Modernizing Legacy Systems through Scalable Microservices and DevOps Practices

Daniel Thomas
daniel.cromwel@gmail.com

***Abstract*—A legacy monolithic application was successfully refactored into a containerized microservices architecture, utilizing modern DevOps practices to enhance scalability, maintainability, and security. The original system, built with Flask, Python2, and SQLite, was re-engineered with an ASP.NET Core backend, a ReactJS frontend, and a PostgreSQL database, following Clean Architecture and SOLID principles. This architecture was deployed on a managed Kubernetes cluster, enabling horizontal scaling and fault tolerance. Key features such as automated SSL encryption, secrets management, and Prometheus-based monitoring were integrated to improve security and observability. The refactor not only supports efficient continuous delivery and infrastructure automation but also leverages microservices for independent scaling, technology diversity, and rapid feature deployment, demonstrating the powerful synergy between modern application design and DevOps principles.**

I. System's Perspective

A. Overview

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

- backend using the *ASP.NET Core web framework* and *EF Core* object-relational mapping.
- *ReactJS* single-page application frontend.
- *PostgreSQL* database.

Upon which a monitoring stack and temporarily a logging stack were added to serve as a containerized application behind a load-balancer on a managed *Kubernetes* cluster (*DOKS*)

B. System Design

The following section discusses our system design by supplying diagrams to illustrate the designs. The backend, whose decomposition can be seen in Figure 1, is the most crucial part of the system as this is the part that contains our logic and API endpoints for the simulation. Some files have been excluded from the diagram to simplify it. These files include Dockerfiles, App settings files, and the program.cs file, which is the system's entry point, a legacy controller, and the interface for the front end. It is now only used in tests.

When we refactored the system, we decided to change the architecture to follow The Clean Architecture guidelines introduced by Robert C. Martin [3]. We wanted the codebase to be easier to maintain, test, and scale and, by that, avoid technical debt. This is achieved by following The Dependency Rule [3], which states that source code dependencies can only point inwards.

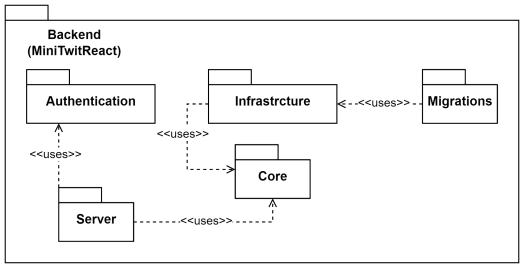

Fig. 1. Decomposition of Backend Package of the System

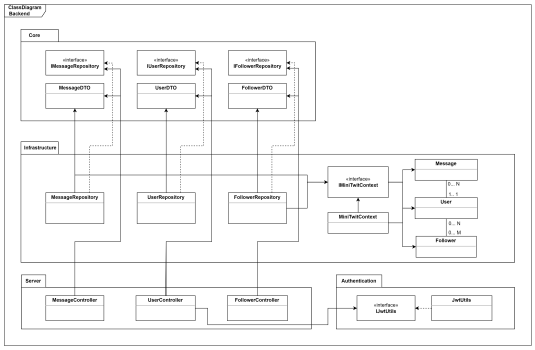

Fig. 2. Class Diagram Illustrating The Clean Architecture-pattern.

In figure 2, we can see the new refactored design: all dependencies point towards the core package, and each lower-level tier only has dependencies pointing towards a tier closer to this package.
Note that each entity has a separate implementation of the repository-DTO- and controller-class; this complies with the single responsibility principle. Additionally, we only depend on abstractions, i.e., the `UserController` depends on the `IUserRepository`, and not directly on the implementation of `UserRepository`. All these principles derive from the SOLID principles (also introduced by Martin).

The system consists of 5 containerized microservices (see Figure 3). One of these is the backend, which contains the business logic, provides two open APIs, an Object-Relational Mapping to our database, and exposes application-specific *PromQL* metrics for *Prometheus*.

This design pattern favors load balancing and horizontal scaling as most microservices serve a singular purpose and are non-critical to each other.

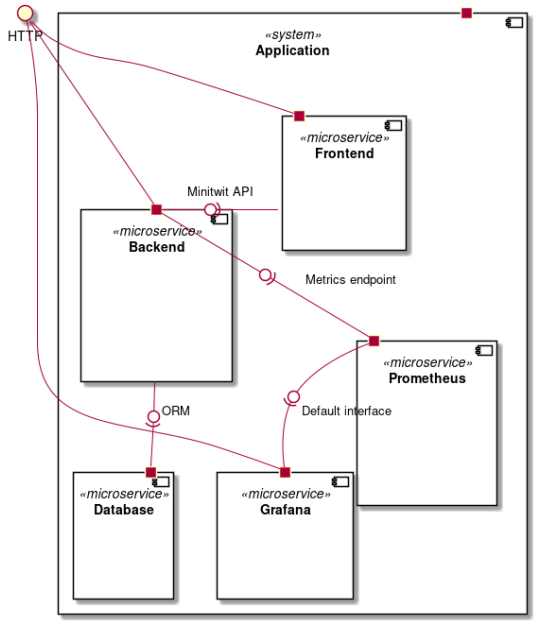

Fig. 3. Component diagram of application microservices

C. System Architecture

The *Microservice Architecture* is achieved by separating the application into independently deployable parts. Figure 4 illustrates how the backend, frontend, database, and monitoring stack are separated and run independently within individual containers. They communicate through the internal Kubernetes Cluster Network and receive requests through the ingress controller proxy, which encrypts requests using an auto-renewed SSL certificate. Persistent volumes are claimed through a persistent volume claim, see Figure 4, while individual services retrieve secrets from key-value storage, see Figure 5.

