Group Q DevOps report

Course code: KSDSESM1KU

Michael Jeppesen `micje@itu.dk`
Marcin Sroka `msro@itu.dk`
Alin Plamadeala `alpl@itu.dk`
Philip Blomholt `phbl@itu.dk`

May 15, 2025

Contents

1 System perspective **3**
1.1 Design and architecture 3
1.2 Technology Stack and Motivations 5
1.3 Important interactions 6
1.4 Current state 7

2 Process perspective **8**
2.1 Stages and tools in CI/CD 8
2.2 Monitoring 9
2.2.1 Metrics 9
2.2.2 Logging 10
2.3 Security assessment 11
2.4 Scaling strategy 12
2.5 Use of Artificial Intelligence 14

3 Reflection perspective **14**
3.1 Software Evolution and Refactoring 14
3.2 Operation 14
3.3 Maintenance 15
3.4 Our approach to DevOps 15

1 System perspective

1.1 Design and architecture

The MiniTwit project is built using a single cloud-based Virtual machine (`droplet`) with 1 CPU, 2 GB RAM, and 70 GB Storage, running 9 Docker containers in a Docker swarm setup. Three of these containers contain the MiniTwit replicas, written in Go with the `Gin` framework for web development. The decision to use Go was based on a desire to learn Go, but more importantly, due to the low memory usage and high efficiency of both Go and Gin, when compared to similar web frameworks and languages, such as C# ASP.NET and Python using Flask (Tien Do Nam, 2025). The MiniTwit application utilizes a cloud-based managed PostgreSQL database, since it allows the database to be retained even if the droplet is replaced. Additionally, to retain data and configuration files independently of droplets, an externally mounted volume with 20 GB of storage is required. Figure 1 depicts the complete setup of deployed services.

