DevOps 2025

IT University of Copenhagen

Team Sad People

People

- *Gábor Tódor* - gato@itu.dk
- *Kálny Zalán* - zaka@itu.dk
- *Nicklas Koch Rasmussen* - nicra@itu.dk
- *Nicolai Grymer* - ngry@itu.dk
- *Sebastian Andersen* - seaa@itu.dk

Subject

- **School:** *IT-University of Copenhagen*
- **Course manager:** *Helge Pfeiffer* - ropf@itu.dk
- **Course Code:** KSDSESM1KU

Introduction

System

A description and illustration of the:

[seb/nick] Design and architecture of your *ITU-MiniTwit* systems

[Nic] All dependencies of your *ITU-MiniTwit* systems on all levels of abstraction and development stages. That is, list and briefly describe all technologies and tools you applied and depend on.

[Nic] Important interactions of subsystems.

For example, via an illustrative UML Sequence diagram that shows the flow of information through your system from user request in the browser, over all subsystems, hitting the database, and a response that is returned to the user.

Similarly, another illustrative sequence diagram that shows how requests from the simulator traverse your system.

- Draw up a single diagram, that shows how a requests propagate through out the system. Given that both sim and regular api are within the same application, just a single request needs to be shown.
- Use docker-compose to list out all the different components, that are hit (e.g. prometheus and logs/kibana)

[G] Describe the current state of your systems, for example using results of static analysis and quality assessments.

[ALL] MSc students should argue for the choice of technologies and decisions for at least all cases for which we asked you to do so in the tasks at the end of each session.

Process

This perspective should clarify how code or other artifacts come from idea into the running system and everything that happens on the way.

In particular, the following descriptions should be included:

[Nic] A complete description of stages and tools included in the CI/CD chains, including deployment and release of your systems.

1. Source Code Management We use Github for handling our repository and tracking process with their issue system. We use a branching strategy, where features written in issues are created in `feature`-branches, which are then merged into `staging` and then into `main`. This enables us to test and deploy the feature before production, at the cost of slightly longer delivery times. Before being able to merge into staging and main, code needs to pass all unit, e2e and simulation tests, check Sonar Cubes quality gate, and lastly be approved by another member of the team.

2. Continuous Integration On push to staging and main, we built and test using Github Actions.

Build tools & environments (e.g., Docker, Make, Gradle)

Automated tests (unit, integration, linting)

Test orchestration tools (e.g., Jest, Pytest, Mocha, Cypress)

3. Artifact Management Build artifacts (e.g., JARs, Docker images, binaries)

Artifact storage (e.g., Nexus, JFrog Artifactory, GitHub Packages)

4. Continuous Delivery Staging environments

Deployment automation tools (e.g., GitHub Actions, GitLab CI, Jenkins, CircleCI)

Infrastructure as Code (e.g., Terraform, Pulumi, Ansible)

5. Continuous Deployment Rollout strategy (e.g., canary, blue/green, rolling)

Orchestration platform (e.g., Kubernetes, ECS, Nomad)

Deployment verification (e.g., smoke tests, health checks)

6. Release Management Versioning strategy (e.g., SemVer)

Release approval workflows

Feature flagging (e.g., LaunchDarkly, Unleash)

Changelog generation

7. Monitoring & Feedback Logging and error tracking (e.g., ELK, Sentry, Datadog)

Metrics & dashboards (e.g., Prometheus, Grafana)

Alerting setup (e.g., PagerDuty, Opsgenie)

8. Security & Compliance Static and dynamic code analysis tools (e.g., SonarQube, Snyk, CodeQL)

Secrets management (e.g., HashiCorp Vault, AWS Secrets Manager)

Policy enforcement (e.g., OPA, GitHub branch protection rules)

- Github Issues
- Local branch
-

[Z/G] How do you monitor your systems and what precisely do you monitor?

[Z/G] What do you log in your systems and how do you aggregate logs?

[Nic] Brief results of the security assessment and brief description of how did you harden the security of your system based on the analysis.

[Nic] TLS - 2FA - Port forwarding - DB encryption - ORM

https://claude.ai/chat/0b811d0e-7c05-4aca-9bb0-e099e4d4bcd5

By running through the OWASP Top 10 list on security assessment, we have done the following analysis:

- A01:2021-Broken Access Control After assessing url endpoints, no evident broken control access is present. Only thing that we have not adjusted for is CORS settings, which
- A02:2021-Cryptographic We've upgraded from HTTP to HTTPS, but still expose the port of the application, meaning IP:PORT still gives users access to the service in non-encrypted ways, such that network eavesdroppers can capture username and passwords. The hashing algorithm has been upgraded from MD5 to SHA256, but unfortunately without salting, allowing attackers who gain access to the database to easily crack passwords with rainbow tables or brute force attacks. Lastly, the simulator protection-key is hard-coded which means anyone with access to the public github repo, can essentially bypass that security measure.
- A03:2021-Injection We use the ORM Ruby Sequel, which includes sanitization of input before constructing SQL statements. Developers can create raw SQL statements, but we have opted not to do this given the impracticality and security risks.