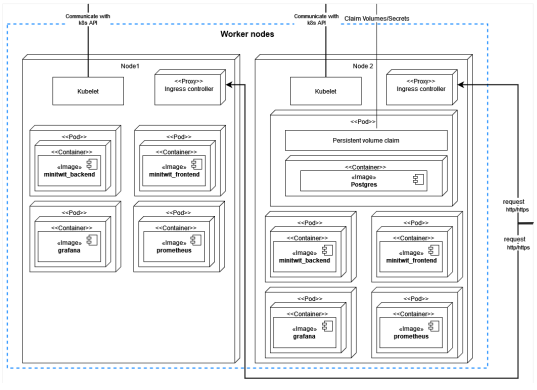

Fig. 4. deployment diagram decomposition of worker nodes.

The benefits of microservices are updating/deploying individual services, independently scaled services to support increased load, and integrating different stacks and programming languages, e.g., frontend, backend, and monitoring stack. However, management and deployment complexity increase when deploying services individually.
According to IBM, "*Microservices both enable, and require, DevOps*," because this approach is unmanageable without implementing DevOps practices like automated CI/CD and monitoring[11].
Scaling and load balancing are achieved by deploying services to a Kubernetes cluster on DigitalOcean (see II-G).
Kubernetes handles container orchestration to automate service discovery, networking, load balancing, rolling updates, and service health checking. Figure 5 visualizes the deployment of the service on the cluster, showing the interaction between the worker nodes and the rest of the cluster. Through the Kubernetes command-line tool, *kubectl*, services can be deployed, updated, scaled, and inspected. The LoadBalancer object automatically proxies client requests to different services hosted on various worker nodes.

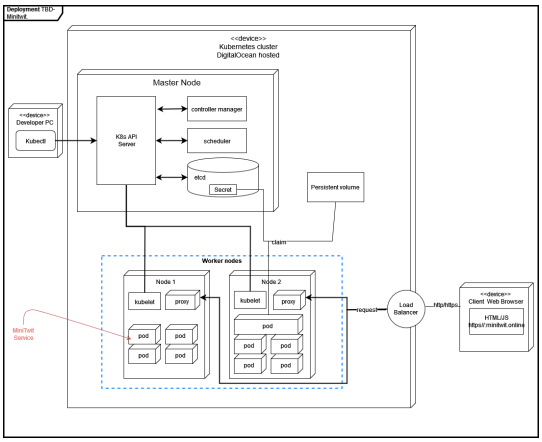

Fig. 5. deployment diagram

D. Technologies & Tools

depends on several different technologies and tools, dependencies from all development stages shown in Figure 6. See Appendix IV-C for additional information on individual dependencies. We have chosen to exclude the representation of packages from package managers to limit the number of dependencies shown.

E. Subsystem Interactions

A notable downside of the microservice architectural pattern is the unreliability of networking interfaces. This makes IP tracking problematic when eliminating single points of failure and providing high service availability.

1) The Service Object: The Kubernetes resource `Service` provides stable IP addresses, DNS names, and ports through loose coupling to Deployments. Traffic to microservices is sent to DNS addresses and resolved by the internal cluster DNS to the IP address of the relevant Services, then routed to healthy pods via an `Endpoint` object that stores a dynamic list of healthy pods matching the Service object's labels [4].

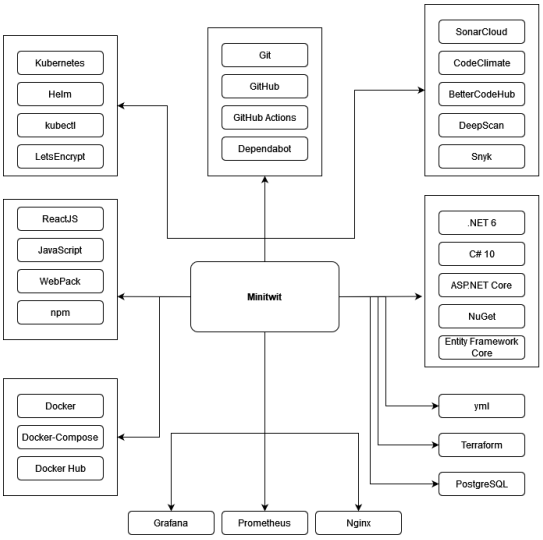

Fig. 6. Technologies & Tools on which depend. Does not display how these tools and technologies depend on each other.

