IT UNIVERSITY OF COPENHAGEN

Project Report

DevOps, Software Evolution and Software Maintenance

Group O "TBD"

Andreas Kongstad
`kong@itu.dk`

Charlotte Schack Berg
`csbe@itu.dk`

Eythor Mikael Eythorsson
`eyey@itu.dk`

Christian Lyon Lüthcke
`clyt@itu.dk`

Frederik Baht-Hagen
`fbah@itu.dk`

Instructors:
Helge Pfeiffer & Mircea Lungu

May 23, 2022

Contents

1. System's Perspective

1.1. Overview

The initial project was a full stack Flask application written in Bash and Python2 with an SQLite database. This implementation (henceforth referenced as TBD-MiniTwit) included refactoring the initial project to the following containerized microservices:

- C# backend using the ASP.NET web framework and EF Core ORM
- ReactJS SPA frontend
- PostgreSQL database

upon which a monitoring and (temporarily) a logging stack were added to be served as a containerized application behind a load-balancer on a managed Kubernetes cluster.

1.2. System Design

The heart of TBD-MiniTwit is the C# backend. It contains all our business logic and provides two open APIs, an Object-Relational Mapping to our database and exposes application-specific metrics for Prometheus.
CLASS DIAGRAMS FOR BACKEND
STATE MACHINE DIAGRAM FOR LOGIN

1.3. System Architecture

This implementation follows the **Microservice Architecture Pattern**
EXPLAIN THE PATTERN
CREATE DIAGRAMS

1.4. Subsystem Interactions

Prometheus
MinitwitAPI
LoadBalancer

Table 1: Network interfaces

1.5. System state

1.6. License Compatibility

The project uses MIT license meaning everyone can use and modify.

2. Process' Perspective

2.1. Overview

The main focus of the Process' Perspective section will be how code and artifacts goes from idea all the way to production and which tools and processes are involved?
2.2 Team Interaction and Organization will discuss communication within the team. 2.3 CI/CD chains will reason for the choice of CI/CD platform and take a deeper look and both CI and CD chains. 2.4 Version Control discusses strategies relates to the way version control is used and managed. These include repository setup and branching strategy. In 2.5 Development process and tools we discuss how issues and projects were used to track, label, and assign tasks within the team. 2.6 Monitoring And Logs will introduce the Prometheus grafana stack and the elk stack and what value these have brought to service. 2.7 Security will be a brief rundown of result from the security analysis and pentest. The final section 2.8 Scaling And Load balancing will focus on the migration from a single server architecture to a kubernetes cluster and provide discussion on some of the pros and cons of using kubernetes for scaling and load balancing.

2.2. Team Interaction and Organization

Most internal planning and communication have gone through [4]. The team has a server with text channels for planning, resources etc. Work has been split between remote using discord voice channels and physical at and in continuation of the allotted exercise time on Tuesdays.

2.3. CI/CD chains

(Andreas)
Utilizing a CI/CD pipeline allows us to incrementally test, build and deploy the MiniTwit service with minimal manual interference. Thus saving us a significant amount of time setting up environments and allows us to deploy fast and often while still being able to easily revert to an earlier version. It provides us with ways of measuring and improving the quality of all code coming from local development to version control and eventually reaching production, and by that reduces human error.[1] In essence an automated CI/CD pipeline puts multiple DevOps ideals into practice:

- Flow (Keeping batch sizes small)[2]
- Feedback (Instant, rapid and continuous feedback on code entering pipeline)[3]

2.3.1. CI/CD Platform

An abundance of CI/CD platforms are available to developers. Our CI/CD chains are set up using GitHub Actions. GitHub Actions integrates seamlessly with our GitHub repositories and allows us to trigger workflows every time an event occurs in a repository[4]. Many other providers

[1]Continuous delivery: Huge benefits, but challenges too[1]
[2]The DevOps handbook[2]
[3]The DevOps handbook[2]
[4]Understanding GitHub Actions[7]

such as Travis CI[6] or TeamCity[5] offer these same features and some even more, but not for free and not with little to no initial configuration. We are however not blind to concerns with using the same provider for most tools. The most notable of these would be bigger consequences when GitHub inevitably goes offline or experiences problems. We accept these concerns as we prefer the ease of use and price tag over distributing our tools and having smaller chance of being impaired due to provider outages.