F2A on Digital Ocean & Cloudflare

A04:2021-Insecure Design is a new category for 2021, with a focus on risks related to design flaws. If we genuinely want to "move left" as an industry, it calls for more use of threat modeling, secure design patterns and principles, and reference architectures.

A05:2021-Security Misconfiguration moves up from #6 in the previous edition; 90% of applications were tested for some form of misconfiguration. With more shifts into highly configurable software, it's not surprising to see this category move up. The former category for XML External Entities (XXE) is now part of this category. A06:2021-Vulnerable and Outdated Components was previously titled Using Components with Known Vulnerabilities and is #2 in the Top 10 community survey, but also had enough data to make the Top 10 via data analysis. This category moves up from #9 in 2017 and is a known issue that we struggle to test and assess risk. It is the only category not to have any Common Vulnerability and Exposures (CVEs) mapped to the included CWEs, so a default exploit and impact weights of 5.0 are factored into their scores. A07:2021-Identification and Authentication Failures was previously Broken Authentication and is sliding down from the second position, and now includes CWEs that are more related to identification failures. This category is still an integral part of the Top 10, but the increased availability of standardized frameworks seems to be helping. A08:2021-Software and Data Integrity Failures is a new category for 2021, focusing on making assumptions related to software updates, critical data, and CI/CD pipelines without verifying integrity. One of the highest weighted impacts from Common Vulnerability and Exposures/Common Vulnerability Scoring System (CVE/CVSS) data mapped to the 10 CWEs in this category. Insecure Deserialization from 2017 is now a part of this larger category. A09:2021-Security Logging and Monitoring Failures was previously Insufficient Logging & Monitoring and is added from the industry survey (#3), moving up from #10 previously. This category is expanded to include more types of failures, is challenging to test for, and isn't well represented in the CVE/CVSS data. However, failures in this category can directly impact visibility, incident alerting, and forensics. A10:2021-Server-Side Request Forgery is added from the Top 10 community survey (#1). The data shows a relatively low incidence rate with above average testing coverage, along with above-average ratings for Exploit and Impact potential. This category represents the scenario where the security community members are telling us this is important, even though it's not illustrated in the data at this time

[Seb/Nick] Applied strategy for scaling and upgrades.

[Nic] In case you have used AI-assistants during your project briefly explain which system(s) you used during the project and reflect how it supported or hindered your process.

This project included the use of both chatbots and in-editor help using copilot. These were provided by OpenAI and Anthrophic.

Copilot increased speed by solving minor problems through it's line-for-line help. To increase the precision, prompt-like comments would be added prior to the line of interest, or specific prompts would be used to concretely specify the desired change.

Chatbots on the other hand involved four primary types of prompts:

- Elaboration: Please explain X technology
- Comparing: What is the difference between X and Y technology
- Creation: I want to X
- Solving: I want X, but instead Y happens.

Elaboration and comparison were primarily used at the planning stage of implementing new technologies, or for developers unfamiliar with existing technologies already implemented.

Creation is used throughout the implementation of technologies or features, but the scale of the issues attempting to address diminishes over time, as the feature or technology becomes more intergrated into the system, and required changes are smaller. As more code is added to the codebase, solving unwanted behavior becomes more important, and makes out large parts of prompts.

Additional reflection on use of chatbots, we found that Claude 3.7 Sonnet provided better code-based responses as well as understanding misconfigurations and bugs. It gives detailed descriptions of different variables and potential flaws in the code and configs. This is measured against ChatGPT o1.

Reflection

Evolution and refactoring

Finishing a sprint and adding a new feature

[G/Z] ELK logging resource heavy + too many fields + all fields were indexed

[G/Z] Filebeat on all swarm nodes

[G/Z] Logging deployment: put config images in

[Nic] Database migrating from SQLite to PostgreSQL to PostgreSQL

The migration from SQLite to Postgresql happened at a stage, where no active users was using our platform (The simulator was yet to start). This meant that we could safely upgrade without having to move over data, which would have otherwise been a hassle given the SQL dissimilarities.