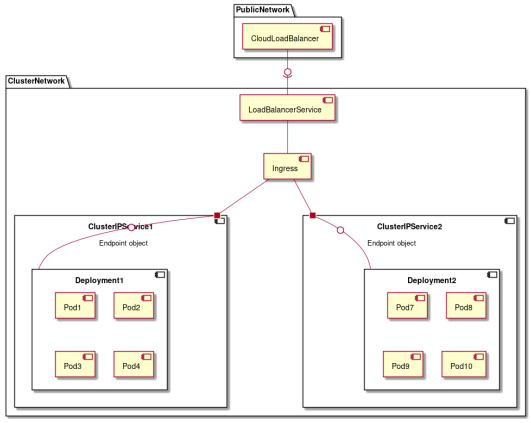

Fig. 7. Network Component Diagram

2) The Cluster Network: Kubernetes abstracts a cluster of hosts to a single platform (see *Cluster Network* in Figure 7), providing a filesystem, DNS, network, memory sharing, and computational resources. On this network, connected processes and devices are also discoverable, allowing communication between pods given the appropriate IP, though traffic via `Services` is preferable.

3) The LoadBalancer: The `Service` object has a few subtypes depending on infrastructure, e.g., a `LoadBalancerService`. When a `LoadBalancerService` is created, Kubernetes provisions a cloud load-balancer (see Figure 7), interfaced by the `LoadBalancerService`, via the cloud-platform in use. Note that `LoadBalancerService`, as all Service types, can only interface one Deployment.

4) The Ingress Object: `Ingress` is a Kubernetes resource allowing access to multiple web applications through a single Service object [4] and, importantly, providing a reverse proxy to multiple Service objects.

F. System state

To enable reasoning about our code quality, we have enhanced our *CI* pipeline with static code analysis tools that analyze and rate per their definition of code quality.

1) Code Quality, Technical Debt, & Maintainability: supports both a simulator API and an API for the frontend resulting in code duplication.
DeepScan detected no quality issues (see IV-B3), while *Sonarcloud* suggests a technical debt of 55min and exclusively from the `Simulation Controller` and infrastructure class, indicating that the simulation `controller` and `repository` requires refactoring. In contrast, *Code Climate*(see IV-B1) scores the technical debt of the system as 2 weeks and finds 17 code smells, highlighting the discrepancy in definitions. However, another factor is Code Climate analyzing the entire repository, while SonarCloud only analyzes the backend (see 20). The front end contributing heavily to technical debt was expected, as its code quality fell in priority compared to other tasks due to time constraints.
In conclusion, the backend has minimal technical debt. Besides the `simulation controller` and `simulation repository` requiring refactoring, code quality is good. However, the technical debt of the front end hurts the maintainability of the system.

2) Vulnerabilities: 2 dedicated security vulnerability scanning tools are used to increase the chance of discovering vulnerabilities. Both *Snyk*(see IV-B4) and *Dependabot*(see **??**) find a vulnerability of high severity stemming from an *npm* dependency, enforcing the fact that it is a known vulnerability. In conclusion, the code base contains two vulnerabilities in the frontend npm modules.

G. License Compatibility

The initial license chosen was *MIT*, meaning everyone could use and modify the code freely. We are using *ScanCode*[12], a tool to scan code for licenses, to determine if the license would clash with another license in the imports by scanning the files in the project. Through the result, we discovered that some of our imports used *Apache License 2.0*[1] and some packages *BSD-3-Clause*[7], meaning we had to change the license. The new license is Apache 2.0 due to running ScanCode to ensure the license is compatible. The scan seemed to fail on some licenses as it stated them as *Unknown license*. However, since the other licenses encountered are the three mentioned previously, it is likely not a problem. Additionally, when manually searching for the licenses on the imports we use in the front end, we discovered the majority used MIT license, and one used Apache license 2.0. Therefor,e the BSD-3-Clause might be some *nodejs* dependency as the license is located in a few development packages alongside Apache license 2.0 and MIT license but in none of the used imports in the code.

[1]Apache License 2.0 document[5]

II. Process' Perspective

The following section focuses on how code and artifacts go from idea to production and which tools and processes are involved.

A. Team Interaction and Organization

Internal communication have gone through Discord. Work has been split between remote using voice channels and physical at and continuing the allotted exercise time.

B. CI/CD chains

a CI/CD pipeline allows us to test, build, and deploy incrementally with minimal manual interference. Thus saving a significant amount of time setting up environments, allowing them to deploy fast and often while still being able to quickly 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 into production, reducing human error.[1] In essence, an automated CI/CD pipeline puts multiple DevOps ideas into practice:

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

1) CI/CD Platform: Our CI/CD chains are set up using GitHub Actions. GitHub Actions integrates seamlessly with GitHub repositories, allowing us to trigger workflows every time an event occurs in a repository[15]. Many other providers, such as Travis CI or TeamCity, offer these same features or even more, but not for free and not with minimal configuration. We are aware of service availability concerns using the same provider for most tools. We prefer the ease of use and price tag over distributing our tools.

2) CI - Continuous Integration: As illustrated in Figure 8, the entry point for the CI pipeline is creating a pull request to the main branch, which triggers GitHub Actions workflows.

1) **.Net Build and test** - The backbone of the CI pipeline. Compiles, builds, and tests the backend. Provides us immediate feedback on whether the changes run and pass the test suite, which requires time investment to set up.
2) **Quality analysis tools** - Provide feedback on the quality code (see IV-B for dashboards.
3) **Dependency scanning tools** - Scans dependencies and codebase for security hazards and fails if the security gate specified to "Critical" severity is met.

