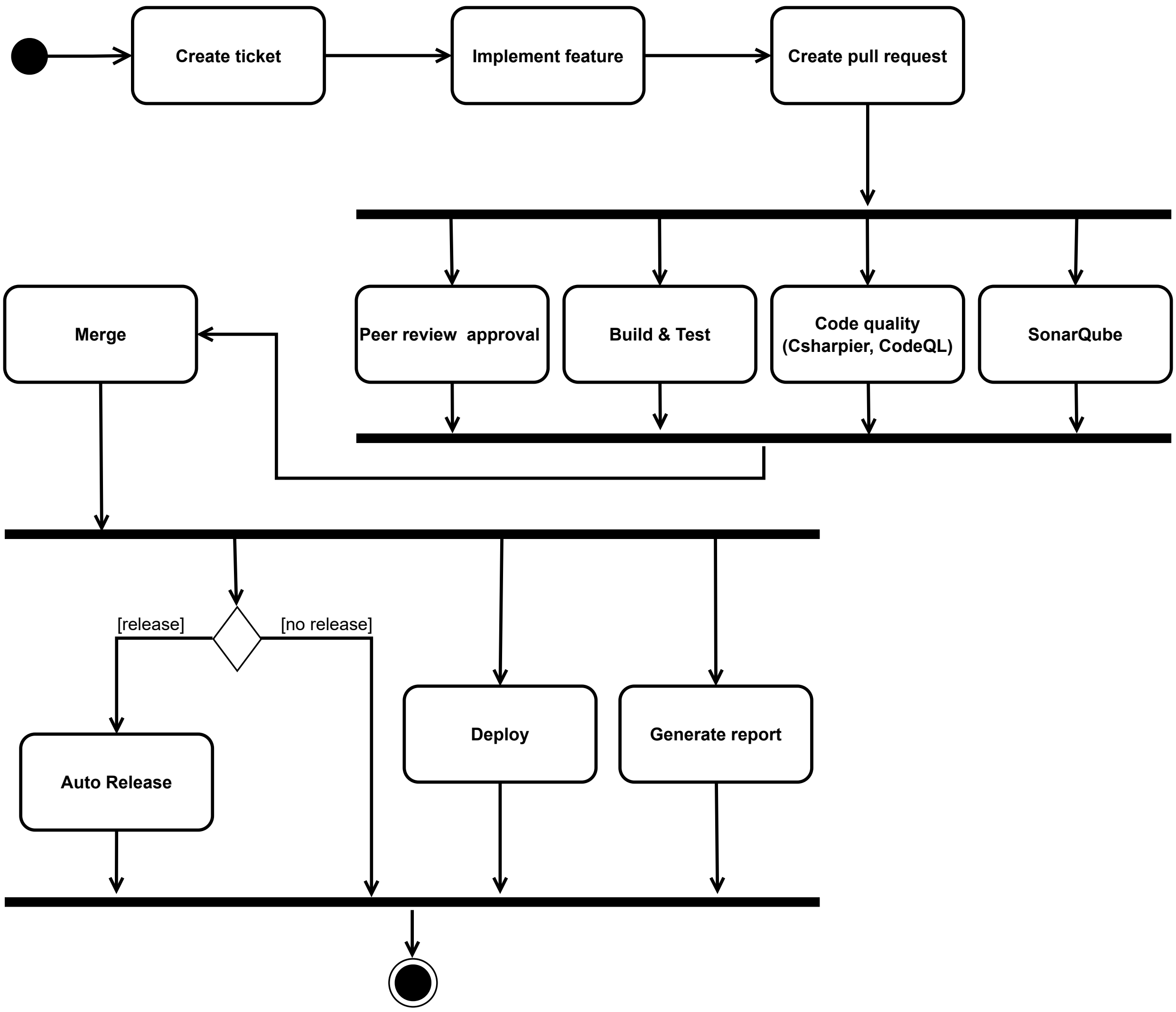

Figure 4: Activity diagram of the CI/CD pipeline

We use `GitHub Issues` to create tickets. The `CI/CD` pipeline is implemented using `GitHub Actions` to automate building, testing, code quality analysis, releasing, and deployment of the `MiniTwit` application.

