DevOps - Final Report

Group I

Anders Latif (alat@itu.dk)
Ivar Cmrečak (ivcm@itu.dk)
Mads Meinert Andersen (mmea@itu.dk)
Mikkel Rahlff Berggreen (mirb@itu.dk)
Nedas Surkus (nesu@itu.dk)

May 2023

Word count:

Contents

1 Problem formulation

We have inherited a legacy system called Minitwit and have been tasked with refactoring it. Besides modernizing the stack it has been our aim to do so using DevOps methodologies. By reflecting on select few pillars of DevOps the final question becomes whether or not we achieved this goal. Is what we've done DevOps?

2 System's Perspective

2.1 Design and architecture of the system

Minitwit uses a container-based architecture where each service runs in its own Docker container. The backend is built with FastAPI and Jinja. Grafana and Prometheus are utilized for monitoring, the EFK stack for logging, Redis and PostgreSQL serve as database.

The deployment view below is to help illustrate how these components are deployed and interact with each other in a production environment, see figure 2.1.

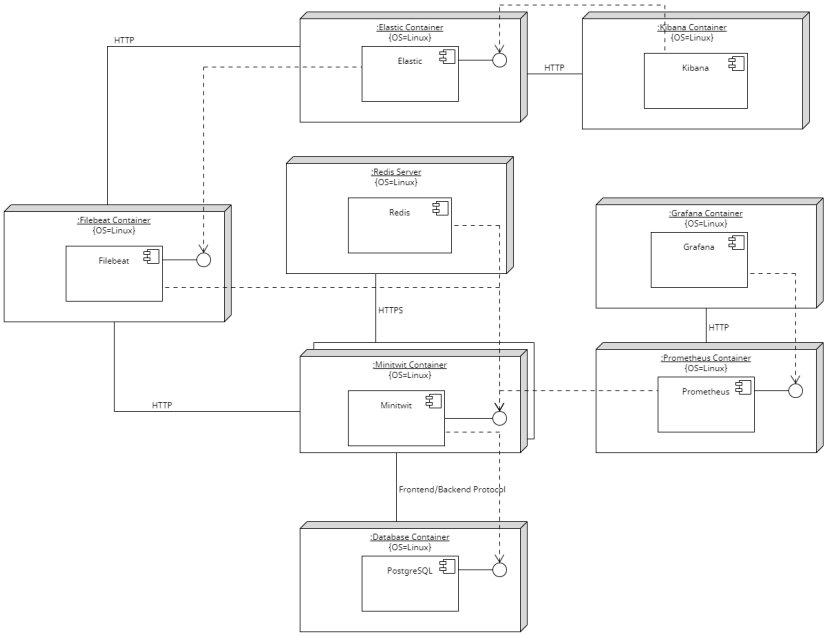

Figure 1: Minitwit deployment view

2.2 Dependencies of the system

Technologies and tools:

- FastAPI for the backend.
- PostgreSQL as database system.
- SQLAlchemy as an ORM framework.
- Docker for containerization.
- Vagrant for managing creation of virtual environments.
- DigitalOcean for hosting virtual environments on a remote server.
- GitHub Actions and Ansible for the CI/CD chain.
- Prometheus and Grafana for monitoring and visualising the state of the system.
- Postman for testing of the API.
- Elasticsearch, Filebeat and Kibana as a logging stack.
- Redis for storing global state.
- Nginx and Heartbeat for a high-availablity setup with a staging and deployment server.
- Certbot for adding HTTPS to the website.

Languages:

- Python for the web application.
- Jinja as a templating language.
- Dockerfile.
- YAML for Docker, Ansible and Github Actions

2.3 State of the system

The application is accessible on `opsdev.gg`. The code can be found in this repository. Everything is running without error and will be accessible until the date June 14th 2023. Our system has been able to process at least 12.778.829 [1] events in the span of 8 weeks.

2.4 Licences

Since profiteering is not our purpose the code is under the MIT License.

2.5 Weekly tasks

2.5.1 Week 1 - Bug fixing the old program

By importing the project to IntelliJ it auto-suggested to update the Python version from 2 to 3 and automatically converted all code. We used shellcheck to lint the bash script.

2.5.2 Week 2 - Refactoring the app in another language

We listed possible frameworks, voted to exclude and then used a roulette to pick between two. The process is described here [2].