As shown in Figure 8, if any of these fail, the CI pipeline will direct to a rejected state. Once all workflows are passed, the pull request awaits review until at least two developers have approved it. Afterward, changes can be merged into the main branch.

3) CD - Continuous Delivery/Deployment: Our pipeline introduces a mix of Continuous Delivery and Continuous Deployment(Illustrated in Figure 9). Deployment is done by the deployment workflow(cluster-deploy.yml). The workflow is triggered every time a release is created. It also supports manual dispatch for hot fixing errors, and the weekly release

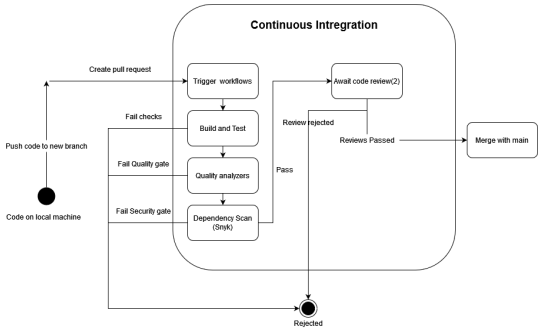

Fig. 8. CI Pipeline State Machine Diagram

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 II-B2.
2) **Build and Push containers** - Builds, tags, and pushes containers to the docker hub. We can then pull all necessary containers from Docker Hub.
3) **Snyk Dependency scan** - Security gate. If a risk exceeds the gate, deployment will stop immediately and move to the canceled state. See Figure 9
4) **Dispatch** - Dispatches the apply workflow in the Operations repository.

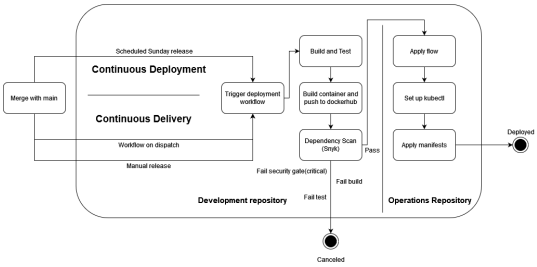

Fig. 9. CD Pipeline State Machine Diagram

When the *apply* workflow has been dispatched, the runner will set up kubectl using a Digital Ocean token, with the cluster name saved as secrets in the repository. The apply shell script is executed, which "applies" all the configuration manifests, thus deploying the changes.

C. Version Control

1) Organization of Repositories: There is a total of 3 repositories Dev Repository (see IV-A1) and two submodules: Ops Repository (see IV-A2) and Report Repository.

2) Branching Strategy: Our organization uses a Trunk Based Development branching model. We have two centralized branches, which were continuously debugged, tested, and in a working state. Features, patches, and bug fixes are continuously developed on dedicated temporary branches, which are deleted after being merged into their relative centralized branch.

Dev Repository @ main (see IV-A1)

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 into the main.

Primary applications of the main branch are:

1) Store working source code
2) Make releases
3) Run workflows
4) Build and push docker images

Ops Repository @ master IV-A2

The master branch of the Ops repository contains infrastructure as code for creating and configuring a Kubernetes cluster, manifest files used to deploy our service to the cluster, and shell scripts for automating these steps.

D. Development process and tools

GitHub issues track the progress of tasks. Upon creation, issues are tagged and organized into GitHub Projects. By using a combination of issues, projects, and enforcement of code reviews, we promote transparency between developers, thus making sure to share progress and knowledge between developers.

E. Monitoring And Logs

The Prometheus microservice implements monitoring. It scrapes a list of endpoints for PromQL metrics every five seconds. The PromQL data is formatted and displayed by the Grafana microservice, which is publicly accessible.

1) Software Quality Definitions: The following definitions provide a clearer image of desirable metrics for monitoring and analyzing logs.

Robustness	The predictability of our software's response to unexpected input
Functionality	The predictability of our software's response to expected input
Performance	The efficiency of resource usage

TABLE I
DEFINITIONS OF SOFTWARE QUALITY

2) Metrics: From the definitions of software quality in table I the metrics in table II were derived.

Robustness	Server error response count Client error response count
Functionality	API mean response time API endpoint call count API endpoint error count
Performance	Host memory usage Host CPU load

TABLE II
METRICS DERIVED FROM DEFINITIONS OF SOFTWARE QUALITY

3) Logging: Logging was temporarily implemented with the *ELK* stack but proved unsustainable with the resources at hand. Logs are generally categorized by levels. The definitions of the levels vary from system to system, but their range is consistently between "Debug" and "Error." The ELK stack was configured to scrape logs from STDOUT and STDERR from frontend and backend microservices, which proved sufficient when investigating errors.

F. Security

To discover, define, and assess vulnerabilities in our system, we have conducted a security assessment and a pen test. This section includes only selected results(see **??** for more).