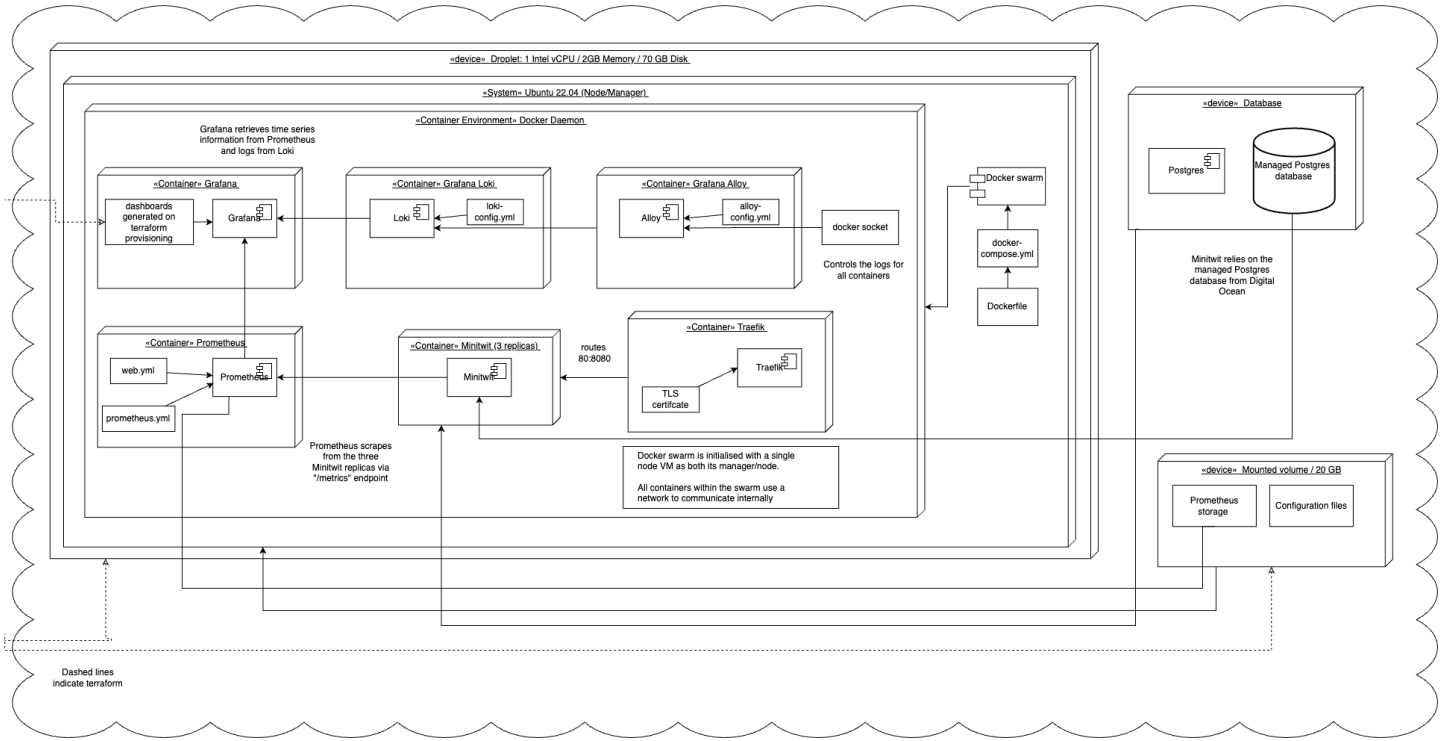

Figure 1: Deployment Diagram over MiniTwit project. On the right-hand side, there are some dashed lines, which indicate provisioning from Terraform, which is used for dashboard visualisations, and setting up a droplet, a database, and a mount.

1.2 Technology Stack and Motivations

When making choices for the MiniTwit technology stack, we emphasised tools that allow for automation, declarative configuration, and scalability.

- **GitHub with GitHub Actions, Projects, and Packages**: Serves as our central hub for version control, CI/CD pipelines, project management, and artifact repository. We chose GitHub because of its seamless integration with container registries, generous CI/CD minutes for public repositories, and robust ecosystem support. The integrated issue tracking and project boards eliminate the need for separate project management tools.

- **Terraform**: Manages infrastructure as code across multiple cloud providers. We selected Terraform because of its cloud-agnostic nature, allowing us to use the same tool for both deploying the infrastructure on DigitalOcean and the Grafana configuration. Its declarative syntax and state management prevent configuration drift that often occurs with imperative deployment scripts.

- **Docker**: Provides containerization for consistent deployment environments. Docker, together with Docker stack and Docker swarm, allowed us to express all of our services in a single file and manage all the communication and integration between them. It ensures our application runs identically across development, testing, and production.

- **DigitalOcean Spaces**: Handles log storage and terraform remote state. This solution was selected because it provides object storage compatible with S3 and also integrates seamlessly with our Alloy log service.

- **PostgreSQL**: Serves as our primary database. We chose PostgreSQL because it's a time-tested technology, open-source and provides great performance. PostgreSQL is also supported by DigitalOcean managed database service which decreased our operational burden.

- **Traefik**: Acts as our reverse proxy and SSL termination endpoint. Traefik was preferred over Nginx because of its automatic service discovery in Docker Swarm environments, through labels and a rich set of features. It also has built-in Let's Encrypt integration, which eliminates the need for manual certificate management.