3.1.1 Build and Test Workflow

This workflow is triggered on every pull request (PR) to main. The purpose of this is to build the application, run tests, and generate a code coverage report. Passing this workflow is a requirement for merging PR's into main.

3.1.2 Code Quality Workflow

This workflow makes use of GitHub's CodeQL[5] to perform static code analysis of C# code. It uses the same triggers as the build and test workflow. Before performing the analysis, it formats the C# code using csharpier[2]. It then performs a semantic code analysis to find security vulnerabilities and displays the results on the PR's. It once discovered that we leaked a third-party secret in our code, which we then removed and updated.

Additionally, we use SonarQube[11], CodeFactor[3], and OpenSSF[7] to perform code analysis such as code coverage, duplication, security risks, and other good practices. The results are displayed on the front page of the repository on GitHub as badges.

3.1.3 Auto Merge Dependabot Workflow

Our project uses dependabot to automatically update dependencies in the project. We have a GitHub Action that detects PR's created by dependabot, and automatically merges them into main if all workflows pass. This ensures that we have the latest dependencies, and we avoid having to manually merge dependabot PR's.

3.1.4 Auto Release Workflow

This workflow is triggered on every push to the main branch, and automatically creates a new release on GitHub if it has a commit message in the format `Release: x.y`. Here x and y are the major and minor version numbers, respectively. Release information such as title and updates, should be added to the file CHANGELOG.md.

3.1.5 Deployment Workflow

This workflow is triggered on a push to main. It is responsible for building Docker images, uploading, and deploying them to Digital Ocean droplets. It uses Docker Hub to push the built images. Then it uses scp and ssh to copy remote files to the relevant droplets, which are then executed to deploy the application. For deployment safeguards, SSH keys, IP addresses, and credentials are stored in GitHub Secrets, which are then used in the workflows to access the Digital Ocean droplets and to authenticate with Docker Hub.

3.2 Terraform

`Terraform` is used to provision and manage our infrastructure as code, to automatically set up droplets in `Digital Ocean`. The result of the `Terraform` setup is illustrated in Figure 2. `Terraform` creates all droplets with `SSH` keys and files. It also configures the `Docker Swarm` for the `API`. It then calls the deploy scripts on each droplet. Finally, it assigns the reserved `IP`'s. The `Terraform` state file is shared in the remote `Digital Ocean Spaces`.

3.3 Security assessment

To analyse security risks in our code, we use tools like `GitHub`'s `CodeQL`[5] that integrates with our `CI/CD` pipeline. This provides a display of security vulnerabilities in our code, which is shown on the `PR`'s.

To analyse security risks of `Docker images`, `Docker Scout` has been used to get quick overviews of security risks from our `Docker images`. It scores the images in four levels: `low`, `medium`, `high`, and `critical`. Additionally, it provides recommendations on how to fix the issues.

We use a risk assessment matrix to assess the probability and impact of security risks. These are split into: probability (`unlikely`, `possible`, `likely`) and impact (`negligible`, `moderate`, and `catastrophic`). The result is shown in Table 1.

Table 1: Risk assessment matrix

<table>
<tr><td></td><td></td><td colspan="3">Possibility of risk</td></tr>
<tr><td></td><td></td><td>Unlikely</td><td>Possible</td><td>Likely</td></tr>
<tr><td rowspan="3">Impact of risk</td><td>Catastrophic</td><td>Access to the database</td><td>Leak of secrets & Droplet failure</td><td></td></tr>
<tr><td>Moderate</td><td></td><td>Overloading the API with requests</td><td>Login credentials leaked</td></tr>
<tr><td>Negligible</td><td></td><td></td><td>Access to Seq</td></tr>
</table>

For details about each risk scenario see Appendix B.

3.4 Scaling

The `Client` uses Blazor WebAssembly (`WASM`) for client-side redering, avoiding server-side rendering. The application logic is executed in the browser, and only lightweight `JSON` requests are made to the `API` to fetch data. This results in a heavly initial load for the user, as the entire application is downloaded locally, however afterwards, this setup offers high responsiveness and scalability by addtionally using a `Hybrid Cache`. To overcome the browser caching limitations (e.g. 16 GB and frequently disks reads), we could introduce a `Redis` cache to potentially support infinite and more efficient caching.

