IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance

Course code: KSDSESM1KU

Andreas Just Petersen - anjp@itu.dk
Mathias M. A. Larsen - mathl@itu.dk
Rokas Kasperavicius - rokk@itu.dk
Matteo Claude Philippe Faura Behague - mcfa@itu.dk
Viktoria Hopeberg - anam@itu.dk

Github page: https://github.com/DevOps-Ben11/minitwit
Minitwit page: http://minitwit.fun

Spring 2024

Contents

1 System's Perspective

1.1 Overview of Minitwit

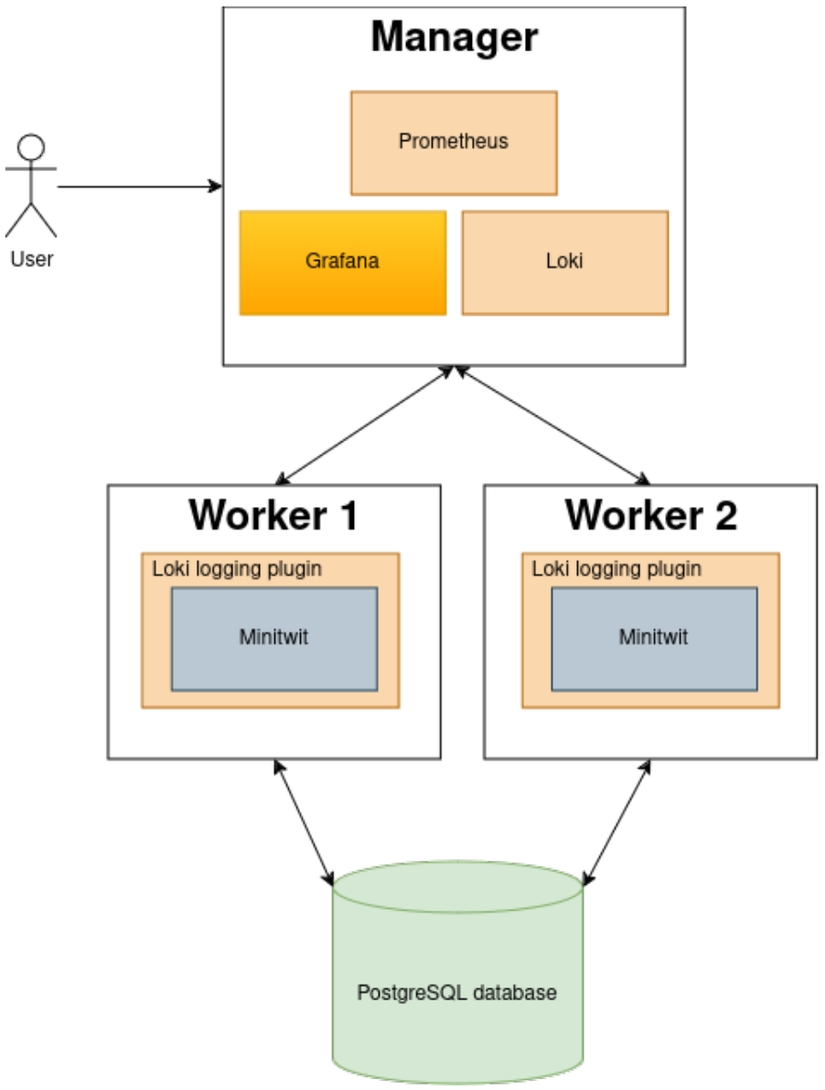

Figure 1: Diagram showing the docker swarm structure of Minitwit

Figure 1 shows an overview of our final setup. Our setup is using docker swarm to distribute the workload between multiple instances of minitwit. In our current system we have 3 nodes, 1 manager and 2 workers. The manager contains running instances of *Prometheus* for collecting metrics, *Loki* for collecting logs and *Grafana* to visualize our metrics and logs.

Each of our worker nodes run their own instance of the Minitwit application. When the Minitwit application wants to log something it just prints it to the standard output. We then have a Loki logging plugin attached to the container, which will send whatever is printed to standard output to the running Loki instance on the Manager. The Minitwit applications both use the same PostgreSQL database, which ensures consistent data. The only internal state that the Minitwit application has is its metrics, which the Prometheus instance on the manager will then collect, and that we can then aggregate and show in Grafana.