- **Grafana**: Provides comprehensive visualization and alerting for system metrics. Grafana's extensive plugin ecosystem and multi-data source support allowed us to integrate all of our observability tools into one place.
- **Alloy**: Collects and forwards logs. Alloy was chosen because it is part of the Grafana ecosystem, maintained by the same organization and allows building complex log forwarding pipelines using a simple declarative syntax.
- **Loki**: Aggregates and stores application logs, and similarly to Allow, is part of the Grafana ecosystem, thus offering an effortless integration with Grafana. It was chosen for its simple setup and support for various external storage, including S3.
- **Prometheus**: Handles metrics collection. Prometheus is the industry standard for container-based monitoring, offering a powerful query language (PromQL) and a pull-based collection model that reduces operational complexity.
- **Golang**: Powers our backend application with excellent performance characteristics. Go was selected for its superior concurrency support, fast compilation times, and single binary deployment model. The single binary allowed us to build Docker images from scratch, achieving very small images and fast deployments.
- **Playwright**: Provides end-to-end testing capabilities. Playwright offers more reliable testing than Selenium-based solutions with built-in wait strategies and cross-browser support. We used TypeScript for defining the test suites and consider the developer experience offered by Playwright superior to Selenium.
- **Ansible**: Manages server configuration and application deployment. Ansible provides clear, version-controlled infrastructure configuration. Initially, we were using startup scripts for setting up the VM environment, but switched over to Ansible to be able to apply changes to the VM without having to re-create it.

1.3 Important interactions

Whenever a user opens or tries to interact with the web-application in any way, the control flow follows figure 2. This applies both to actual users and

API requests that are made against the application. There is, however, a small change if it is an API request; instead of looking for the individual user, it instead tries to validate the API-key.

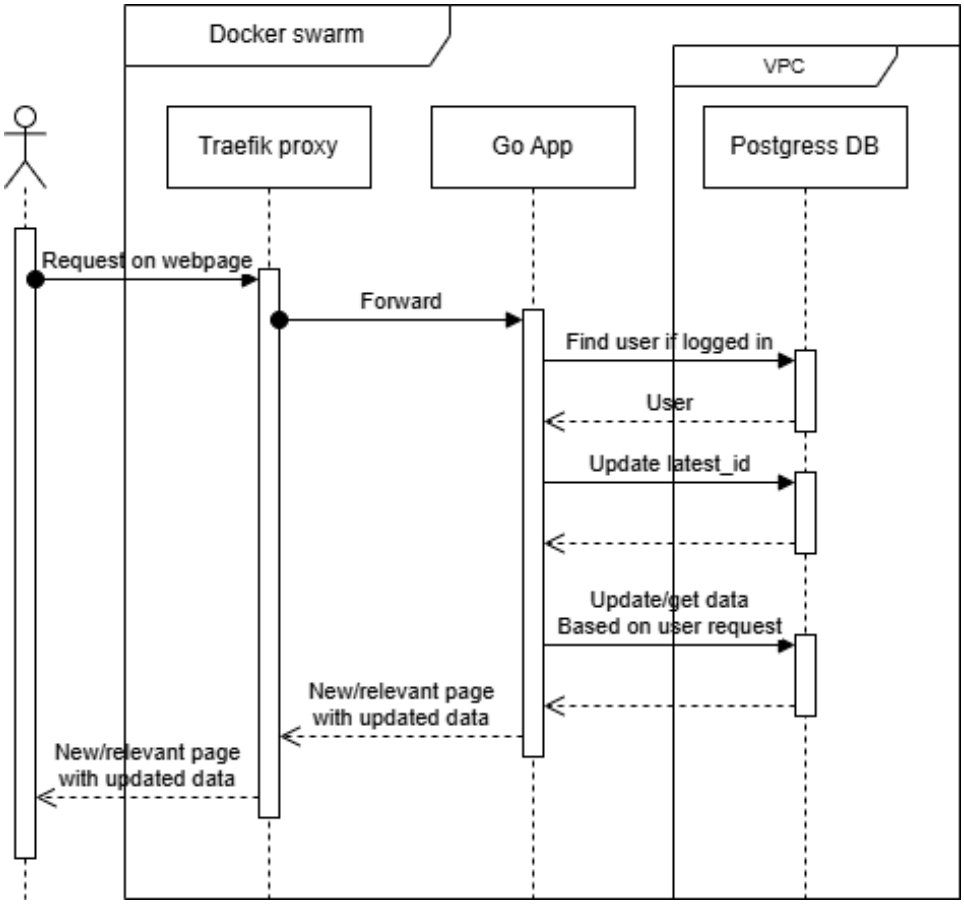

Figure 2: User sequence diagram

1.4 Current state

We used several static analysis tools and tests to keep the code in good shape. The current, most recent implementation is working correctly and passing all tests, shown in figure 3.