2.3.2. CI - Continuous Integration

As illustrated in Figure 1, the entry point of for the CI pipeline is creating a pull request to the main branch, which will trigger a number of GitHub Actions workflows.

1. **.Net Build and test -** The backbone of the CI pipeline, which compiles, builds and tests the backend. It provides us immediate feedback on whether the changes run and passes the test suite. When implementing the backend we spent a significant amount of time and effort setting up a test suite comprised of unit tests and integration tests, which allows us to trust that this workflow will catch breaking change coming in to the codebase.

2. **Quality analysis tools -** These tools all provide feedback on the quality of the pushed code in accordance with their own definition of code quality. These include .Net analyzers, Sonarcloud, Better Code Hub, and DeepScan.

3. **Dependency scanning tools -** Scans dependencies and codebase for security hazards and fails if the security gate specified to "Critical" severity is met. These include snyk and CodeQL.

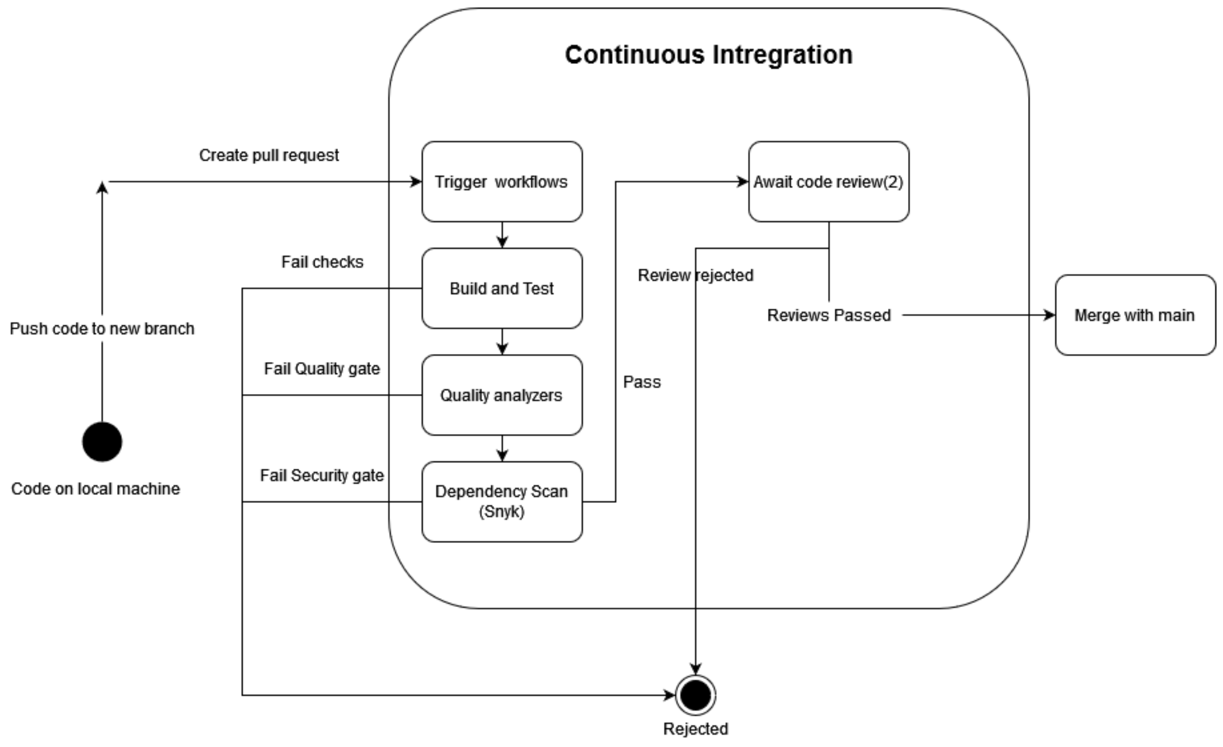

Figure 1: CI Pipeline State Machine Diagram

As suggested by figure 1, in the event that any of these fail, the CI pipeline will direct to a rejected state, from where a developer can fix problems, push to the rejected branch and the workflows will rerun. Once all workflows are passed, the pull request await review until at least 2 members of the team has approved it. From here changes can be merged to the main branch.

2.3.3. CD - Continuous Delivery/Deployment

Our pipeline introduces a mix of Continuous Delivery and Continuous Deployment(Illustrated in Figure 2). Deployment is done entirely by the deployment workflow(cluster-deploy.yml). The workflow is triggered every time a release is created. It also supports manual dispatch for hotfixing errors, and the weekly release workflow runs every Sunday evening, triggering the deployment pipeline. The deployment workflow is comprised of 4 jobs.