Using the OWASP Top 10[14] we identified possible insecurities in our system. We constructed risk scenarios and analyzed their likelihood and impact. The analysis yielded "*Outdated Components*" as a top concern. As security breaches are discovered half a year later on average, the way to combat security threats is proactivity[13]. To decrease the chance of having outdated components in production, we added Dependabot to our GitHub repository. Dependabot creates pull requests automatically, suggesting updating outdated components. Snyk, which was mentioned in I-F1, scans the repository for vulnerabilities and sensitive data leaks. Handling of secrets to prevent leaks is described in II-F1. We conducted an automated penetration test to:

1) Detect vulnerabilities
2) Test the system under stress

Using the logging system, we noticed the server received requests from worldwide, e.g., Nevada. In conclusion, except for acting as a DOS attack on our system, eventually crashing the ELK stack. Pen testing did not yield any system vulnerabilities.

1) Secret Handling: It is challenging to secure sensitive data while allowing multiple parties access. We use GitHub secrets to secure tokens and keys that are accessed and injected into our CI/CD pipeline. Alternatively, we have a local folder containing the database connection string, database password, and jwt token key, all shared through Discord. Splitting the key into multiple parts and sharing them separately could improve security. However, we deemed this excessive.

G. Scaling And Load balancing

1) Single server setup: The original single server setup is located on **Dev Repository @ production** (see IV-A1), and can be seen in Figure 10. Although now deprecated, the production branch had our production server and was our main branch's lean and automated counterpart. It was developed such that our multi-container application could be rebuilt and updated on the production server to include updated Docker images and changes in configuration files with a single command, minimizing downtime without using load balancers. Only vertical scaling using the digital ocean API was possible. It would become exceedingly more expensive as more virtual CPU cores, RAM, and disk space are 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.

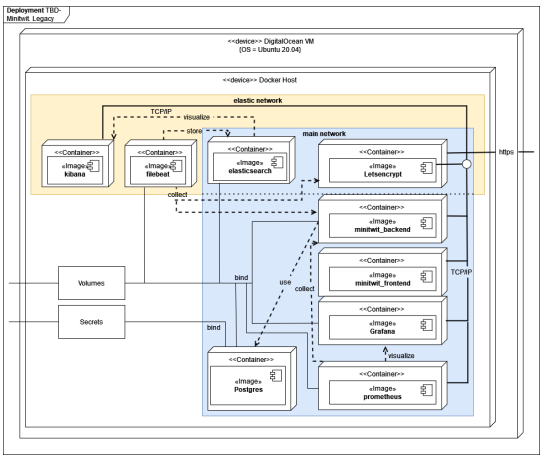

Fig. 10. single server deployment diagram

2) Scaling the application: Eliminating the server as a single point of failure while allowing for rolling out updates without shutting down the application could be accomplished by introducing a setup with a primary server and backup server while swapping around IPs, but that would not allow for horizontal scaling.

Options that, out of the box, come with horizontal scaling, load balancing, and rolling updates while eliminating the single point of failure include container orchestration tools like Docker Swarm and Kubernetes.

We have chosen to migrate our application to a Kubernetes cluster. One can argue that the additional complexity and setup required is overengineering since Docker Swarm fulfills our requirements, thus conflicting with the simplicity principle: "*Simplicity–the art of maximizing the amount of work not done–is essential*"[6]. One reason for this choice is that documentation on the setup and management of a Kubernetes cluster was significantly more extensive, although the increased complexity might cause that. Secondly, we wished to gain experience using Kubernetes for container orchestration. Automated scaling saves developers' time, even if monitoring service load and manually scaling a swarm when required is a possible solution. In conclusion, by migrating to a Kubernetes cluster, we support horizontal scaling, load balancing, and rolling updates. However, we must 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 thus still have a single point of failure.

III. Lessons Learned

The following section will focus on the lessons learned, significant issues encountered, how they were solved, and reflections on the project.

A. Refactoring

We had issues´*transcribing* the flask frontend to ReactJS. Trial and error resulted in us basing the new front end on the look instead of the code of the old one.

C# was chosen for the backend for convenience since most of the team had experience with it and did not wish to learn a new language on top of the course material. Since the initial refactor, we have incrementally refactored the backend, first migrating from accessing the database through raw SQL to using an object-relational mapper. After doing the initial refactor to raw SQL, it became apparent that not immediately adding the abstraction layer of an object-relational mapper meant wasted effort since we had to completely rewrite the initial refactor for this purpose during the following weeks. We have decoupled the original MiniTwit class into separate classes, adhering to the single-responsibility principle. Continually adding tests while refactoring allowed us to regression-test changes.

B. Operation

We now see the benefits of infrastructure as code. The original legacy system felt fickle as it had undocumented manual changes. Thus, reverting or redeploying became hard. Without logging all changes to the system, it is impossible to reach a state of transparency where every team member knows what is going on in the system. After implementing infrastructure as code for provisioning a cluster[8] we had detailed documentation of what was done to the system, and reprovision was automated.

C. Maintenance

As the simulation was running, we made maintenance on the program, fixing issues such as authentication for the simulator[2]. This issue caused us to lose data because we could not register users. However, the problem no longer occurred when the simulator moved to the next batch. Another issue we had was that our follow and unfollow endpoints suddenly stopped working[3]. The problem was indirectly solved when we did more refactoring on the system. However, we would likely not have caught the issue if it were not for the monitoring we had implemented at that time.

