Report - DevOps 2024 ITU

Casper Holten, Mads Andersen, Magnus Kristensen, Mikkel Kristensen, Oliver Laursen

Table of Contents

1. Introduction

During the course DevOps, Software Evolution and Software Maintenance an application stemming from an earlier course has been subjected to a simulation simulating real world usage by users. This has forced us to make the application more durable, more performant, and more maintainable. This has taught us to develop more performant applications, implement robust CI/CD workflows, and to use metrics, monitoring and logging to monitor, and keep our application healthy.

2. System's Perspective[MADS, MAGNUS]

2.1. Design and Architecture of ITU-MiniTwit Systems

2.1.1. Overall Architecture

Figure 1. Architecture Overview

The system architecture depicted in the figure 1 represents a CI/CD pipeline for deploying a software system.

1. **Development and Source Code Management**: Developers work on the source code locally and manage it using a GitHub repository. The repository contains the source code and a Vagrantfile for environment setup.
2. **CI/CD Workflows**: When changes are pushed to the main branch in the GitHub repository, GitHub Workflows are triggered. These workflows, defined in .yml files, handle the build, test, release, and deployment processes.
3. **Containerization and Delivery**: The workflows build the Docker image for Minitwit, before pushing it to Docker Hub. The workflows then pulls the images for Prometheus and Grafana to use in the deployment process.
4. **Provisioning and Deployment**: The deployment process, triggered by GitHub Actions, provisions virtual containers (VContainers) on DigitalOcean. A collection of containers run a Minitwit swarm, while a single other container runs a MySQL DBMS.

Overall, this architecture automates the continuous integration, delivery, and deployment of the application using GitHub, Docker, and DigitalOcean.

2.1.2. Onion Architecture

Figure 2. Onion Overview

As seen in figure 2, the source code consists of the three main pillars: Core, Infrastructure, and Web. This coincides with the Onion architecture with the innermost layer, the domain model, being Core. The next layer, infrastructure, holds the business logic. The outermost layer, Web, contains the user interface. These layers each only depend inwards which has been beneficial for the project as it enforces a clear separation of concerns, which allows for easier management of dependencies and

testing.

2.1.3. Digital Ocean

Figure 3. Digital Ocean Overview

The Digital Ocean architecture, as seen in figure 3, consists of a collection of VContainers. The VContainers are provisioned by the Vagrantfile during the deployment process and run the Minitwit swarm. The swarm is a collection of containers that run the Minitwit application. The swarm is managed by Docker Swarm, which ensures that the application is highly available, load balanced, and fault tolerant. It also makes the system highly scalable, as more workers can be added to the swarm as wished. The swarm is connected to a single container running a MySQL DBMS, which stores the application's data. This architecture ensures that the application can handle increased traffic and continue operating even if some nodes fail.

2.2. Dependencies of ITU-MiniTwit Systems

The ITU-MiniTwit system is built using a variety of technologies and tools, some of which are used during the development process, while others are important at runtime.

- ASP.NET Core: The foundation of our application, used for implementing the web server, user authorization and identification, and our minimal web API.
- Entity Framework: An ORM used in the database abstraction layer and all CRUD operation logic.
- Playwright: A testing library enabling end-to-end (E2E) tests.
- Prometheus: Monitoring system and time-series database for collecting and querying metrics.
- Grafana: A visualization tool for Prometheus metrics.
- New Relic: A tool used for collecting, storing, and visualizing logs.
- Xunit: A testing library for integration and unit testing.
- Coverlet: A tool for determining code coverage in tests.
- Moq: A test library for generating mock objects for unit testing.
- Docker: Used for containerization of the application.
- Docker Hub: A container registry for storing and sharing Docker images.
- Digital Ocean: A cloud provider for hosting the application.
- Vagrant: A tool for provisioning and managing virtual machines.
- GitHub: Used for source code management and task management.
- GitHub Actions: Used for automating the CI/CD workflows.
- GitHub Copilot: An AI-assistant for code completion and generation. Read more about the use of AI-assistants in the section Use of AI-Assistants.
- ChatGPT: An AI-assistant for code completion and generation. Read more about the use of AI-assistants in the section Use of AI-Assistants.

2.3. 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.

2.4. Current State of Systems

3. Process' Perspective [OLIVER, MIKKEL]

3.1. CI/CD Chains

3.2. System Monitoring

3.3. System Logging

We initially tried deploying the ELK stack for logging and monitoring but faced difficulties. We then chose New Relic, which logs everything written to a the node's console, similar to local debugging. It logs stack traces, exception messages, and preceding events when requests fail, allowing us to monitor API access, track server actions, and identify errors.

The aggregated logs were sent to New Relic and were accessible through their proprietary dashboard.

We configured the New Relic agent in our Dockerfile, ensuring consistent logging across all application nodes.