Given the educational purpose of the project, we later sought out an opportunity to perform a database migration. Such an opportunity arose when database optimization became a necessity. Before optimizing, we deemed it necessary to introduce an ORM, which would improve the developer experience as well as migration experience going forward. Given the new database structure introduced by the ORM, although quite similar, we needed to migrate from one database with one schema, to another database with another schema. The approach taken involved extracting data from one postgres instance in the shape of SQL Insertion statement, which we then manipulated to fit the new data scheme, and then simply ran the sql insertion statements in the new database.

As soon as the migration was done, we switched to the new application image, meaning we now served requests from the new database. This approach involved having 5 minutes of forgotten data, and 3 seconds of lost availability. We found this price and strategy reasonable, although the 5 minutes of lost data, could have had serious impact on the business. As we will later discuss, we found that using logical replication, proved to be a much nicer approach to copying data.

[Nic] Transition from docker compose to docker swarm (networking problems).

Transitioning onto multiple machines with docker swarm came with multiple obstacles. First, the docker compose version running on certain of the docker compose scripts, were unsupported by docker swarm. **[Seb/Nick] Docker compose versioning problem (moving to stack)**.

Second, the swarm nodes were able to communicate with each other, but self-instantiated virtual networks defined in the docker-compose file, did not propagate to worker nodes, leaving application containers unable to contact the database, and prometheus unable to collect monitoring events. To accommodate the issue, we destroyed and redeployed new virtual machines, and this time used the VPC IP address to define the IP address of the manager node. This meant that workers are referring to the manager using the virtual network layer, and solved the communication issue.

[G/Z] scp files onto server and then deployment (Discuss ups/downs)

```
- You can destroy the prod environment with wrong files/wrong docker compose
```

[G/Z] Transition from config files to docker images (Tagging docker containers)

[Seb/Nick] Large amount of features cloging up in staging (Impossible to migrate to production)

Operation

Keep the system running

[Nic] Database logical replication resulting in db crash

Migrating from docker compose to the docker swarm included the use of postgres feature: Logical replication, which allows postgres instances to live sync data from running postgres instance to the other. This feature is typically used to keep a hot stand-in database ready. In our case, it meant we would actively sync data from the active production database, onto the new production database, and allow us to switch from one stack to the other with zero downtime, as the stand-in database would become the new default.

Unfortunately, after switching a few days later, the pub/sub mechanism of logical replication in Postgres accidentally corrupted a tracking file, meaning the postgres would immediately crash on start. This problem was accomodated by immediately running `pg_resetwal` on startup to reset the corrputed file, and then unsubscriping from the expired subscription. The subscription does not provide any value at this point, as we swapped from the old to the new production machine, and the old one has been turned off.

[G/Z]Log overflow problem. Access denied to machine. Massive clutch

[Nic] Backup strategy (cron job every three hours)

Although it's great to solve problems on yuor own, sometimes other have done a great job already. And this proved to be the case for backing up a Postgres database. Simply adding the `eeshugerman/postgres-backup-s3:15` container image to the manager node and configuring environment variables, we successfully setup a cron job that automatically backs up daily, and sends the backup to DO's space storage, which is S3 compatible. Using the exact same script, it also includes functions that easily allow restoring from a previous backup. The latter is a crucial step, for when things are burning. The container likewise provide clean-up functionality, such that only 7 days of backups are kept.

Maintenance

Keep system up to date and fix bugs

[Seb/Nick] Stale ReadMe.md throughout project

[Seb/Nick]Returning wrong statuscode (Misalignment with simulation)

- Thanks to running similator in the CI/CD pipeline we found this

[Nic] Upgrading to NGINX, setting up ufw, moving to domain

Upgrading to NGINX, we learned multiple things about running a system. First being that having a staging environment to learn how to run command in the correct order proved great to build a shell script that immediately upgrades the production service. Second, that although we had configured our ufw with all the right ports, actually the service was disabled, which it is by default. Only by realizing that the 443 port was open although no specified, did we realize that ufw needs to be actively activated. From this we gathered, that it is important to double check firewall and other security measures, to ensure they're configured properly. Luckily, our database was not exposed by PORT from the docker network, and therefore inaccessible, but having full access to other ports may have exposed other vulnerabilities on the machine.

[Nic] Simulator IP protection stopped sim access (causing errors)

Refactoring simulator requests to only be accepted from a single IP address helped us prevent other malicious users from interfering with the active simulation requests. However, when the new update was pushed, unfortunately the IP protection feature also protected us from the actual simulation IP. Although this worked perfect locally, moving it onto production showed that the feature declined all sim api requests. From this we learned about the importance of being able to quickly roll back an update. During this experience, we found that we did not have a previous versioned container ready to roll back to, and instead had to allow all IP's which was possibly by passing in `ACCEPTED_IPS=*`, which essentially disabled the IP protection.