1. **.Net Build and Test** - This job is described in CI - Continuous Integration.
2. **Build and Push containers** - Builds docker containers for the frontend and backend, tags them with proper versions, then pushes them to docker hub. This allows us to keep our Operations repository lean as we can then pull all necessary containers form docker hub.
3. **Snyk Dependency scan** - Our security gate. If a risk exceeding the gate is found the deployment will stop immediately and move to the cancelled state. See Figure 2
4. **Dispatch** - Dispatches the apply workflow in the Operations repository(submodule?).

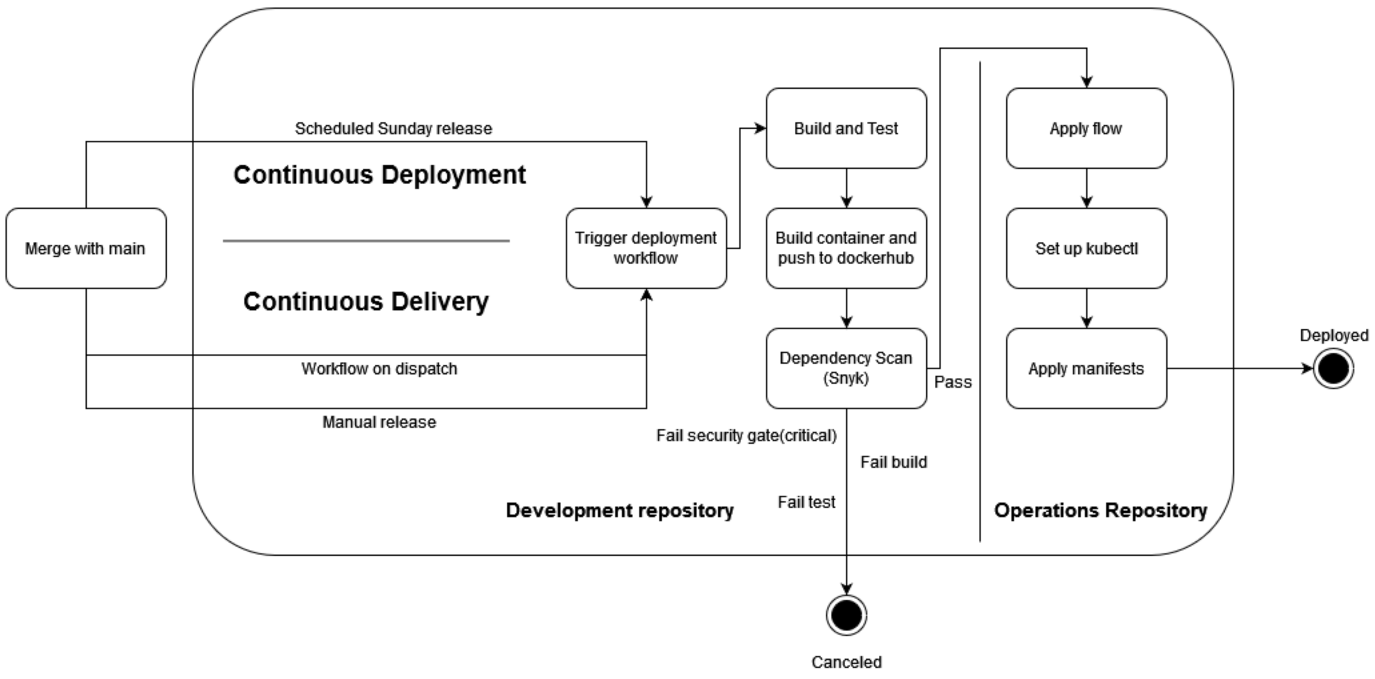

Figure 2: CD Pipeline State Machine Diagram

When checks have passed and the apply workflow has been dispatched, the runner will setup kubectl using a Digital ocean token and cluster name saved as secrets in the repository. With access to kubectl the runner now has access the kubernetes cluster. The apply shell script is executed, which "applies" all the configuration manifests, thus deploying the changes.