3.4. Security Assessment

The team has uncovered three major security threats via risk assessment. These risks are described in the Risk Assessments chapter of the Appendix. The team has addressed these threats, but due to the time constraint of the project as well as low probability of an attack, the tasks to implement fixes was underprioritized.

3.5. Scaling and Upgrade Strategy

For scaling the application, Horizontal scaling with Docker Swarm was applied, as it offers numerous benefits, including improved application availability, load balancing, and fault tolerance. By distributing containers across multiple nodes, Docker Swarm ensures that the application can handle increased traffic and continue operating even if some nodes fail. This scalability allows for seamless expansion of resources in response to the growing demands during the course. Given the existing use of Docker in the application, adopting Docker Swarm was a logical choice, leveraging

the teams familiarity with Docker's ecosystem while enhancing their ability to manage and scale containerized applications efficiently.

We chose rolling updates for our deployment strategy as it is the default method in Docker Swarm and aligns well with our existing infrastructure. This approach provides continuous availability by updating services incrementally, minimizing downtime without requiring additional resources. The alternative for this strategy is the Blue-Green upgrade strategy, but the additional ressources and implementation time was what additionally made the team favour Rolling Updates. ([szulik_2017])

3.6. Use of AI-Assistants

In this project these AI-assistants were used:

- OpenAI's ChatGPT version 3.5, 4.0.
- GitHub Copilot

The AI-assistants were mainly used for:

- Breaking down code logic. I.g. In order to recreate the Python API controller provided by the course, the code needed to be translated into C# and modified to fit our application. ChatGPT was a great tool for understanding each endpoint and what data would be included in a call and a response.
- Code completions. GitHub Copilot acted as an extension of IntelliSense, in the sense that it could auto-complete simple pieces of code, such as loops, if-statements, and method signatures.
- Research. ChatGPT was also used to provide a secondary explanation when researching new technologies, in situations were the documentation either was difficult to understand, or if subsidary information was needed.
- Stacktrace breakdowns. ChatGPT was used to breakdown stracktraces, summarizing the information as well as providing a more user-friendly format to read.
- Identify functions that could be made more performant.

Downsides of using AI-assistants:

- Both ChatGPT and Github Copilot are flawed, which makes them unreliable tools. Sometimes it would take as much time to doublecheck the output of an assistant as would have to complete the task without it, which defeats the purpose of using them.
- If used without careful inspeciton of the provided code, the LLM is likely to introduce bugs into the application. This is due to the fact, that LLM's have a difficult time understanding the context in which the requested code is supposed to operate.

4. Lessons Learned Perspective [EVERYONE]

4.1. Evolution and Refactoring

4.2. Operation

4.3. Maintenance

4.3.1. Challenges

One of the primary challenges encountered during the maintenance phase was identifying the precise elements requiring upkeep. This encompassed several aspects:

- **Error Detection**: Determining the root causes of errors in the system.
- **System Status**: Monitoring the status of the website, including instances of downtime.
- **Issue Diagnosis**: Pinpointing specific failures or malfunctions within the system.

Initially, without proper tools, these tasks appeared daunting and time-consuming. The lack of convenient visibility into system performance and error tracking contributed to uncertainty regarding the system's health. In the beginning, we relied mostly on the general monitoring provided by the course.

4.3.2. Solutions Implemented

To address these challenges, we integrated monitoring and logging solutions into our maintenance workflow:

1. **Monitoring Systems**:
    - Implemented real-time monitoring tools, Prometheus and Grafana, to continuously observe website performance and availability.
    - Configured alerts in Grafana to notify the team of any anomalies or downtime events.
2. **Logging Mechanisms**:
    - Established detailed logging processes using New Relic to systematically record all system errors and events.
    - Utilized the centralized logging platform in New Relic to aggregate and analyze log data, facilitating quicker diagnosis and resolution of issues.

These tools significantly enhanced our ability to manage and maintain the system effectively. Real-time insights and detailed logs provided a clearer picture of the system's operational state, enabling proactive maintenance and faster response times.

4.3.3. Outcomes

While the introduction of monitoring and logging tools did not entirely eliminate maintenance issues, it considerably reduced their complexity. Key improvements included:

- **Improved Error Tracking**: Enhanced ability to trace and resolve errors promptly.
- **Proactive Maintenance**: Ability to detect potential issues before they escalated.
- **Efficient Response**: Faster response times due to real-time alerts and comprehensive log data.

4.3.4. Lessons Learned

- **Scope of Maintenance**: Maintenance is inherently a substantial and ongoing task that demands continuous attention and resources.
- **Utility of Monitoring and Logging**: Effective monitoring and logging are critical components of a robust maintenance strategy. They provide essential visibility into system operations, aiding in quick issue identification and resolution.
- **Threshold-Based Alerts**: Implementing threshold-based alerts is vital for timely intervention, preventing minor issues from escalating.
- **Continuous Improvement**: Maintenance processes should be continually refined and improved to adapt to evolving system requirements and comming challenges.