1.2 Dependencies of Minitwit

1.2.1 Application Dependencies

We decided to do our refactoring of Minitwit in the language Go. The reason for this is that it is the language that one of our group members had the most experience with for

backend development, and the rest of the group wanted to learn the language. The language has a solid standard library for writing backends, as well as a great ecosystem with useful libraries. The language also has support for rendering templates that are very similar to the flask templates used in the old Python 2 Minitwit, which made the initial 1-to-1 refactor easier to do.

For interacting with the database, we decided to use the ORM (Object-relational Mapping) called *GORM* [1]. This made it easier to interact with the database and map data to objects that we could then use more easily. It also made it easier to modify the setup of the database, for example when creating indexes to speed up queries.

Our other major dependency is the library called *Gorilla* [2], which makes handling things like cookies and API routing much simpler and adds useful functionality. Both GORM and Gorilla were something we were familiar with, as to why we chose to use them.

For monitoring we chose to use *Prometheus*, *Grafana* and the built in tools from Digital Ocean. We chose *Prometheus* and *Grafana* because we were introduced to them in class and it seemed like they were tools that were used in the real world. *Prometheus* also had the added benefit of being easy to integrate into our Go backend.

For logging we decided to use *Loki*, as *Grafana* had built in support for it. Initially we used *Promtail* to retrieve data from standard output and send it to *Loki*, but after switching our system to use Docker Swarm we ended up using a Loki logging plugin for our containers.

For the frontend we started out by using the templates that are part of Go, but switched to a React frontend. We did this because it had more features and a better development experience. Multiple members of the group already had experience with React, and the rest wanted to use this course as an opportunity to learn it.

1.2.2 Infrastructure dependencies

As a provider for our servers we chose *Digital Ocean*. It was one of the suggestions from the course and some of the group members already have had experience with it. It also helped that we were able to get free credits since we were students.

For running our different services we chose *Docker*. It made handling dependencies much easier, and some of the services, like *Grafana* and *Loki*, already had readily available images for us to use with little setup. We used *Docker Compose* to run all of our services with one file. Later we decided to use *Docker Swarm* to do load-balancing, as it fit well with our setup at the time and was easy to set up. Using *Docker Compose* and *Docker Swarm* made also easy to do continuous deployment with little-to-no downtime.

For creating our virtual machines, referred to as *Droplets*, we initially used Vagrant as we were introduced to it in class. We weren't happy with it, as the file was hard to maintain. After we changed to use *Docker Swarm*, we also ended up switching to using *Terraform*. It fit much better with our setup, made it easier to maintain it, and also easily allowed us to add more droplets to our swarm if we needed to.

For all of our CI/CD needs we use GitHub Actions. We chose this as we were already hosting our code on GitHub, so setting things up like automatic releases was easy. For CI

we use multiple static analysis tools. These are *staticcheck* [3], *Go vet* along with *Sonar Cloud* and *Code Climate*. A formatter is already included as part of the Go language. For automated end-to-end testing we are using Playwright because it had some nice features.

1.3 Important interactions of subsystems

Figure 2 shows the interactions between subsystems and actors in our system. Minitwit is accessed by the users, which then saves its data in the database. The logs from minitwit are sent to Loki, and Prometheus collects the metrics from the exposed metrics endpoint. Grafana retrieves the data collected so that it can be displayed to the maintainer.