Below is an overview of two static code analysis tools applied to the current implementation:

- SonarQube: 3.3k lines of code, 0 security issues (grade A), 5 reliability issues (grade C), 17 maintainability issues (grade A), 2.3% code duplication (Below the suggested 3% threshold), 237 cyclomatic complexity.
- CodeClimate: maintainability grade A, 5 code smells, 2 duplications, 0 other issues.

2 Process perspective

2.1 Stages and tools in CI/CD

We have 3 pipelines that are triggered by various events in our version control system:

- Committing to feature/bugfix branch.
- Merging pull request from feature-branch to develop-branch.
- Merging the develop-branch into the main-branch.

Furthermore, we have set up some rules for protecting the main-branch.

Feature/working-branches

The workflow shown in Figure 3 is triggered on every commit, and the status check is used to decide whether a PR to develop/main can be merged.

Merging PR into develop-branch

A merge into this branch triggers a build with DEV environment variables, which is then deployed to our development environment.

Merging PR into main-branch

The workflow triggered by a merge into the main-branch is almost the same as when merging into the development-branch. A build is made with production environment variables, and it is deployed to our PROD environment. When a new release is successfully deployed, a release is made on GitHub.

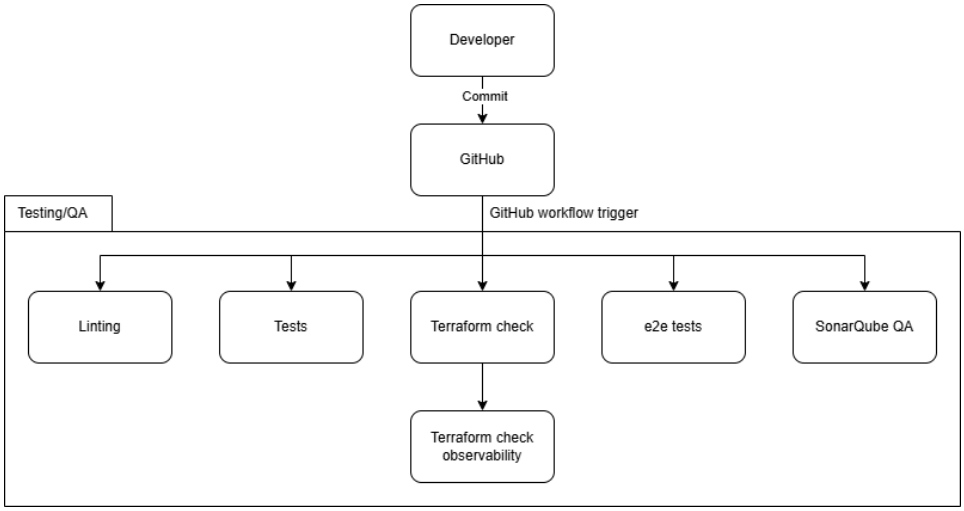

Figure 3: Testing workflow

GitHub rule sets

We have 3 rules specified on GitHub to secure against potentially broken, as a release is made whenever anything is merged into main.

- The first rule is that it is only possible to merge into main from the development-branch. This is done such that our automatic tests cannot be circumvented.
- The second is that pull-requests are only allowed to be merged into develop if all of our tests pass.
- The final rule is that it is not allowed to commit directly to our main- or develop-branch, but instead always require a pull-request, such that the correct workflows are triggered.

2.2 Monitoring

Our monitoring efforts were displayed in Grafana, as can be observed in Figure 4.