The API is scalable as it is stateless and uses Docker Swarm, allowing horizontal scaling of both manager and worker nodes. The database is not currently optimized in terms of scalability, as it is a single instance of PostgreSQL. The database is the potential bottleneck of the system, as it is the only shared resource between the API nodes. To fix this, we could use sharding to distribute the load of the Database across multiple instances. By splitting the database into multiple shards, we can distribute the load of the API across multiple PostgreSQL instances. This could introduce some complexity and overhead, as it would be harder to manage and query the data.

3.5 AI-assistants

In general we use CoPilot to autocomplete code. We used ChatGPT and other chat bots to help with especially bash script files and converting the Python application to C#.

When we were setting up Vagrant (which was later replaced by Terraform), we used ChatGPT to try and debug an error we had with the database migration. However, we could not get it to detect what was wrong, and we ended up solving it ourselves.

4 Reflection Perspective

4.1 Biggest issues

These are some of the problems we spent the most time solving.

4.1.1 Migrations and Docker Compose

Initially, after we had refactored `MiniTwit` to `.NET`, we ran the program with `dotnet run`. When the `API` started up it would also perform a database migration, in case the model had changed. This is a part of `EF Core` that makes it easy to evolve the database schema. However, when we started using `Docker Compose`, it would not run the migration. We spent a lot of time trying to make it work. Additionally, we did not have much experience with `Docker`, resulting in us spending over 30 hours total on the ticket "Dockerize application".

The solution was to create a separate `Dockerfile` to run the migration. When we later implemented the `CI/CD` pipeline, the migration became part of the deployment process.

4.1.2 Docker Swarm

Before we implemented `Docker Swarm`, we had a single droplet with a single `Docker Compose` file. It was a challenge to change to several droplets with seperate `Docker Compose` files for each module. With a single `compose` file it was easy to have health checks and dependencies between the modules. Fortunately, the only crucial dependency was the database migration. The migration cannot happen before the database is running. However, with the database continuously running on its own droplet, it is no longer an issue.

Another related issue was where to run the migration. Since the `API` module is the one communicating with the database, we initially included it in the `API Docker Compose` file. However, this was an issue now that the API was running on several worker droplets as part of a `Docker Swarm`. Workers are designed to be able to crash and start up again, meaning it is unpredictable when they run their given services. We want to only run the migration once, when we deploy, to prevent race conditions or similar issues on the database. This was solved by separating the migration into its own docker compose file. This is run once on the leader node, when we deploy. We ended up spending over 14 hours on the ticket "Set up `Docker Swarm`".

4.2 Lesssons learned

4.2.1 "It works on my computer"

We learned the importance of having an environment that is self-contained, so it can run across different platforms. The computers used in the group were a mix of `Mac`, `Linux`, and `Windows`, and the `droplets` were running `Ubuntu`. Using `Docker` environments made it easy to run the same program on different machines without too much trouble.

4.2.2 DevOps vs ClickOps

Automation was a big focus in this project. There were numerous things in this project that would have been bottlenecks, if we had to do them manually every time. Especially the `deployment process` required a lot of steps of copying files and `SSH`'ing into `droplets` to run commands. Not only is this time consuming, it is also prone to errors. Forgetting a single step would cause most of the program to not work. An example of this was when we manually created a `droplet` and forgot to add the group `SSH keys`. We were then unable to access the `droplet` and had to tear it down and create a new one.

4.3 Unresolved issues

There are some issues that we have still not managed to solve.

4.3.1 Domain URL has long load time

For some unknown reason, when going to the `Client` with the `HTTPS` domain name, it loads for several seconds before the page is shown. We did spend some time trying to find the cause, but we could not find anything.

4.3.2 A hidden dependency

The `API` must allow the `Client` to access it by setting the `CORS` (Cross-Origin Resource Sharing) policy. We do this in the `appsettings`. The problem is that it is done manually. When the `Client` changes `IP-address`, we must include that `IP` in the `CORS` policy of the `API`. We could allow all origins to avoid this issue, but optimally we would have liked to dynamically inject it somewhere in the workflow.