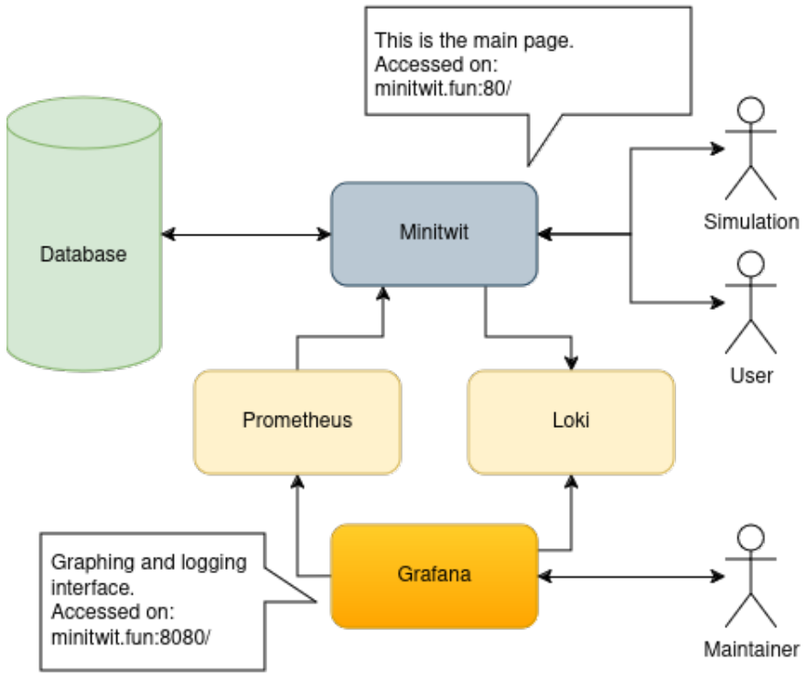

Figure 2: Informal context diagram showing the connections between subsystems in the Minitwit system.

Figure 3 shows in more details how different clients interact with minitwit. The simulation has it's own endpoint. Users accessing the site through a web browser will get served a react frontend, which acts as an interface to communicate with the endpoints in the API. The endpoints do not directly speak to the database, but go through a repository structure which has different methods for accessing data in the database through the ORM.

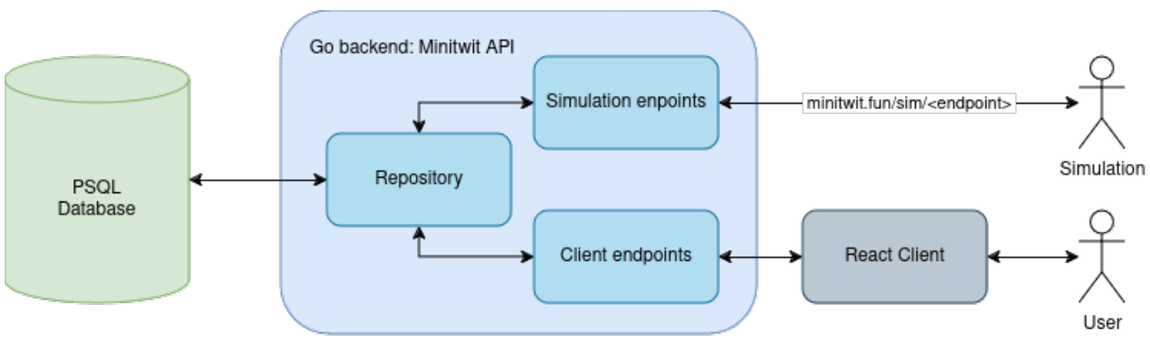

Figure 3: Client model view of the Minitwit application, showing how users interact with the React frontend, and the simulation interacts directly with the Go-lang backend.

Figure 4 shows a sequence diagram of how a successful registration of a new user is processed. A request is sent to the API. The API then checks with the database if the user already exists. The database returns that the user does not exist. The API then tells the database to create the new user. The API then increases it's metrics counter for new registrations. At last it returns to the client, in this case the simulator, that the user has been registered.

