IT UNIVERSITY OF COPENHAGEN

The Pentuple MiniTwit

Adam Hadou Temsamani (ahad)

Christian Bank Lauridsen (chbl)

Jacob Grum (jacg)

Rasmus Herskind (rher)

Thor Liam Møller Clausen (tcla)

Deadline for submission: May 30, 2025

Word count: XXX

(excluding front page, table of contents, references and appendices)

Contents

1 Introduction

This report covers our project of converting a Python flask application into a modern .NET application using the latest technologies.

The appilcation is called MiniTwit. It is an application where users can register and post messages, which are then displayed on a timeline for all users to see.

The project was part of the ITU DevOps course 2025 and required various tools, workflows, and technologies. The main focus was on continuous integration (CI) and continuous deployment (CD), as well as software qualities like maintainability. This report covers how these things where included.

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

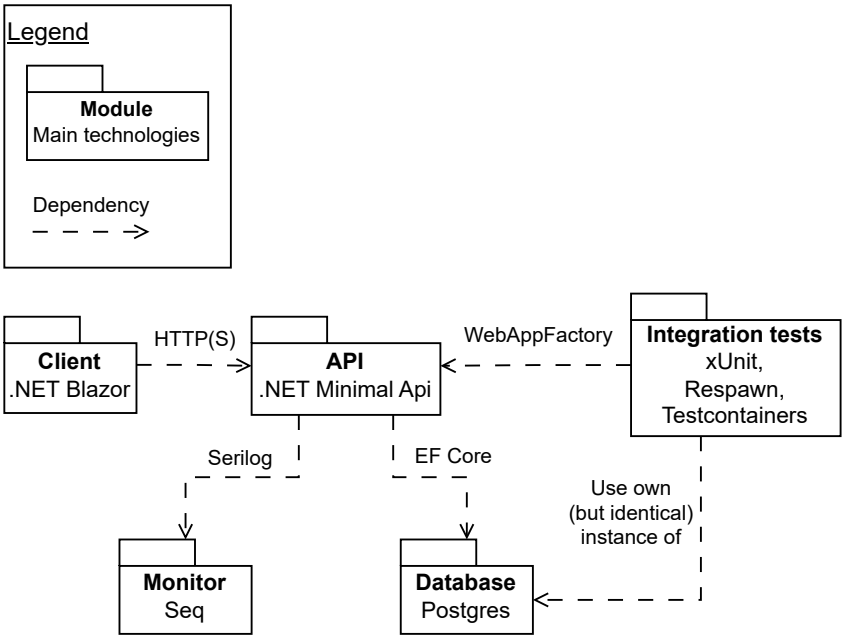

Figure 1: Diagram showing the modules that make up the system with their dependencies and main technologies.

Users interact with the system through the Client module, which send requests to the API module. The API has a database connection and writes logs which are read by the Monitor module. The Integration tests use services from the API and its own instances of the database to test the API module. This is illustrated in Figure 1.