4.3.3 Seq limitations

Adding monitoring with `Seq` was easy, since it is made specifically for `.NET`. However, it has some limitations regarding graphs. Specifically, it can only use SQL operations that contain the following operations: `select, where, group by, having, order by, limit`. We wanted to make a business-relevant `graph` testing the 1% rule[12]. In short, group users based on how many posts they have made. In order to combine this data, we only found ways that involve `join` operations, which `Seq` does not support.

Furthermore, we have not found a way to save the graphs that we make. This means, if we tear down the `droplet` running `Seq`, we lose all the custom graphs we have made. To avoid starting completely over, we have saved the `queries`, so they can be manually pasted back in when we run `Seq` again.

4.3.4 File structure

`Terraform` was one of the last things we added to the project. Currently, the `Terraform` files are placed in the "`MiniTwitSolution`" folder next to "`remote_files`" which are the files copied to the `droplets`. If the project were to continue, we would move these things to a deployment folder.

4.4 The DevOps difference

These are the things that made this project different and more DevOps-y than other projects we have worked on.

4.4.1 Continuous integration/deployment

Deployment was a big focus in this project, which is something we are not very used to. The issues we spent the most time on were often related to deployment. However, once the workflow was in place, it was an awesome advantage to be able to push some code, whereafter it was automatically available for everyone on the `URL`, meaning there was no need to run the program locally to see the results.

4.4.2 Monitoring

Monitoring is also something we haven't used much in previous projects. But given how easy it was to add, and the advantages it gave, we will definitely include it in future projects. It was super useful for detecting errors, unexpected behavior, and slow response times, as well as general traffic to see what kind of requests the program receives. This helps us make systematic optimizations. For instance, we considered optmizing our caching to speed up the `API` response times. However, the monitor revealed that the number of writes were greater than expected compared to the number of reads, meaning there would be too much cache invalidation for it to be worth it. At one point we also accidentally created a droplet on with an American server. We could immediately tell from the monitor that the response times were much longer because of this. We quickly changed it back to the European server that the other droplets were running with.

References

[1] Apache. Serilog. https://serilog.net/, 2025. [Online; accessed 26-May-2025].

[2] Bela VanderVoort. Csharpier. https://csharpier.com/, 2025. [Online; accessed 28-May-2025].

[3] CodeFactor. Codefactor. https://www.codefactor.io/, 2025. [Online; accessed 28-May-2025].

[4] Datalust. Seq. https://datalust.co/seq, 2025. [Online; accessed 26-May-2025].

[5] GitHub. Codeql. https://codeql.github.com/, 2025. [Online; accessed 27-May-2025].

[6] Microsoft. Minimal apis quick reference. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-9.0, 2025. [Online; accessed 27-May-2025].

[7] OpenSSF. Openssf. https://openssf.org/, 2025. [Online; accessed 28-May-2025].

[8] PostgreSQL Global Development Group. Postgresql. https://www.postgresql.org/, 2025. [Online; accessed 27-May-2025].

[9] Pulumi. Pulumi. https://www.pulumi.com/docs/iac/languages-sdks/dotnet/, 2025. [Online; accessed 28-May-2025].

[10] Redis. Redis. https://redis.io/, 2025. [Online; accessed 28-May-2025].

[11] SonarSource SA. Sonarqube. https://www.sonarsource.com/sem/products/sonarqube/, 2025. [Online; accessed 28-May-2025].

[12] Wikipedia. 1% rule. https://en.wikipedia.org/wiki/1%25_rule, 2025. [Online; accessed 27-May-2025].

Appendix

A Project repository

https://github.com/Grumlebob/The-Pentuple-MiniTwit

(The repository `README.md` contains links to our `client`, `api`, and `monitoring`).

B Security assessment expanded

This section gives a more detailed description of the security risks of our `MiniTwit` application.

B.1 Access to Seq (negligible, likely)

In the case of resetting/failure of the `Seq droplet`, the logs are lost. Moreover, the `Seq` password is reset every time, meaning we do not have a fail-safe default. This gives malicious users temporary access to our `Seq` dashboards, until we reset the password. However, this is not a big issue, as `Seq` is only used for monitoring and logging, and does not contain any sensitive information. However, only one user (stakeholder or developer) can access `Seq` at a time, which can be abused to deny availability to legitimate users.