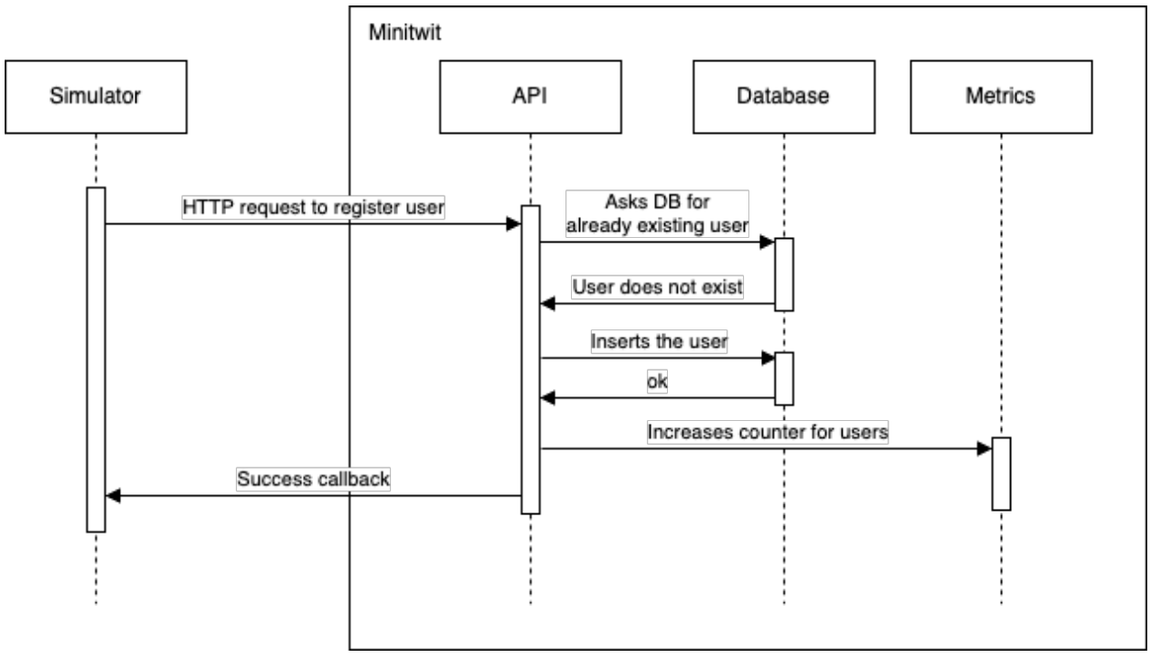

Figure 4: Sequence diagram of the simulator registering a user.

2 Process' perspective

2.1 Complete description of CI/CD pipeline

In order to create a seamless integration of new updates in the code with a focus on high confidence of the quality of the code, we have created a CI/CD pipeline. All code which is passed on to the deployed production environment will have to go through two stages. The first pipeline triggers on the creation of a GitHub pull-request, and the second triggers when a pull-request is approved and a merge happens with the main branch.

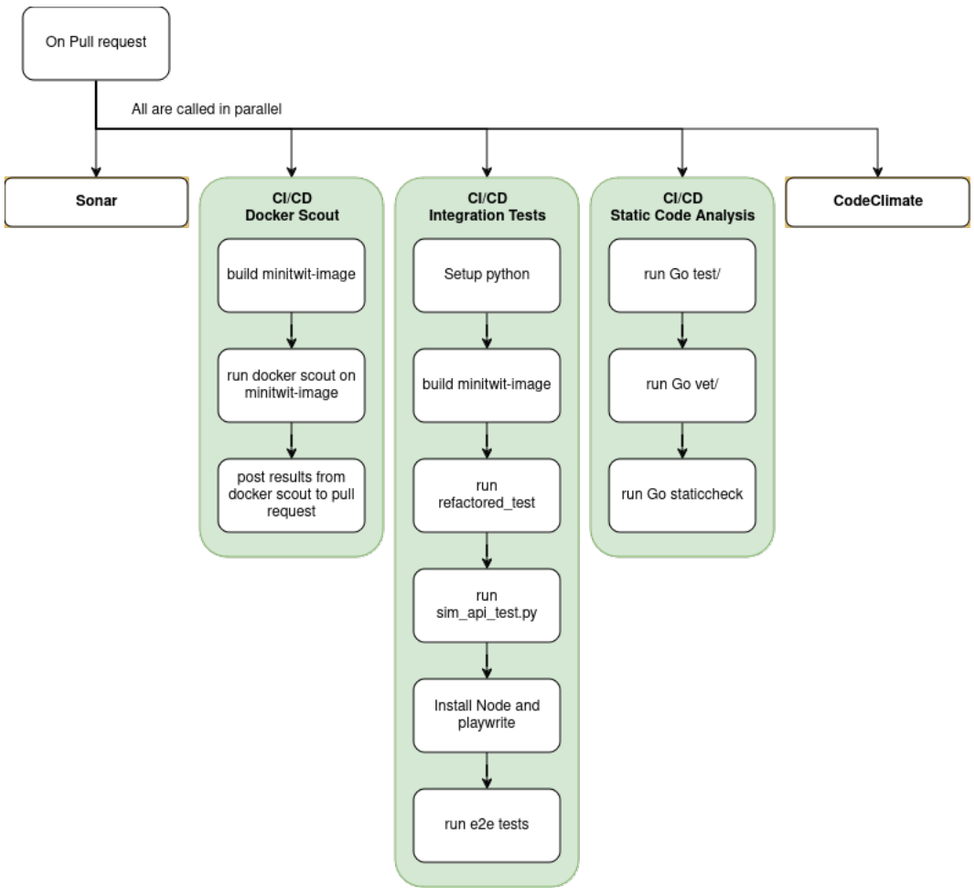

Figure 5: Diagram showing the CI/CD actions run when a pull request is created.