The arguments for FastAPI, besides its familiarity to some of us, was how little change from the existing python code base was required and being able

[1] `http://opsdev.gg:3000/d/DVJQxp-4k/minitwit-responses?orgId=1&viewPanel=2`

[2] `https://docs.google.com/document/d/1EOrFKD0XFW5gxv1NycgtTuj9rO6dggau99LybLYIRq0/edit#heading=h.r8lyr8ciz36h`

to reuse Jinja. This would let us focus on the DevOps tasks rather than web development.

We experienced some unexpected downsides to FastAPI:

- The implementation of redirection for instance redirecting from a POST to another POST request would mean the loss of the original body and no way to pass it along.
- Redirecting between different routes that return different status codes was also an unexpected challenge.
- Since it is multi-threaded server it excludes the ability to have a global state and a session. We solved it with Starlette Session but still caused limitations in regards to combining it with middleware.
- The tests required to be translated to "pytest" framework in order to be utilised.

2.5.3 Week 3 - Virtualisation and deployment to the server

We created a single Dockerfile and set up Docker Compose.

A new Docker feature called Dev Environments gave us feature errors with mounting from files to containers and hot reloading.

2.5.4 Week 4 - Setting up CI/CD system

Github Actions gave us a simple configuration process without needing additional hosted infrastructure. See section 3.

Fetching secrets between Github Actions and Ansible were solved by creating a file on the server instead of storing all of them in Github Actions Secrets. This was only fixed 2 weeks later.

2.5.5 Week 5 - ORM

Three ORMs have been considered and the arguments are mapped [3]. A benchmark repository was created [4] and after a Github Discussion [5] a vote was held. The choice fell on the most popular choice for Python - SQLAlchemy - and it did not cause regret.

2.5.6 Week 6 - Monitoring

Prometheus with Grafana is considered the most popular choice in the industry [6].

[3] `https://docs.google.com/document/d/1EOrFKD0XFW5gxv1NycgtTuj9rO6dggau99LybLYIRq0/edit#heading=h.auxry4rgtrx1`

[4] `https://github.com/MinitwitGroupI/ORM_Benchmarks`

[5] `https://github.com/MinitwitGroupI/MiniTwit/discussions/57`

[6] https://stackshare.io/stackups/grafana-vs-nagios

FastAPI is multi-threaded and thus doesn't support having a global state. Storing the total http calls would only pertain to the current thread. This was solved by storing the value in Redis. This was done in an asynchronous way that wouldn't impact performance.

We audited our partner groups website and created issues for known issues [7] [8].

2.5.7 Week 7 - Maintainability, external code analysing

The Flake8 linter used by pylint does static code analysis in our CI pipeline to provide soft feedback that does not block builds but hint at potential problems. SonarQube and CodeClimate integrated perfectly with our repository but gave vastly different views on code quality than us. It required manually dismissing a lot of warnings about duplicated code which existed in our unit tests. It ended up being a burden rather than helpful.

2.5.8 Week 8 - EFK stack

Significant issues were encountered when setting up the EFK stack:

- Poor support for encoding custom user credentials in Kibana and Elastic Search.
- User credentials on Filebeat had to be manually hard coded.
- With regards to security it was difficult to enable for each service and could only be tested in production given our setup.
- Adjustment of versions were required from the provided guide [9].

Implementing the EFK stack required vertical scaling the server in terms of CPU, memory and disk size.

The postmortem exercise was completed. A bug was introduced an it was discovered with the help of logging. TODO DESCRIBE IT link?

2.5.9 Week 9 - Security assessment

Metasploit and OWASP ZAP scans revealed minor vulnerabilities in the frontend. WHAT WERE THEY???

The idea of integrating OWASP ZAP to the CI/CD pipeline was considered but dismissed on grounds of being beyond the scope.

A security assessment was performed on our partner group .

[7] `https://github.com/organizationGB/DevOps/issues/23`
[8] `https://github.com/organizationGB/DevOps/issues/25`
[9] https://github.com/itu-devops/itu-minitwit-logging

2.5.10 Week 10 - Scaling and rolling out updates

We opted to go with the high-availability setup, since using a scalable deployment would require refactoring several components of our stack.

We already implemented deployments to a staging environment prior to production, so adding the staging server to our high-availability setup also provided us with rolling updates.

2.5.11 Week 12 - Encode your infrastructure setup

Changing our existing IaC from Vagrant to Terraform did not provide any perceived benefits. Vagrant works for both provisioning and configuration management. Besides the lack of benefits the downside of Terraform is having to deal with state.