2.2.1 Metrics

Our metrics collection was done using the web-scraper Prometheus, which collects metrics from the MiniTwit endpoint "/metrics". The metric end-

point was set up using the `ginprom` package in Go, which seamlessly integrates with `gin`, providing a multitude of information about the state of the MiniTwit app, `gin`, and Go runtime. Prometheus maintains its own storage locally, and to ensure that replacing the virtual machine does not lose its time series data, we created a dedicated volume in Digital Ocean to store this data, alongside the configuration files for Prometheus.

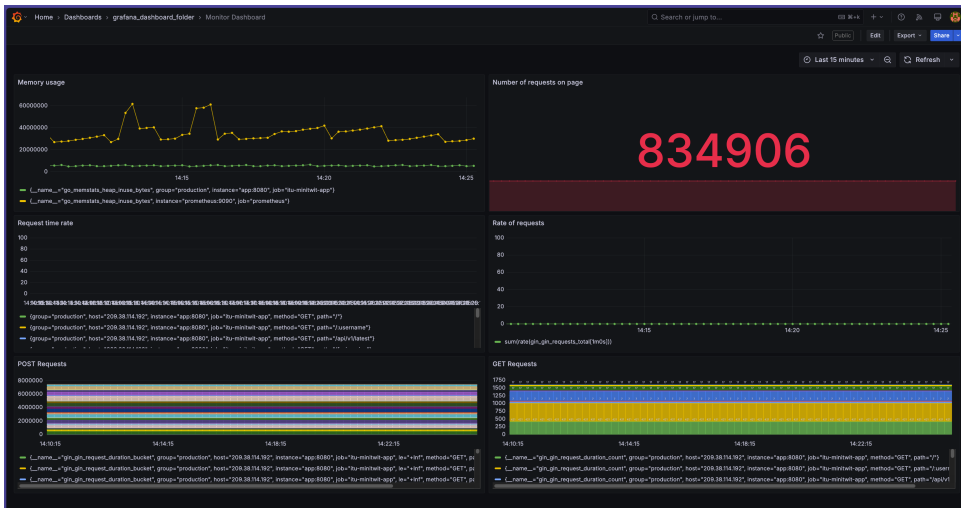

Figure 4: The monitoring dashboard, showcasing the number of requests being made, on which endpoints, and their frequency

2.2.2 Logging

Our logging setup has been based around Grafana Loki as a log aggregation tool, and Grafana Alloy as the log collection tool. Our logging efforts were divided among our Docker containers, reading their respective logs. We decided to split up the dashboard, such that we could see the logs for each Docker container, as observed in figure 6. In addition to this, we implemented logs for our MiniTwit application using the standard logging library in Go `slog`. Here we focused on implementing log events for all of our event handlers providing using different log levels to signify the nature of the problem.

```go
func FollowHandler(c *gin.Context) {  1 usage   Sir Eraisuithon +2
    log := logger.Init()
    db := c.MustGet( key: "DB").(*gorm.DB)
    username := c.Param( key: "username")

    userLoggedIn := utils.GetUserFomContext(c)
    if userLoggedIn == nil {
        log.Error( msg: "[FollowHandler] Error message: User not logged in")
        c.AbortWithStatus(http.StatusUnauthorized)
        return
    }

    var userToFollow models.User
    if err := db.First(&userToFollow, models.User{Username: username}).Error; err != nil {
        log.Error( msg: "[FollowHandler] Error message: %v", err)
        c.AbortWithStatus(http.StatusNotFound)
        return
    }
```

Figure 5: Example of logging with FollowHandler showcasing how errors are propagated as a log.