2.2 Deployment view

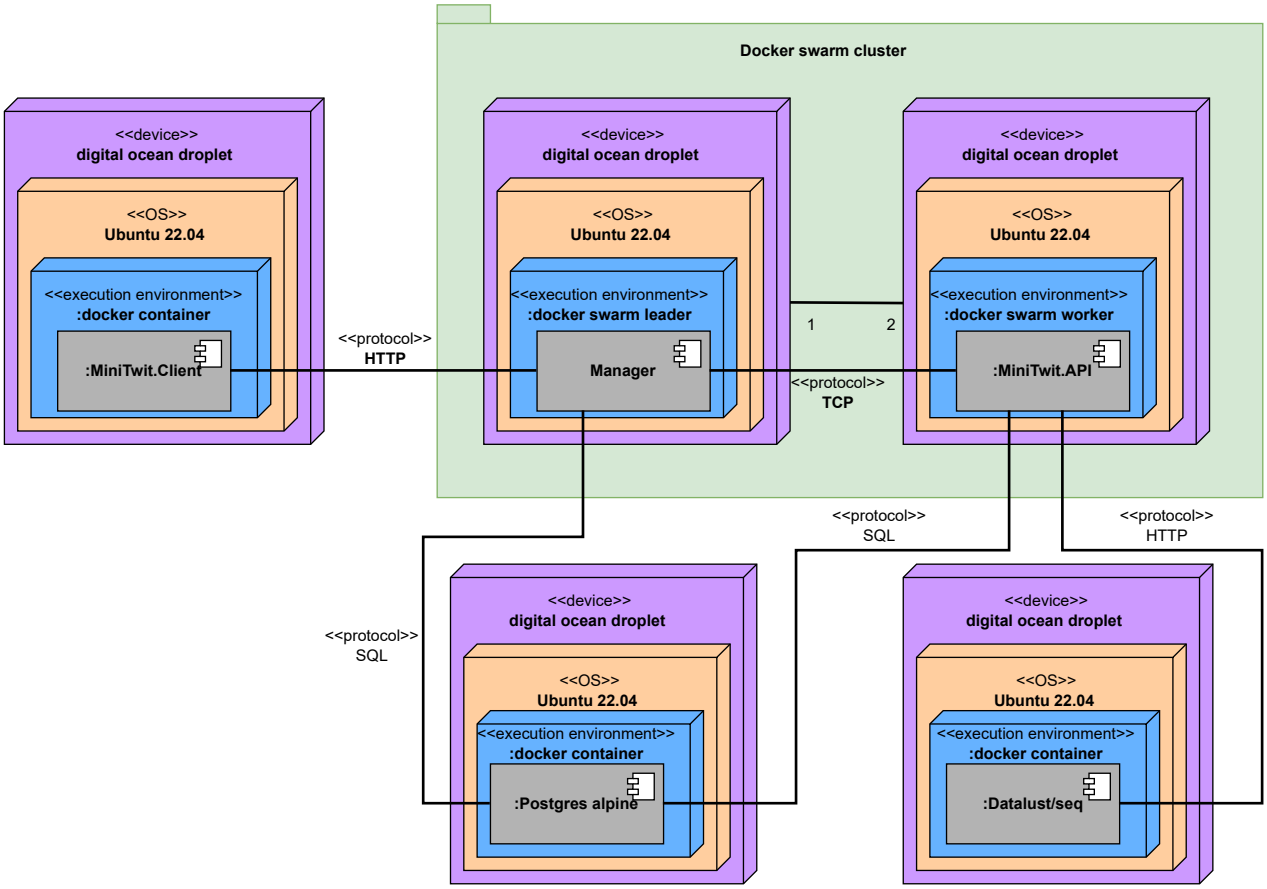

Figure 2: Deployment diagram showing how the modules were deployed.

We used Digital Ocean to deploy our system. The modules were deployed to "droplets" running Ubuntu 22.04. This is illustrated in Figure 2. It worked by creating and pushing docker images that the droplets pulled and ran.

2.3 API

The API functions as a backend for the MiniTwit application. It is deployed as three separate units. It uses docker swarm with one leader/manager giving tasks to two 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 to function optimally as a load balancer. We were not sure, how it would be affected by being another worker. Instead we can use it to run other side tasks, like databse migrations.

The API project is implemented in .NET using minimal API. The database communication is done using Entity Framework Core. It logs using Serilog[1], which is configured to write to Seq[3].

2.4 Client

The Client is the frontend for the MiniTwit application. It is implemented as a Blazor webassembly app. This means the load is distributed to the users' browsers rather than a server. It sends requests to and receives responses from the API module.

2.5 Monitor

Our Monitor uses the tool, Seq[3]. It handles logs and displays custom graphs based on the 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. It is deployed on its own Digital Ocean droplet using an image provided by datalust[3].