5. DevOps Style of Work

As the entire team has been taking the course "Second Year Project: Software Development in Large Teams" which introduces working by the Agile principles and with Scrum as a framework, it's only natural that some elements have been taken into the project especially since these frameworks align well with the DevOps style of work as shown in table 7 of ([jabbari_2016]).

The effects of learning Scrum seeped into the working style of the team, not by introducing scrum events and the like, but by using the 3 pillars of Scrum; Adaptation, Transparency, and Inspection ([scrum_guide_2020]) as guidelines. Each Friday the team held physical meetings, where the state of the project was discussed, keeping each member up to date while answering questions any member might have. Breaking down the work each week, increased understanding of the project, transparency, and ensured openness amongst the team. GitHub allowed for fine-grained inspection through peer-reviewed code inspections facilitated with Pull requests. GitHub also provided a Kanban board to showcase the backlog, as well as the status of ongoing work.

In the same way the agile principles were introduced to the project. Of the twelve principles; "Welcome changing requirements" ([agile_principles_2001]), was the most prevalent as new requirements were added almost weekly. Furthermore how to meet those requirements wasn't set in stone. In situations where the team would find a better way to fulfill a task, there would be little resistance to incorporating it into the project.

Another vital principle was; "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation". To implement this principle, the team had both the weekly physical meeting, but would routinely also hold pair-/ or mob-programming sessions. The latter part, contributed to increasing the ownership of the codebase, generally raises the quality of the code produced, and minimises the time spent on code inspections. **FIND STUDY THAT SHOWS BENEFITS OF PAIRPROGRAMMING**

6. Conclusion

A: Appendix

A.1. Risk Assessments

A.1.1. HTTP Transfer Protocol

Risk Identification

Assets:

This threat concerns the Web Application, as well as services that communicate over the Web Application.

Identify threat source:

- The Web Application uses HTTP as Transfer protocol

Construct risk scenarios:

A malicious person gains access to a session, and from there have multiple ways to cause harm;

- As the messages aren't encrypted in HTTP they can eavesdrop on messages sent between a client and the server.
- They can create man-in-middle attacks, potentially tampering with the data sent between server and client.

Risk Analysis

Determine likelihood:

As the team consists of relatively unknown developers, and the project is a course-project with no real users or data, the motivation for attacking the system is low. However there are multiple guides online on how to commit such an attack, making it accessible for any user with basic knowledge about Network communication.

Improve the security of our system:

The best course of action would be set up the HTTPS for the web application and redirect the Users to that endpoint. This requires that a SSL certificate gets registered an activated.

A.1.2. Database Credentials

Risk Identification

Assets:

This threat concerns the Database, and the nodes which the database is hosted on.

Identify threat source:

- The credentials for the database is saved in a ".env"-file, which is distributed to any node on which the database runs.

Construct risk scenarios:

A malicious hacker forces himself into the filesystem of a Node, there they could find ".env" file, giving them multiple options of causing harm including;

- Dropping the database
- Holding the data ransom
- Tampering with the User's data
- Utilizing User data to cause problems for the Users
- Leaking data

Risk Analysis

Determine likelihood:

As mentioned in the previous risk assessment HTTP Transfer Protocol, the motivation for causing such an attack is low. Compared to the previous threat, this attack requires more orchestration and skill as the hacker would have to gain access to the Node, and know what to look for.

Improve the security of our system:

There are multiple options to remove this threat, such as any service providing 2FA for secret repositories, like Docker vaults.

A.1.3. Database back-up

Risk Identification

Assets:

This threat concerns the database.

Identify threat source:

- There's no virtual or physical back-up copy of the state of the database.

Construct risk scenarios:

- In lue of the threat from Database Credentials, there wouldn't be a way to restore data if a person with malicious intent gained access to a database node, found the credentials and removed data.

Risk Analysis

Determine likelihood:

The likeness of this happening, would be the same as for the Database Credentials threat.

Improve the security of our system:

There are many options as to how to improve on this threat. A minimum effort would be to have physical copy of the state of the database on one or more harddrives. DigitalOcean has a Collaboration with SnapShooter, a service that enables virtual backups of databases, that would integrate nicely into the project.

References

Szulik, Maciej. "Colorful deployments: An introduction to blue-green, canary, and rolling deployments." *Opensource.com*, 2 May 2017. https://opensource.com/article/17/5/colorful-deployments

Scrum guide, 2020. https://scrumguides.org/scrum-guide.html

The Agile Manifesto, 12 principles, 2001. https://agilemanifesto.org/principles.html

Jabbari, Ramtin. "What is DevOps? A Systematic Mapping Study on Definitions and Practices.", 2016 https://www.researchgate.net/publication/308857081_What_is_DevOps_A_Systematic_Mapping_Study_on_Definitions_and_Practices