2.4. Version Control

THIS SECTION SHOULD PROB BE BEFORE CI/CD

2.4.1. Organization of Repositories

2.4.2. Branching Strategy

Our organization uses a Trunk Based Development branching model. we have two centralized branches which were always debugged, tested and in a working state. Features, patches and bug-fixes are always developed on dedicated temporary branches which are deleted after being merged to its relative centralized branch.

Dev Repository @ main

The **main** branch lives on github and is the alpha of our centralized workflow. While we develop features and patches on temporary branches, everything worthwhile is eventually merged to main.

Primary applications of the main branch are:

1. Store our working source code

2. Make releases
3. Run workflows
4. Build and push docker images

Ops Repository @ master

The master branch of the Ops repository contains infrastructure as code for creating and configuring a kubernetes cluster with digital ocean[5] as provider, manifest files used to deploy our service to the cluster, and shell scripts for automating these steps. It is developed in a manner such that our multi-container application can be rebuilt and updated to include updated Docker images and/or changes in configuration files requiring minimal effort.

2.5. Development process and tools

GitHub issues are used to track what needs to be done, and how far a task is from completion. Upon creation, issues are tagged and organized into GitHub Projects.

2.6. Monitoring And Logs

2.7. Security

2.7.1. Security assessment

2.7.2. Secret Handling

2.8. Scaling And Load balancing

2.8.1. Single server setup

The original single server setup that was deployed on a digital ocean droplet using docker compose is located on **Dev Repository @ production** Although now deprecated, the production branch had our production server and was our main branch's lean and automated counterpart. It was developed in a manner such that our multi-container application could be rebuilt and updated on the production server to include updated Docker images and/or changes in configuration files with a single command, minimizing downtime without using load-balancers.
Although cheap and easy to setup the original single server architecture had a number of shortcomings if we had to handle additional traffic while minimizing downtime. Only vertical scaling[6] using the digital ocean API. This approach has multiple downsides. It will become exceedingly more expensive as more virtual CPU cores, RAM, and disk space is added eventually reaching an upper limit, the single monolithic VM will forever be a single point of failure, and upgrading the VM will require the server to shut down. (MAYBE INCLUDE COMPONENT DIAGRAM)

[5]Digital Ocean[3]
[6]

2.8.2. Scaling the application

In order to scale for handling additional traffic while minimizing downtime, the team had a couple of options. Eliminating the server as a single point of failure while allowing for rolling updates out without shutting down the application, could be accomplished by introducing a setup with a hot server and backup server and swapping around IPs. But that would not allow for horizontal scaling[7].

Options that out of the box come with horizontal scaling, load balancing, and rolling updates while eliminating the single point of failure are the container orchestration tools Docker Swarm and Kubernetes. We have chosen migrate our application to a Kubernetes cluster. There is an argument to be made, that the additional complexity and setup required for deploying a Kubernetes cluster is unnecessary since Docker Swarm does fulfill our requirements. The first reason for the choice is that documentation on setup and management of a Kubernetes cluster was a lot more extensive, although that might be caused by the increased complexity. Secondly, the team wished to gain experience using Kubernetes for container orchestration. That said, automated scaling[8] does save time for developers, even if monitoring service load and manually scaling a swarm when required is a possible solution. In conclusion, by migrating to kubernetes cluster we support horizontal scaling, load balancing, and rolling updates. We must however admit, that even though the database was flushed and deployed to the cluster, we do not have a replicated database because of consistency concerns, and can therefore not claim to have eliminated all single points of failure.(THIS IS TRUE RIGHT? WE ARE NOT REPLICATING THE DATABASE?)

[7]

[8] https://kubernetes.io/

3. Lessons Learned

3.1. Refactoring

3.2. Maintenance

3.3. DevOps adaptation

References

[1] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, 2 2015, ISSN: 07407459. DOI: 10.1109/MS.2015.27.

[2] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook : How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. 2016.

[3] *Digitalocean*, https://www.digitalocean.com/, (Accessed: 17/05/2022).

[4] *Discord*, https://discord.com//, (Accessed: 19/05/2022).

[5] *Teamcity*, https://www.jetbrains.com/teamcity/, (Accessed: 18/05/2022).

[6] *Travis ci*, https://travis-ci.org/, (Accessed: 18/05/2022).

[7] *Understanding github actions*, https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

A. Appendix

A.1. Constitutional Artifacts

A.1.1. Development Repository

https://github.com/Akongstad/DevOps-group-p

A.1.2. Operations Repository

https://github.com/mikaeleythor/itu-minitwit-ops/tree/master