2.6 Database

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

2.7 Sequence diagram

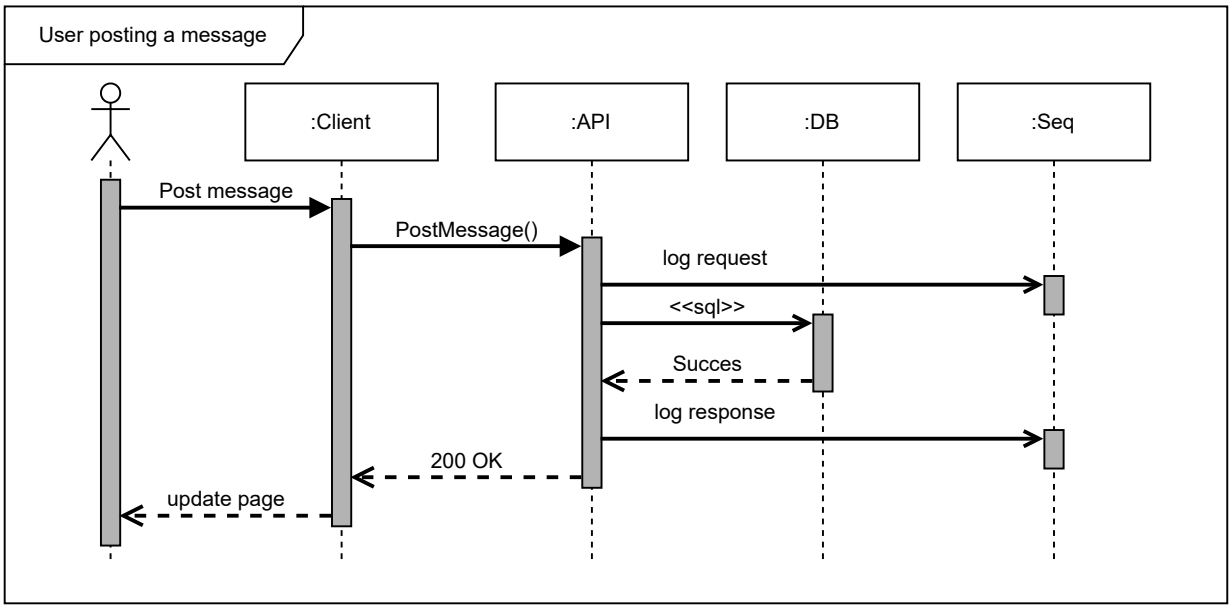

Figure 3: Sequence diagram showing how the system acts upon a user posting a message.

In summary, users uses the Client to for instance post a message. The Client sends a request to the API, which logs the request information, saves the message to the database, logs the reponse information and sends a response back to the Client. The Client then updates the UI for the user. This is illustrated in Figure 3.

3 Process' Perspective

3.1 CI/CD pipeline using GitHub actions

Our CI/CD pipeline is shown in Figure 4, which illustrates the process of creating a ticket about a feature or an issue, developing the feature or fixing the issue, creaating a pull request (PR), reviewing the PR, merging it into the main branch, and finally realeasing and deploying the application.