3 Process' perspective

3.1 Team Organisation and Developer interaction (Ivar)

Since creating the Github organisation MinitwitGroupI team interaction was centered around Github and Discord. Discord was used for informal chatter and to contact each other. For tasks and discussions Github issues were used. This gave us a Project View with a visual Kanban [10]. A higher use of Github as a communication platform was observed as time progressed.

We strived to ensure that everyone worked on every aspect of the project, promoting a well-rounded understanding of each component. However, certain sections of the project were more esoteric and hacky, making it challenging for everyone to fully grasp the intricacies of every component.

3.2 Stages and tools utilized in CI/CD chains (Ivar)

We use GitHub Actions for performing all CI/CD operations.

3.2.1 Continuous Integration (CI)

Upon each commit, Github Actions triggers a job that builds our application using Docker. This process creates a Docker image, which is then used as the environment for running our custom unit and integration tests.

For a pull request to be merged, it must successfully pass all unit and integration tests.

3.2.2 Continuous Deployment (CD)

Once a pull request has been merged with our main branch, our CD pipeline is automatically triggered. The first step in this pipeline is pushing the new versions of the Docker images to our image repository.

[10] `https://github.com/orgs/MinitwitGroupI/projects/1`

Using Ansible, we first deploy our entire application stack to a staging server. This environment allows us to verify the functionality and stability of our application before deploying it to production.

After successful deployment to the staging server, we run liveness tests using Postman. These tests are designed to validate the overall health and performance of our application, ensuring that it meets the necessary requirements for production deployment.

A passing liveness test is required to deploy to our production server.

3.3 Organization of the repository

`https://github.com/MinitwitGroupI/MiniTwit/blob/main/docs/CONTRIBUTE.md` That is, either the structure of of mono-repository or organization of artifacts across repositories. In essence, it has to be be clear what is stored where and why.

this is already mostly explained in our CONTRIBUTE.md file

3.4 Branching strategy (Anders)

We use the feature branches strategy for individual development. For releases [11] we achieved trunk based development by pushing into main and manually bundling a release. In order to push to main we had setup branch protection that require pull requests to be reviewed by at least one other team members[12]. Our branching philosophy is described here [13].

3.5 Monitoring (Mads)

We use Grafana and Prometheus to gather and visualize data on various aspects of our system, including endpoint hits, response times, total number of requests, CPU usage, and request duration distribution. These metrics are displayed across five different panels that provide a comprehensive overview of our system's performance. By monitoring these key metrics, we can quickly identify any issues that may arise and take action to ensure that our system is running smoothly and efficiently. You can find the dashboard here[14].

todo: Explain why we find the metrics to be of interest.

3.6 Logging (Anders)

For logging we use the EFK stack. The Elasticsearch dashboard can be found on `http://opsdev.gg:5601` with the username helgeandmircea and password sesame0uvr3toi.

[11] https://github.com/MinitwitGroupI/MiniTwit/releases

[12] Initially two members were required to review it but we found it stifling to development and change it to one.

[13] https://github.com/MinitwitGroupI/MiniTwit/blob/main/docs/CONTRIBUTE.md

[14] `http://opsdev.gg:3000/d/DVJQxp-4k/minitwit-responses?orgId=1&from=now-24h&to=now`

3.7 Security Assessment

- All attacks with Metasploit and OWASP ZAP failed to produce any significant results on our website.
- The peer team attempted to exploit the website but also did not produce any note worthy results.
- Reviewing the peer team resulted in the other teams database and cookies being compromised. We go into depth in the appendix .

3.8 Scaling and Load balancing (Anders)

We have opted out of a scalable setup since this would require us to heavily refactor our infrastructure and separate our database from individual nodes. This would cost more with no added benefit in the confinement of the expected user growth before the deadline.

3.9 High-availability setup (Anders)

We have opted for a high-available setup by using heartbeat using a guide provided by DigitalOcean [15]. This utilizes 'floatip' which uses the DigitalOcean API to update the reserved IP to point at the other server when one is unavailable.

????

4 DevOps principles

4.1 What is DevOps

Going by the DevOps Handbook [2] we have chosen a couple defining concepts to compare our result against.

Todo: DevOps principles and how much we adhered to them (and how).