Our follower issues were related to inconsistent or missing

[2]Pull request fixing login & register[9]

[3]Issue illustrating problem[10]

user data in the database, which was common among the other groups. This made us realize the importance of keeping a database backup in case of data loss. This is especially important when serving live users, which the simulation allowed us to do, as service uptime and data consistency are essential for users.

D. DevOps Adaptation

The most notable difference from earlier projects was incrementally releasing in small batches by adopting the flow principle. Continually providing value instead of doing 1 big hand-off made it easier to focus on finishing individual assignments. It did, however, limit options for distributing work. For example, it is hard for multiple people to be productive when setting up a droplet on DigitalOcean or creating a CD workflow file. It also provided us with instantaneous feedback through the CI chain, allowing us to quickly solve problems and keep a consistent level of code quality. We realize now that at least minimal CI should be present in all projects, and we will implement it in future projects. Continual Learning and Experimentation are something we feel are lacking in our adaptation of DevOps. While we did learn a wide range of new tools. We feel we stagnated in improving our development practices between increments. This might have been solved by implementing something akin to a retrospective meeting focusing on improvements.
The course has given us a comprehensive range of tools and technologies for developing, maintaining, and operating software. Some of which will prove helpful for future projects and when we enter the industry.

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] R. C. Martin, *Clean Architecture - A Craftsman's Guide to Software Structure and Design,* 2017, ISBN: 9780134494166.

[4] N. Poulton, *The Kubernetes Book: 2022 Edition*. 2022.

[5] *Apache license 2.0*, https://www.apache.org/licenses/LICENSE-2.0, (Accessed: 31/05/2022).

[6] K. Beck, M. Beedle, A. van Bennekum, *et al.*, https://agilemanifesto.org/.

[7] *Bsd-3-clause*, https://opensource.org/licenses/BSD-3-Clause, (Accessed: 31/05/2022).

[8] *Create a k8s-cluster provisioning script #5*, https://github.com/mikaeleythor/itu-minitwit-ops/issues/5, (Accessed: 31/05/2022).

[9] *Frontend login/register. #102*, https://github.com/Akongstad/DevOps-group-p/pull/102, (Accessed: 31/05/2022).

[10] *Investigate and solve follow issues #172*, https://github.com/Akongstad/DevOps-group-p/issues/172, (Accessed: 31/05/2022).

[11] *Microservices*, https://www.ibm.com/cloud/learn/microservices, (Accessed: 31/05/2022).

[12] *Scancode toolkit*, https://scancode-toolkit.readthedocs.io/en/latest/index.html, (Accessed: 26/05/2022).

[13] *Security lecture slides*, https://github.com/itu-devops/lecture_notes/blob/5df8b93e7af22f57a0ce6a556202510c35a92b29/sessions/session_09/Security.ipynb, (Accessed: 31/05/2022).

[14] *Top 10 web application security risks*, https://owasp.org/www-project-top-ten/, (Accessed: 31/05/2022).

[15] *Understanding github actions*, https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions, (Accessed: 18/05/2022).

IV. Appendix

A. Constitutional Artifacts

1) : Access at https://minitwit.online/.

2) Operations Repository: https://github.com/mikaeleythor/itu-minitwit-ops/tree/master
Repository used for deploying to production.

3) Monitoring: Access the monitoring dashboard: https://minitwit.online/monitor

Fig. 11. Monitoring Dashboard from Grafana

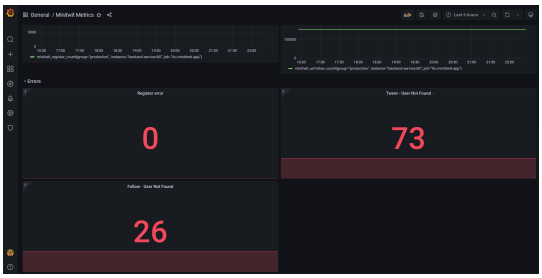

Fig. 12. Monitoring Dashboard from Grafana

4) Tests:

B. Code analysis dashboards

1) Code Climate:
2) Better Code Hub:
3) DeepScan:
4) snyk: https://app.snyk.io/org/akongstad

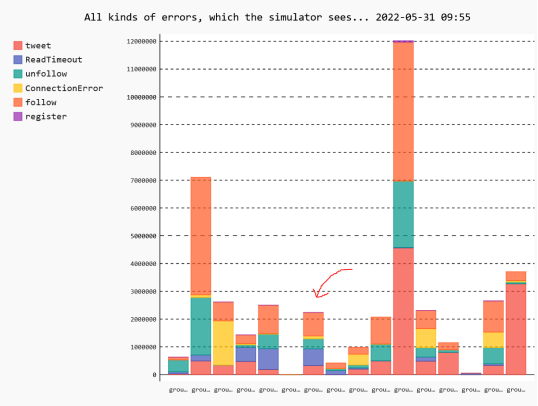

Fig. 13. Monitoring Dashboard from the simulator

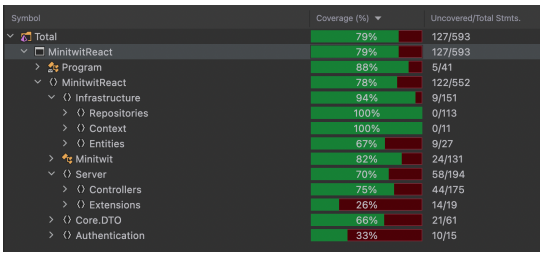

Fig. 14. Test coverage backend 31/05/22. (Generated code and main excluded)

