# DevOps, Software Evolution and Software Maintenance

**IT University of Copenhagen**

Thomas Tyge Andersen
thta@itu.dk

Ask Harup Sejsbo
asse@itu.dk

Joakim Hinnerskov
jhhi@itu.dk

Petya Buchkova
pebu@itu.dk

Kasper Olsen
kols@itu.dk

May 10, 2021

# Contents

# 1 System's Perspective

Automatic CI build test

## 1.1 Design of your ITU-MiniTwit systems

## 1.2 Architecture of your ITU-MiniTwit systems

## 1.3 Important interactions of subsystems

# 2 Process' perspective

## 2.1 How do you interact as developers?

There were a few channels of communication. Every monday, after the lecture we would have a "traditional" standup-meeting where each member could lay out potential issues.

After the task of the finished week were discussed, and dealt with, we would turn our focus on the coming week, and allocate which tasks should be solved by next monday. More on this later.

Since the course was organized with a lecture a week, and corresponding deadlines for the work, we would also often have a less formal standup on Sundays, in the case that certain tasks were lingering, or needed discussion.

By attempting to keep our tasks small and keeping up frequent communication on teams, we tried to keep a steady value stream going throughout the week. Thus we were also hoping to decrease the amount of lead time for tasks. This was however not always achievable because of external time pressures, and much of the processing time was often placed at the end of the week.

It's probably worth noting that we never met physically during working together, which increased the importancy of frequent communications on teams, and clear communication regarding which times one would be available.

## 2.2 How is the team organized?

Our team was inspired by the agile philosophies in how we organized work between us, although we did not adhere to any specific agile framework. As mentioned in the previous section, we started the week i.e. just after the lecture finished, with a meeting. This meeting served both as our "daily" standup, our sprint review, and the sprint planning. The start of the meeting was where our standup took place and everyone reported on whether they had completed their previous task, and if they were facing any impediments. During the standup we would then be updating the kanban board containing our backlog items. We would then proceed to do a smal lreview of our progress and determine if the goal of the previous sprint had been achieved. If there were any tasks from the previous sprint that we were not satisfied with, these would be carried over into the new sprint. When our review was finished we would proceed to break down the task for the given weeks sprint into smaller subtask where possible. The members of the team would then volunteer for working on whichever backlog items that they wanted to focus on, often having a pair of people working on the same task, though pair programming was not enforced at any point.

The only meeting that we had set in our calendar on a regualr basis was the one just after the lecture. We did meet throughout the week to work on the tasks, tohugh this was conducted adhock, as the individual pairs of developers deemed necessary. As mentioned in the previous section, some of these adhoc meetings had a tendancy to be held on Sundays. This was in order for everyone to have a quick catchip on what had happend, and what still needed doing before the sprint goal had been achived. These meetings usually only occured if we had determined in advance that we were still missing a couple of tasks.

## 2.3 Stages and tools

## 2.4 Organization of repositories

## 2.5 Applied branching strategy

In week 2 we discussed what branching strategy we should use. We quickly narrowed the field of possible strategies to two - Git Flow and GitHub Flow, since these were the two strategies most people had experience with within the group.

Git Flow is a strategy where the main branch is used for released code e.g. stable releases. A development branch is merged out from the main branch, which is used to branch out feature branches where actual new work is done. New features are then merged back into the development branch. Once the team is ready to make a release, the development branch is merged into a release branch, then into main and released.

GitHub Flow is more simple than Git Flow since it does not have release and development branches. New features are instead merged out from the main branch, and back into the main branch once the feature is done. This will result in that the main branch containing code that is not yet released if deployment is not automatically set up.

We initially decided to go with Git Flow, since we would work on features throughout the week, and then create a single release on Sundays. We quickly realized that releasing once a week was not satisfactory for our needs, since we often wanted features to be pushed immediately once ready, and merging back and forth between the main branch and the development branch was cumbersome.

By week 7, we decided to change our branching strategy to GitHub Flow due to the addition of static code analysis and the rewrite of API simulation tests from Python to Golang. The motivation for the change in strategy is a move towards Continuous Deployment. Our pipelines now supported a full deployment process from our main branch to our production servers with automatic releases. This means that a Git Flow strategy will only hinder our lead time from user story to production.

# 3 Lessons learned perspective

## 3.1 Evolution and refactoring

## 3.2 Operation

## 3.3 Maintenance