The categories are from here: `https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_05/README_TASKS.md`

4.1.1 Flow

Make work visible Limit Work in Progress Reduce Batch Sizes Sometimes it was not possible to create small batch commits. The limitation being

Reduce the Number of Handoffs Continually Identify and Evaluate Constraints Eliminate Hardships and Waste in the Value Stream

[15] https://www.digitalocean.com/community/tutorials/how-to-create-a-high-availability-setup-with-heartbeat-and-reserved-ips-on-ubuntu-16-04

4.1.2 Feedback

See Problems as They Occur Swarm and Solve Problems to Build New Knowledge Keep Pushing Quality Closer to the Source Enable Optimizing for Downstream Work Centers

4.1.3 Continual Learning and experimentation

Institutionalize the Improvement of Daily Work Transform Local Discoveries into Global Improvements Inject Resilience Patterns into Our Daily Work Leaders Reinforce a Learning Culture

4.2 Another view - DevOps as a culture

DevOps has enjoyed many different views through different eras [3]. The supplemental book, The Phoenix Project [1] and the guest lecture by Zander Havgaard from Eficode provides a fresh perspective of DevOps as a culture.

Psychological safety: (Explain how we were at odds in the beginning when selecting a framework).

"The Bus Factor" was repeated during the guest lecture which is the concept that no single developer should become key to development or infrastructure to the degree that progress can't occur without them. We failed in this regard. We each carved out our niche based on previous knowledge. While PR reviews occurred there was insufficient knowledge sharing for role shifting.

4.2.1 Ways we are not DevOps

We don't follow the Toyota Kata.

TDD... We could've written more code tests. Though with Postman we were able to test the availability of our infrastructure before deploying. Reflection: This seems like the typical case of tests being on the cutting block when time is scarce.

5 Lessons Learned Perspective

5.1 Issues and lessons learned (All)

5.1.1 Refactoring

Refactoring was a breeze because we choose FastAPI which allowed us to reuse big parts of the original Flask application. While Flask would've sufficed we managed to add better security through password encryption in our refactoring.

5.1.2 Evolution

We were often blocked by upcoming tasks. We were waiting for the ORM task to finish so that we could create database migrations. We also anticipated the lecture about infrastructure as code since it would solve ????.

5.1.3 Maintenance (Anders)

Thanks to our monitor dashboards and Postman monitors we picked up on some infrequent outages. We could not verify whether some were caused by the simulator pausing or a problem on our end. We experienced at least three types of known outages:

1. Outage during deployment while the Docker image was pulled and started. This was solved by our high-availability setup.
2. Didn't we experience some outages that were caused by the high-availability setup being misconfigured?
3. The daily 5 AM outage. Through research we discovered that PostgreSQL by default restarts and does garbage collection and other maintanance.

Overall we felt that a minimal amount of maintenance is needed and would confidently hand over the project to another team.

5.2 Uptime

The metrics in our SLA[16] are based on postman and our monitoring dashboard.

5.3 Software Quality

Static code analysis (the linting pipeline)
SonarQube through SonarCloud
Code Climate
Terms we could use: SDLC, Technical debt

5.4 Reflection (All)

Also reflect and describe what was the "DevOps" style of your work. For example, what did you do differently to previous development projects and how did it work?

5.4.1 THE SECTION BELOW IS JUST FOR NOTING DOWN REFLECTIONS, NEEDS TO BE REWRITTEN

- Our branch strategy of having to review each others PRs for knowledge sharing and keeping a high code quality quickly became somewhat useless as we worked on a lot of new technologies with no prior experience, so it was hard to tell if something was inaccurate.
- Optimally we should have a test environment as well as an production environment, as some changes were hard to test locally and we just pushed strait to production and sometimes caused the application to go down.

[16] https://github.com/MinitwitGroupI/MiniTwit/blob/main/docs/SLA.md

- We waited until the CI lecture to fully implement testing, meaning that all our deployments were based on educated guesses, meaning that things which worked locally often failed in production. Additionally, we also didn't have CD until the same period, which resulted in weekly releases that failed most of the time. That required us to go through all changes made since the last release and bugfixing them.

5.5 Hand over

We consider that no further maintenance is required. If a new group were to inherit our codebase we have created a document to guide them TODO.

6 Conclusion (All)

Answer the question: Is what we've achieved DevOps?

References

[1] Kevin Behr Gene Kim and Georg Spafford. *The Phoenix Project*. IT Revolution Press, 2018. ISBN: 9781942788294.

[2] Patrick Debois Gene Kim Jez Humble and John Willis. *The DevOps Handbook*. IT Revolution Press, 2021. ISBN: 1950508404.

[3] Github. *What is DevOps?* URL: `https://www.youtube.com/watch?v=kBV8gPVZNEE`. (accessed: 07.05.2023).

7 Appendix

A Hand over guide

B Security Assessment (Partner Group)