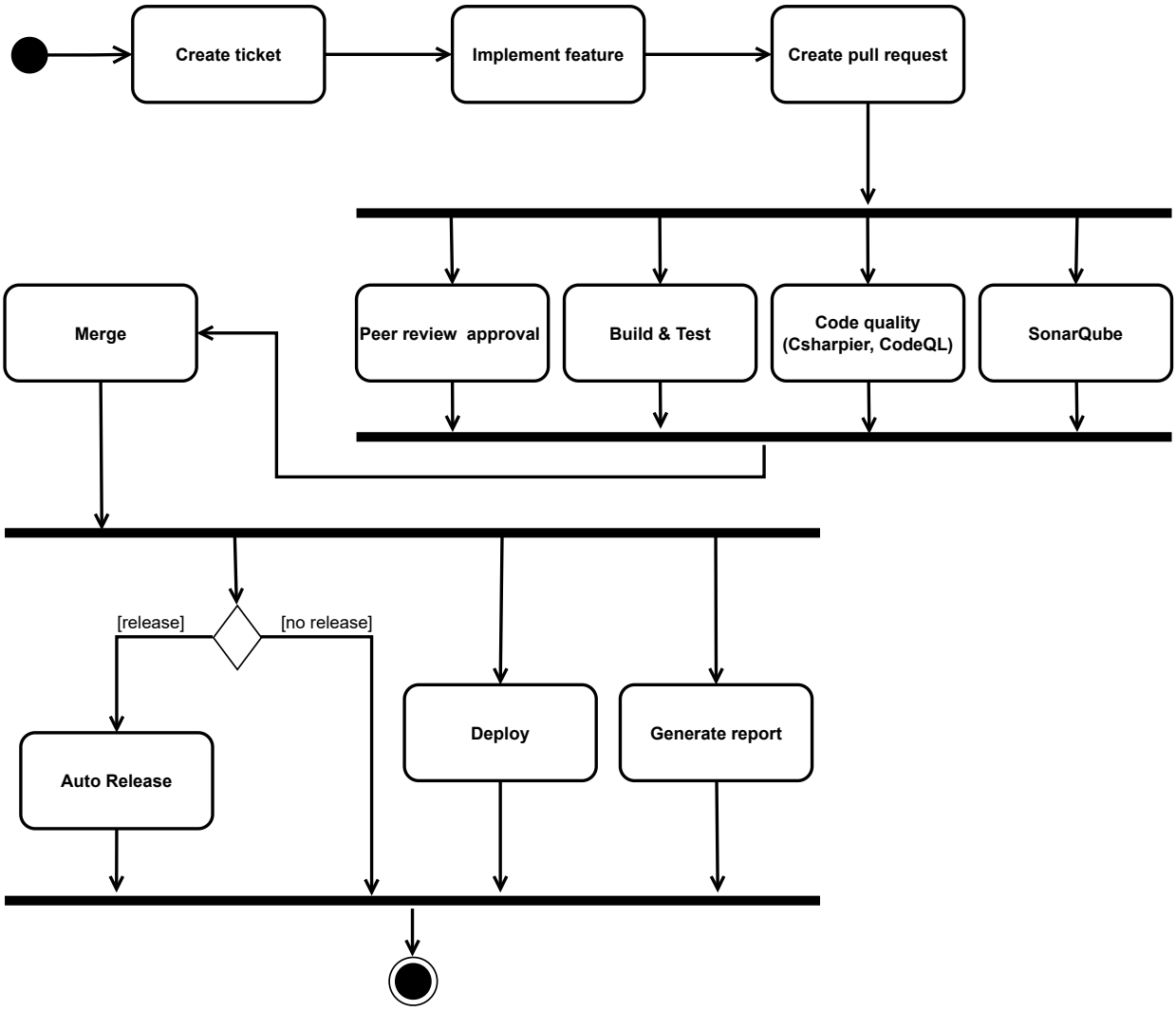

Figure 4: Activity diagram of the CI/CD pipeline

We use GitHub issues to create tickets for new features and issues.

The CI/CD pipeline is implmented using GitHub actions to automate the building, testing, analysis of code quality, releasing, and deployment of the MiniTwit application. The pipeline consists of multiple workflows: build-test.yml, codeql.yml, auto-merge-dependabot.yml, auto-release.yml, and deploy.yml. Workflows are located at *.github/workflows* in the repository.

3.1.1 Build and Test Workflow

This workflow is triggered on every PR to *main*. The purpose of this is to build the application, run unit tests, and generate/upload 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` to perform static code analysis of C# code only. This workflow uses the same triggers as the build and test workflow, however, it is also triggered on a schedule, which is set to run every Thuesday via CRON. Before performing the analysis, it formats the C# code using `csharpier` to ensure consistency and automatically commits and pushes formatting changes. It then performs a sematics code analysis to find security vulnerabilities and automatically uploads the results to GitHub, which is displayed on PR's. This became very relavant when it once discovered that we leaked a third party API key in our code, which we then removed and updated.

Addtionally, we use SonarQube, CodeFactor, and OpenSSF to perform code analysis such as code coverage, duplication, security risks, and other best practices. SonarQube's analysis is done by using the `sonarcloud-github-action` GitHub action, which uploads the results to SonarCloud, and is displayed on the PR. CodeFactor and OpenSSF have read access to our repository, and automatically perform code analysis whenever our repository is updated. The updated results are displayed on the front page of the repository on GitHub as batches.