C. Dependencies

1) **Containerization** - Includes:
   - **Docker** - Containerization platform.
   - **Docker-Compose** - Running multiple containers in development.
   - **Docker Hub** - Pushing images to and pulling images from during deployment.
2) **Markup** - docker-compose files, manifests, workflows.
3) **Backend** - Includes:
   - **10** - Programming language used for the backend.
   - **.NET 6** - Used for building the backend.
   - **ASP.NET Core** - Framework for building web applications.
   - **NuGet** - The .NET package manager. The application dependency tree include all packages installed and their dependencies.
   - **Entity Framework Core** - Object relation mapper.
4) **Frontend** - Includes:
   - **JavaScript** - Programming language used for the frontend.
   - **React** - Frontend framework.
   - **WebPack** - Module bundler for JavaScript files.
   - **npm** - The node package manager. The application dependency tree include all packages installed and their dependencies.
5) **PostgreSQL** - Database
6) **Grafana** - Operational dashboards

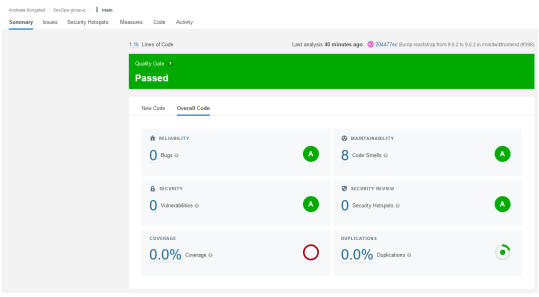

Fig. 15. Sonarcloud maintainability scores 31/05/22

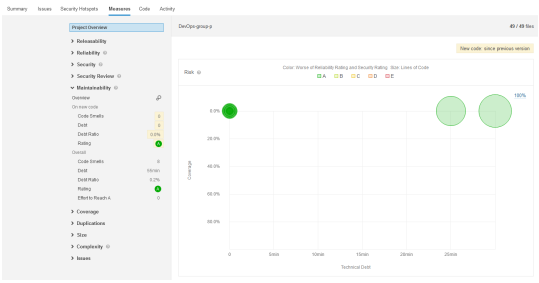

Fig. 16. Sonarcloud maintainability scores 30/05/22

7) **Prometheus** - Monitoring System
8) **Nginx** - Web serving and reverse proxying in development.
9) **Version Control** - Includes:
   - **Git** - Version control.
   - **GitHub** - Distributed version-control platform.
   - **GitHub Actions** - Automates, customizes, and executes software development workflows in repositories. CI/CD Pipeline is built with GitHub actions. Used for creating weekly releases and automatically updating submodules.
   - **Dependabot** - Automatically scans and update dependencies in the GitHub repository by creating pull requests. Also scans dependencies for vulnerabilities.
10) **Code scanning tools** - Scans code during CI. Provides an indication of the state of the codebase.
   - **Sonarcloud** - Maintainability and Technical Debt estimation tool.
   - **CodeClimate** - Maintainability and Technical Debt estimation tool.
   - **Better Code Hub** - Source code analysis service that checks a codebase for compliance with guidelines for maintainable code
   - **DeepScan** - Static analysis tool for JavaScript.
   - **Snyk** - Developer security platform. Integrated into CI chain scanning the code for vulnerabilities. Scans containers and cancels CD if quality gate is met or exceeded.
11) **Terraform** - Infrastructure as code used to create and deploy a Kubernetes cluster on digital ocean.
12) **Bash** - Shell and command language. Used in all stages

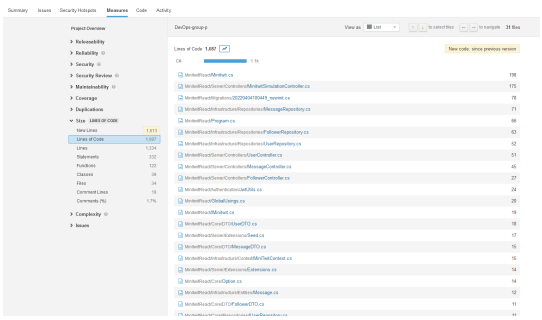

Fig. 17. Sonarcloud language distribution 31/05/22

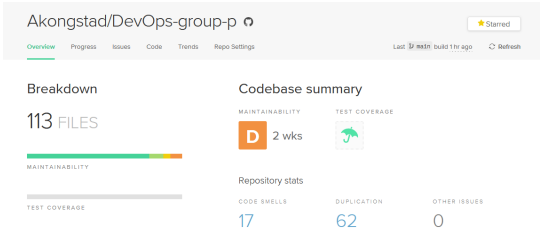

Fig. 18. Code Climate repository status 31/05/22

of development. From shell scripts that automate deployment of onto the cluster to simply running the development environment locally.

