Large Systems Exam Synopsis

By Kristofer, Mads, Michael and Søren

Github Repository

Intro

This report presents the development and delivery of an integrated software system for the MTOGO food delivery business. The project serves as the final exam for the Development in big systems course. By following Agile methodologies, incorporating advanced architectural patterns, and utilizing quality assurance practices, we ensured the system meets business needs while prioritizing scalability, maintainability, and user satisfaction. The report outlines our approach to meeting course-specific requirements, including security, continuous integration/delivery (CI/CD), and scalability strategies.

Agile Methods and Framework:

In our development process we used agile software development methodologies, such as extreme programming (XP) and frameworks like Scaled agile framework (SAFe), empowered us to deliver MTOGO iteratively and adapt to changes when required. Agile emphasises on adaptability, collaboration and customer feedback.

XP is a methodology focused on delivering high-quality software and quickly responding to changes. They focus on several XP Core practices, the ones we use include TDD (Test driven Development), Pair Programming, Continuous integration, code Refactoring and coding standards. Xp Emphasizes collaboration and customer involvement. While we did not have a direct customer for the exams project, it did help in collaboration between teammates.

SAFe is a framework designed to implement Agile at a larger scale. It aligns with business objectives though structuring hierarchy and ceremonies like program increments and Agile release trains. This method ensures that projects are focused and teams can coordinate across businesses.

For managing the project we used Github project as our Kanban board, this helped us manage the tickets/tasks/issues and allowed us to focus effectively on implementing features. During the planning phase we made user stories, these stories were then converted into milestones and issues in the backlog. When the issues needed to be prioritized they were pulled into the Todo column for the sprint.

Balancing cost, Quality and resources:

- **Cost**: Agile reduces costs by eliminating waste, prioritizing high value features and minimizing rework with changes. XP's focus on clean code reduces technical debt, while SAFe integrates financial consideration into the planning to optimize resources.
- **Quality:** XP's TDD and continuous integration ensured that we could deliver high quality. While we incorporated SAFe for automated testing and retrospective to improve quality.
- **Resources:** We used agile to promote cross-functional teams for optimal collaboration. While SAFe adds strategic resource allocation to ensure organizations meet market demands effectively.

CI/CD:

At the start of our project we used Digital Ocean to set up a simple droplet for our application to run on. We created a yaml file for our pipeline that would run in GitHub Actions when we choose to update a branch named “deploy”, so that we had more control over when our deployment were updated.

Our pipeline was made at the start of the coding part of our project, so that we could test while we coded to ensure what we made were working as expected when deployed. It was made before we started working on using Kubernetes and Docker compose, and so it only builds individual images of the frontend and the backend.

With our droplet breaking due to memory problems, we choose to not update it and instead focus on ensuring that our local deployment would run using Kubernetes and Docker compose.

Our yaml file, “testBuildDeploy.yml”, is split into three parts, “Clean”, “Frontend” and “Backend” “Clean” was meant to remove previously stored images on docker. For the “Frontend” section we first would run our automated test using Jest using the command “npm test”, after which we would build the image and upload it to the droplet using ssh and lastly we would use ssh to run commands that would run the frontend using nginx. Lastly for the “Backend” we did mostly the same as “Frontend” with the difference being not having any commands regarding nginx.

The advantage for our pipeline was how easy we could update our deployment. It made it possible for any group member to update the deployment despite not needing to remember all the steps and ensuring everything is up to date. By using a specific branch we also had more control so that when we were working on the droplet we didn’t need to worry about something breaking.

The main disadvantage of our pipeline was that we lost oversight of our deployed application on our droplet which resulted in our droplet breaking overtime.

Understanding of scalability:

Kubernetes - Usage:

We choose Kubernetes over Docker Swarm because while Docker Swarm is a more simple tool, Kubernetes have a better focus on scaling which we saw as the main reason for using either of them.
We started by creating an overall Docker compose file for the entire project after which we used a tool called kompose to convert the compose file into a single kubernetes manifest file. Both of which are made using yaml files. From there we made some modifications to the manifest file. For our frontend and backend we needed to specify to never pull the image and the specific name of the built image created when using the command “docker compose build”. Lastly we needed to open up the applications to the local network including specifying the use of a load balancer which would enable us to have multiple instances of our backend that runs on the same port.