Style of work

Reflect and describe what was the "DevOps" style of your work.

[Seb/Nick] Reflect on the workflow. Extensive Friday meeting. Split work into three groups

[Seb/Nick] Development environemnt: local => branch => staging => production

[Seb/Nick] Repo settings. Workflows on merge. Require 1 team member on pull requests.

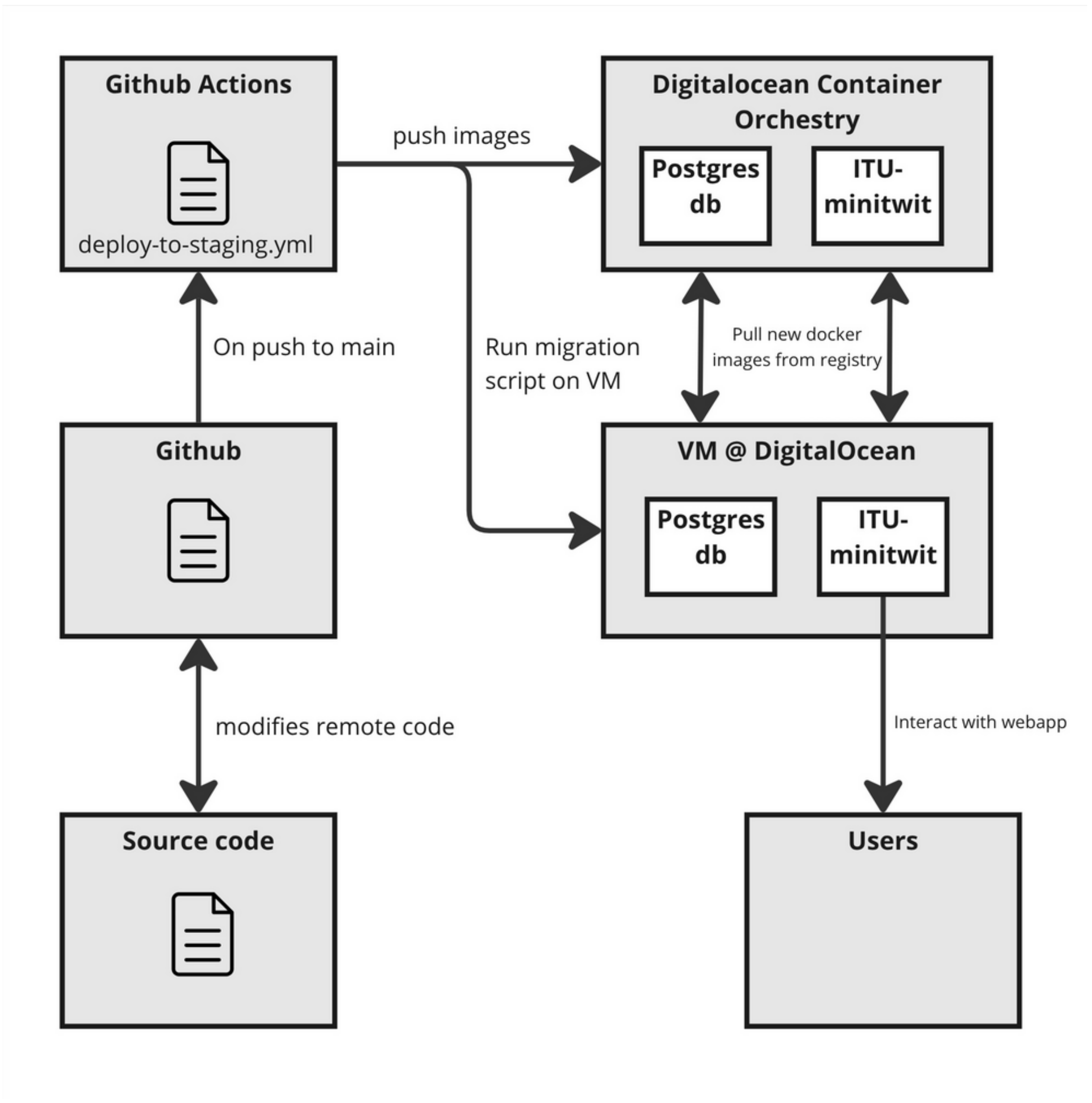
[Seb/Nick] Running simulator in workflows

That's it folks!

REMEMBER TO REMOVE THIS

THIS IS JUST AN EXAMPLE

THIS IS JUST AN EXAMPLE



hello a LOL fdsa