13) **DigitalOcean** - Cloud infrastructure provider, that hosts the Kubernetes cluster.
14) **Container orchestration** - Includes:
    - **Kubernetes** - Container orchestration tool. Used for networking, scaling, load balancing.individual microservices runs in pods on the worker vm of the Kubernetes cluster
    - **Kubernetes command line tool (kubectl)** - Tool allowing developers to interact with a cluster, given they have the cluster configuration file.
    - **Helm** - Kubernetes package manager. The application dependency tree include all packages installed and their dependencies.
15) **LetsEncrypt** - Automatic SSL certificate generator and renewer.

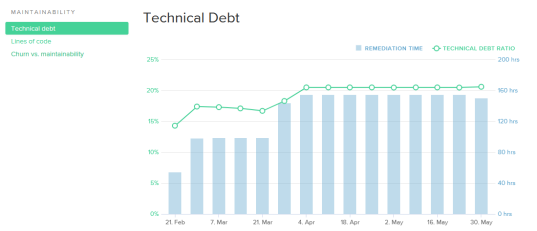

Fig. 19. Code Climate technical depth progression graph 31/05/22

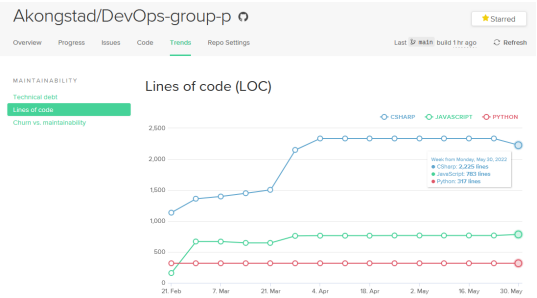

Fig. 20. Code Climate language distribution graph 31/05/22

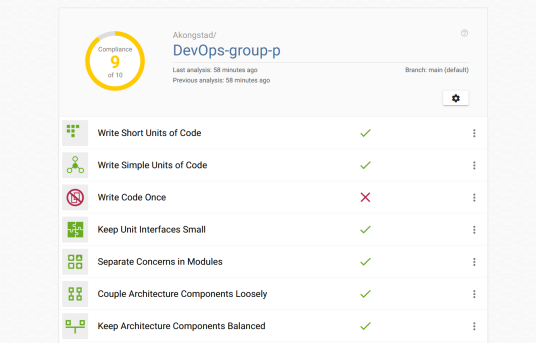

Fig. 21. Better Code Hub repository status 30/05/22

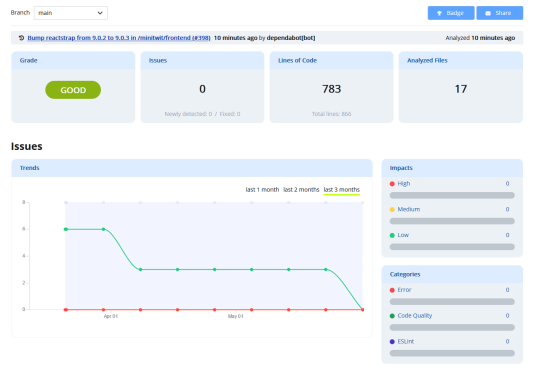

Fig. 22. DeepScan dashboard 31/05/22

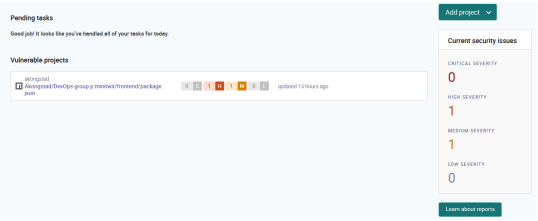

Fig. 23. Snyk Dashboard 31/05/2022

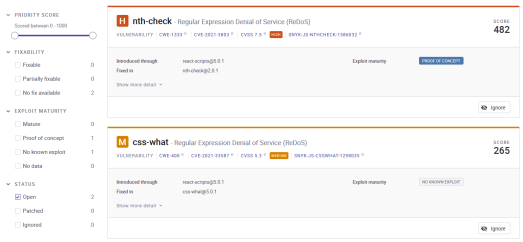

Fig. 24. Snyk scan alert 31/05/2022

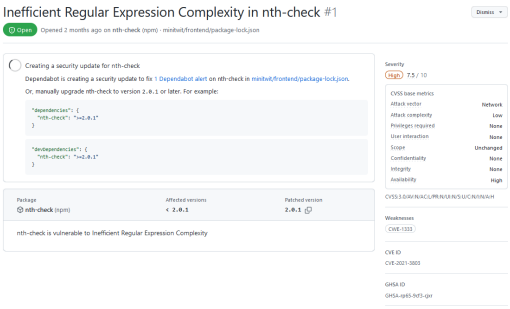

Fig. 25. Dependabot scan results 31/05/2022

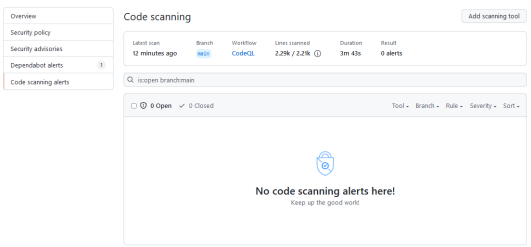

Fig. 26. CodeQl scan results 31/05/2022