JMeter - Testing:

Using JMeter we would be able to test our deployment against a large number of users with four different numbers of users being given in the work assignment, a minimum and maximum for now a minimum and maximum for five years ahead. Our results included testing our payment service which even at the lowest number of users would result in 100% failure rate which we believe to be from Stripes api and not an internal error on our application.
It also shows that the main reasons for failure are connection related, that the connection is being refused due to the maximum number of connections to a single port which highlight that we need to make use of a range of ports instead of a single port.
For the connections that aren’t refused we see a high average request time, however this result may be worse considering we needed to run the test locally alongside the deployment which resulted in a 100% usage of CPU which would have slowed our backend which would increase the request time.

Terraform - Discussion:

Terraform’s core features include infrastructure automation, resource management, dependency resolution, and execution plans. It creates cloud-native environments for provisioning infrastructure, managing scaling, and implementing disaster recovery strategies.
It has, however, a rather steep learning curve which can make it hard to use and which was also the reason we did not believe we could reasonably make use of it within our project.
If we had made use of Terraform for our droplet deployment it would also most likely have been able to recover from the memory problem.

Quality measurement:

Planning Techniques:

Event Storming and Domain-Driven Design (DDD) are key to ensuring quality in our development. Event Storming helps us map out how the system works by breaking it into steps and user roles. This makes it easier to identify the services we need and the user stories with clear acceptance criteria, helping us stay on track. It also lets us focus on the most important parts of the system, like the customer and admin flows, within our time limits.

DDD helps divide the system into smaller parts, each with its own model, making it easier to understand the business needs. It uses the business language, which makes communication clearer between developers and stakeholders. However, the system diagrams can get outdated during development, and they don't always show all the details, like variable types, which can cause issues later. Together with best practices like code reviews, pull requests, and CI/CD, these methods help maintain quality throughout the development process.

User stories were crucial in guiding the design and development process by providing clear, concise descriptions of features. By outlining specific goals, acceptance criteria, and priorities, we focused on delivering value to end-users. By connecting user stories to the events identified in Event Storming, we could easily define the required services and track progress. This alignment helped us manage time constraints by prioritizing essential flows, such as customer and admin operations, while ensuring all requirements were met.

By incorporating these planning techniques alongside best practices like branching and pull requests, we improved team collaboration and code quality. Using Domain-Driven Design (DDD), we broke the system into smaller, manageable parts (bounded contexts) and used the business language to bridge the gap between technical and non-technical stakeholders. CI/CD tools automated quality checks and deployments, ensuring a robust, well-tested system. Together, these practices supported quality assurance and fostered a resilient development process.

Best Practices:

Branches and GitHub Project.

We used GitHub Project to keep an overview of our implementations as well as ensuring that any group member were only working on one implementation at a time. When a group member would start implementing an issue they would first create a new branch from the issue which ensures correct naming.

Eslint:

We used Eslint because of its integration with our IDE so that a general coding style was upheld by all members of our group.

Models:

Before we started coding we first created multiple diagrams such as domain model, use case, event storming, and based on these models we created user stories with acceptance criteria for each of the four different roles that a user could have.

Error handling and logging:

Error handling, logging, and metrics analysis are vital for building high-quality, reliable applications. Together, they provide insights into system behavior, enable efficient debugging, and support proactive performance optimization.

Error Handling

In our implementation, middleware captures errors, logs details such as messages and stack traces, and provides uniform client responses, improving stability and debugging efficiency. For when our applications crash we have set the restart policy in our kubernetes manifest file to always restart.

Logging

Logging tracks critical events like HTTP requests, errors, and system actions, using structured fields such as timestamps, hosts, and metadata. Winston categorizes logs by levels (eg. info, error). These logs help us identify issues, trace their origins, and maintain an audit trail for accountability.

Metrics Analysis

Metrics analysis complements error handling and logging by offering a broader view of application performance. Tools like Prometheus and Grafana collect and visualize data such as:

- **Number of HTTP Requests Over Time**: Tracks traffic patterns and identifies unusual spikes.
- **Response Times**: Monitors application latency to ensure optimal user experience.
- **Error Rates**: Helps detect recurring or systemic issues.

By analyzing these metrics, we can identify bottlenecks, monitor trends, and make data-driven improvements to the application.

Impact on Quality

1. **Faster Debugging**: Logs and metrics provide detailed insights into application behavior, reducing resolution time.
2. **Proactive Monitoring**: Metrics identify potential issues (e.g., performance degradation) before they escalate.
3. **Auditability**: Logs and metrics ensure accountability for actions like logins, transactions, and permission changes.
4. **Enhanced CI/CD Support**: Logging aids in identifying deployment issues, enabling smoother development workflows..

In summary, error handling and logging (via Winston) work in tandem with metrics analysis (via Prometheus and Grafana) to enhance application quality. This combination ensures transparency, accountability, and actionable insights for maintaining and improving system performance.