B.2 Login credentials leaked (moderate, likely)

Usernames and passwords are stored in plain text in our database (no encryption or hashing applied). If an attack gains read access to the users table, all credentials are immediately comprised, leading to unauthorized account takeover.

B.3 Overloading the API with requests (moderate, possible)

A `DDoS` (distributed denial of service) attack can flood our `API` with excessive traffic, rendering it unavailable to legitimate users. We do have load-balancing and basic firewalls, but there is no further safeguard against `DDoS` attacks. Neither do we enforce `CAPTCHA` or rate-limiting on critical endpoints. The impact is limited to service unavailability.

B.4 Access to the database (catastrophic, unlikely)

If an attack obtains direct database credentials or exploits a vulnerability to gain read/write access. They could delete or modify critical data, or have access to all user data, including usernames and passwords.

B.5 Leak of secrets (catastrophic, possible)

Some of the project secrets are shared informally (e.g., via `Discord`). Thus, if the wrong person is invited, these keys could be exposed. An attacker with a valid key could impersonate our services, or cause damage. The probability is possible until stricter secret-management policies are enforced, such as using a secret manager or vault.

B.6 Droplet failure (catastrophic, possible)

Any `droplet` could fail due to software or hardware issues. We currently have no database backups. If the database `droplet` dies, we lose all data. The impact is catastrophic as we could lose all data (including user accounts and tweets), and the application would be unavailable until the `droplet` is restored or replaced.

C Reasons for tool choices

It is important to state, that our group agreed that job prospects and learning within `.NET` enviroment was a goal. As such, one main motivator was, what is commonly used in the `.NET` industry. As such, choices such as using `.NET Minimal API`, was quite obvious.

C.1 Infrastructure as code

We debated using `Pulumi`[9] vs `Terraform`. `Pulumi` is a newer tool, with a `.NET` SDK, making the language more native to what we are used to. But none of us could find any job postings that used `Pulumi`. So we decided to use `Terraform`, which is more widely used in the industry.

C.2 Blazor vs MVC

We wanted to send smaller `JSON` data instead of full `HTML` pages. `Blazor` is newer and higher performance. By using `Blazor WASM`, the client loads the code upon initial visit, and uses `JSON` data sent by our `Minimal API`, to render the `HTML` pages. This is an alternative to using `ASP.NET MVC`, which would require the server to render full `HTML` pages on site requests.

C.3 GitHub and GitHub actions

Everyone had experience, and found no learning to be gained from switching to example `gitlab` and `Jenkins`.

C.4 Hybrid cache

We noticed the simulator made several reads sequentially without a write, making caching quite benificial. `Hybrid cache` supports both local and distributed caching for example through `Redis`[10]. We planned to do `Redis`, but it is still in the backlog.

C.5 Testcontainers and respawn

As we are refactoring code, testing is insanely useful, so we do not regress. We wanted to do integration testing, to avoid mocking. `Testcointainers` is also directly supported in `Github Actions`, and as such makes contionuous integration a breeze.

C.6 Minimal API vs controllers

We had both at some point. `Minimal API` is newer, more performant, and easy to read and write. Therefore we refactored the code, to only have `Minimal API` endpoints.

C.7 PostgreSQL

`PostgreSQL` is free, and was part of the database course we all took. We found learning about a different databases to be out of scope for this course. As well as going from `sqlite` being a relationship based database, we did not want to switch to for example a document based DB.

C.8 Seq

We wanted to use a log server, and `Seq` is made for `.NET`. It is easy to set up, has a nice UI, and is widely used in the industry.

C.9 Docker Swarm

We had a hard time following the course alongside the other courses at the moment this part of the project was ongoing. Therefore we did not debate any alternatives, and followed the course guide. We only heard `Kubernetes` mentioned in the course, but as being “highly advised against using”.

C.10 DigitalOcean

We are poor students, using 7 taped together calculatores as a computer. Digital Ocean was free with the GitHub student credits, this made it an easy choice.