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 automaticaly merges them into *main* if the build and test workflow passes. This was done to ensure that we always have the latest dependencies, and to avoid having to manually merge dependabot PR's.

3.1.4 Auto Release Workflow

This workflow was inspired by a GitHub actions auto release template[2]. It is triggered on every push to the *main* branch, and automatically creates

a new release on GitHub. It requires a commit containing a message in the format `Release: x.y`, where x and y are the major and minor version numbers, respectively. To add title and release note information, this is added to the file *CHANGELOG.md*. The purpose of this is to automate the release process, to replace the process of manually create a release on GitHub every time we want to release a new version of the application.

3.1.5 Deployment Workflow

This workflow is triggered on succesful completion of the build and test workflow after push to main. It is responsible for building docker images, uploading, and deploying them to Digital Ocean droplets. To deploy, it uses Docker Hub to authenticate and push the built images. It uses `scp` and `ssh` to copy two remote files `deploy.sh` and `docker-compose.yml` to the relevant droplet, which are then executed to deploy the application. For deployment safeguards; SSH keys, IP addresses, and creditials 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 setup droplets in Digital Ocean, which is illustrated in Figure 2. Terraform creates a client droplet, which is connected to a docker swarm cluster. Terraform configures the docker swarm cluster to have one leader/manager droplet which manages two worker droplets that executes the API. Addtionally, it creates a Database droplet, which runs a Postgres database, and a Monitor droplet, which runs Seq for logging and monitoring.

To maintain infrastructure consistency and avoid creating duplicate droplets, the Terraform state file is stored in a remote Digital Ocean Spaces when existing droplets are running.

3.3 Logging and monitoring

Serilog is used for logging API requests, responses, and errors. Together with Seq, we can view the logs in a web interface to monitor applciation activity, and display graphs containing application performance for developers, and business statistics for stakeholders to use. In Seq, we have three dashboards: Overview, Endpoint Overview, and Business. The Overview dashboard contains a general view of the application errors/exceptions, and the total number of events. The Endpoint Overview dashboard contains a view of the API endpoints, showing the number of requests, and average response times. Lastly, the Business dashboard contains a view of user activity, showing when and how often users posts tweets, and the total number of tweets over time.

3.4 Security assessment

To analyse security risks in our code, we use tools like GitHub's CodeQL[**?**] that integrated 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 the droplets, Docker Scout have 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. Addtionally, it provides recommendations on how to fix the issues.

3.5 Scaling

Client is scalable since it is client side rendered. Api is scalable as it is stateless and uses docker swarm.

3.6 AI-assistants

Gpt was of great help in translating python to csharp. Generating files. Not so great at ...?

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

In the beginning of the project 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".

As a solution we ended up creating a separate Dockerfile for the migration. The sole purpose of this file was to use the API code project to run the migration. Later on as we implemented the CI/CD pipeline, the migration was made part of the deployment process.

4.1.2 Docker swarm

Before we implemented docker swarm, we had a single droplet and a single docker compose file that defined all the MiniTwit modules. This made it quite the 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 up and 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. Otherwise, multiple concurrent migrations could cause 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 selfcontained, so it can run across platforms. The computers used in the group were a mix of Mac 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. It has a specific format

for the SQL queries you can make graphs from. Specifically, it can only contain the following operations. `select, where, group by, having, order by, limit`. We wanted to make a business relevant graph testing the 1% rule[4]. 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.

Furthormore, we have not found a way to save the graphs 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 "MiniTwitSoultion" 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 devopsy 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 higher because of this. We quickly changed it back to the European server that the other droplets were running with.

4.5 Reasons for tool choices

.NET, minimal api, Blazor,
Docker
Digital ocean.
Seq, serilog
Terraform
Csharpier
SonarQube
Postgres
dependabot
coveralls
testcontainers
github actions

References

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

[2] CupOfTea696. Auto release github action. https://github.com/marketplace/actions/auto-release, 2021. [Online; accessed 26-May-2025].

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

[4] 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 An Appendix Chapter (Optional)

...