Figure 6: Logging dashboard, showcasing the events by Docker containers

2.3 Security assessment

To identify possible attack surfaces, we made a full port-scan of our server with nmap, which came up with the following open ports;

22 SSH into droplet

80, 443 web-app through traefik

3000 Grafana

3100 Loki

12345 Alloy

9090 Prometheus

We reasoned that the following were already secure: SSH, MiniTwit, Grafana, and Prometheus. That left Loki and Alloy, where data could be extracted without requiring any form of authentication, which can be seen in table 1.

Risk	**Likelyhood**		
	Low	Medium	High
Low	MiniTwit		
Medium		Grafana, Prometheus	Loki, Alloy
High	SSH		

Table 1: Risk matrix

Our solution to this was to stop exposing the ports for Loki and Alloy, as they are on an internal network in the Docker Swarm, and all relevant data from them can be obtained through Grafana, where authentication is required.

The reason our database does not show up as a potential attack surface is that it is running inside a VPC without public internet access, such that only our application has access to it.

2.4 Scaling strategy

Our project implements a containerized architecture using Docker Swarm for orchestration, with a setup that provides several key scalability advantages:

Container Orchestration with Docker Swarm

- All the services are deployed as a Docker stack, configured in a single `docker-compose` file, which allows for declarative configuration and simplified management
- Docker Swarm provides built-in service discovery, load balancing, and rolling updates

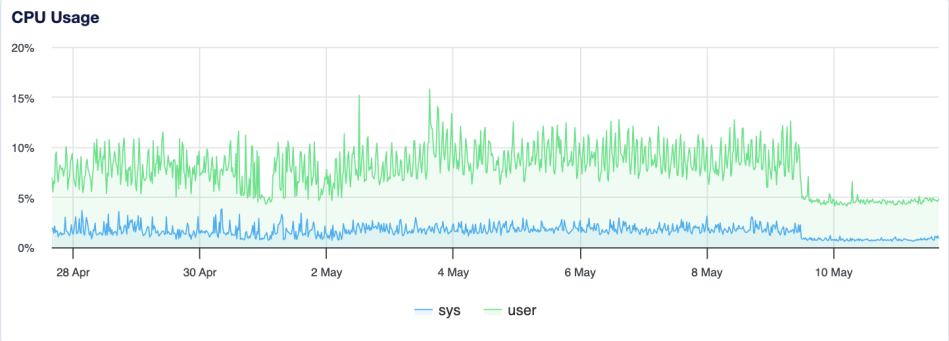

Figure 7: VM CPU Usage graph

- The current configuration maintains 3 replicas of the main application to support high availability, redundancy, and rolling updates

The Docker swarm cluster currently consists of just a single node, because additional nodes are not necessary at the current load levels, the load on the server reaching only 15% of the capacity, as can be seen in figure 7, even at the highest levels of load from the simulator. For the database, we made the choice to use Digital Ocean's managed database service, which provides a fully managed PostgreSQL database. This choice was made to simplify database management and ensure high availability and scalability. The managed offering from Digital Ocean allows us to offload the database administration tasks and has built-in scalability and reliability capabilities, such as:

- Adding standby nodes for high availability
- Deploying read-only nodes for scaling read operations
- Automated backups and point-in-time recovery

The architecture we've implemented provides flexible scaling capabilities through both vertical and horizontal approaches. Vertical scaling can be achieved by upgrading to more powerful VM instances with increased CPU, memory, and storage resources. For horizontal scaling, Docker Swarm's orchestration allows us to seamlessly add additional nodes to the cluster and increase the application replica count. At the same time, Digital Ocean's managed PostgreSQL service enables horizontal read scaling through read-only database nodes. This dual-scaling capability creates a robust foundation for future enhancements, including automated scaling based on load

metrics and multi-region deployments for improved global performance and redundancy. Our infrastructure design emphasizes scalability as a core principle, ensuring the system can efficiently adapt to growing demands without significant architectural changes.

2.5 Use of Artificial Intelligence

In our project, we have used Anthropic's artificial intelligence tool *Claude* (Anthropic, 2025). Claude has been helpful when it came to adopting new technologies and frameworks that none of us had worked with prior. A major concern however, has been that Claude often times seems to miss the broader context, we are working on, ignores security concerns, and generally tends to overcomplicate its output. As such, while the use of AI has been helpful in providing an understanding of the different components in our project, it has not been able to meaningfully replace the work of us as developers.

3 Reflection perspective

3.1 Software Evolution and Refactoring

We learned about tools that assists with refactor software to a later major version, such as the python library (`2to3`), and static analysis tools, such as `shellcheck` and linters. These tools are invaluable for ensuring that code follows best practices and remains well-maintained. Equipped with the awareness of these tools, we are able to keep all future projects in good shape. Additionally, with the use of these tools, we were able to work more efficiently and spend less time on code reviews, since these tools directly enforce a certain convention in the CI chain

3.2 Operation

Throughout the project, we learned a major lesson about software. Namely, the key to success is automation and fast feedback loops. If we automate testing, integration, and deployment, then the friction to get from a finished software update to a deployed, working service is minimal. This encourages software evolution in small steps, which in turn makes it much easier to spot bugs and find their cause, and bugs are also found a lot earlier.

At some point, we found that we didn't set up authentication at the front of Loki and Alloy, possibly allowing unauthorized access. We then realized that Loki and Alloy are contained as part of the same network and don't

need to expose ports, so we stopped port exposition for them, preventing unauthorized access.

3.3 Maintenance

Our system maintenance strategy focused on proactive monitoring and efficient issue resolution. To keep track of any problems, we used our Grafana dashboards, the unified dashboard (itu-devops, 2025, "Status for All Groups"), and centralized logging with Loki to identify potential issues. The simulator's continuous load testing provided additional validation, with the low number of reported errors confirming our system's overall stability in production.

Despite the general reliability, we encountered occasional challenges when implementing infrastructure changes. For example, when optimizing our Docker container (Issue #250), we migrated to a minimal `scratch` base image to reduce size and improve security. This revealed a dependency issue where our application could no longer read and write to the filesystem to track the latest message ID, as the `scratch` image lacks filesystem operations. We resolved this by migrating this functionality to a single-row database table, better aligning with stateless container principles.

We also addressed user experience issues, such as timestamps displaying in UTC rather than users' local time zones (Issue #322). Beyond implementing the immediate fix with client-side time zone conversion, we added end-to-end tests simulating requests from different time zones to prevent regression.

Throughout the project, we continuously improved our maintenance processes by adding automated tests and monitoring for issues we encountered. This allowed us to be proactive rather than reactive, with many potential problems being caught during continuous integration before reaching production.

3.4 Our approach to DevOps

Throughout this project, we have sought to implement several DevOps practices to increase the efficiency of our workflow. Here, our initial goal was to reduce the amount of time between committing code and code being reviewed and subsequently merged, which we found to be a tedious process. One of the core ideals of DevOps is being able to reduce this lead time between committing and merging, such that developers can work faster and in smaller batches while still maintaining high code quality (Kim et al., 2016).

We achieved this principle through the implementation of continuous integration, providing a *standardisation* measure for our coding quality using linters and tests as a requirement for merging a commit. This provided a much lower lead time, since code reviews were no longer being done before a merge, instead requiring our testing suite to pass. This fostered a culture where each of us would be able to work on a task without any major repercussions, thereby increasing both risk-taking and experimentation. Another aspect is that DevOps relates to our ability to automate repetitive tasks in an effort to reduce the time spent on a task, but in extension also avoid human errors that might occur when multistep workflows are required.

References

Anthropic. (2025). Claude. https://claude.ai/new

itu-devops. (2025). Status for all groups. http://206.81.20.113/status.html

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The devops handbook: How to create world-class agility, reliability, and security in technology organizations*. IT Revolution Press.

Tien Do Nam. (2025). Sharkbench: Benchmarking programming languages and web frameworks.