When a pull-request is created five actions are run, which can be seen in figure 5. The actions run are `SonarCloud`, and `CodeClimate` which scan the code and evaluate the linting of the code as well as code practices. Secondly, we also run docker scout, self written integration tests, and a static code analysis tool, which have their execution defined through `.yml` files within the repository. All the actions mentioned priorly are run in parallel and show each their own report on the pull request, see figure

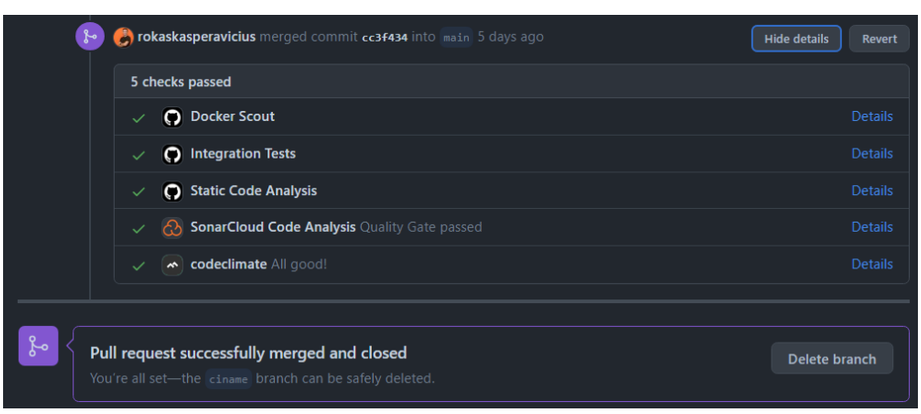

Figure 6: Screenshot showing each action on a pull request.

Once a pull request has been reviewed and is approved for merge, the second pipeline activates. The purpose of this pipeline is to ensure that once changes are added to the main branch of the git, they are synchronized with our production environment. This pipeline consists of two different actions, where the first action synchronizes the contents of the main branch with the digital ocean droplet.

The second action is dependant on the first actions completion, and will only execute once the first action is finished. The second action creates an automatically generated release on the GitHub page, incrementing the minor version number by one and publishing all the commits in a collected report.

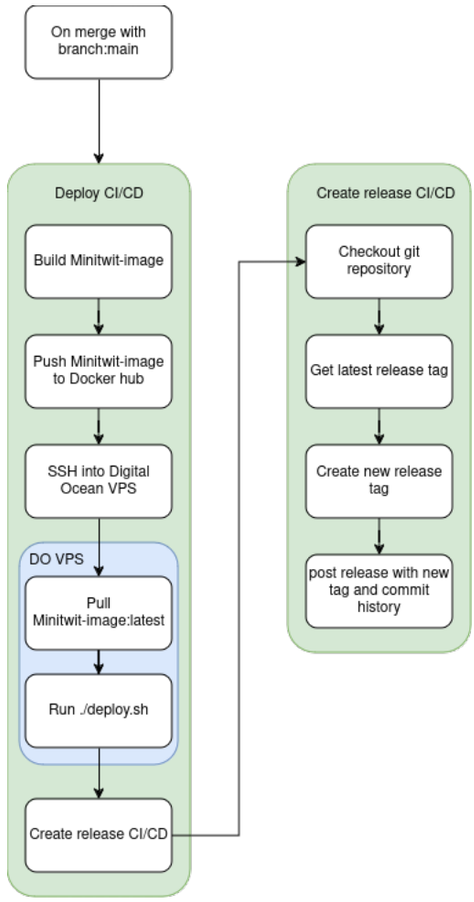

Figure 7: Diagram showing the CI/CD actions run when git detects a merge into main.

2.2 Monitoring - Grafana and Prometheus

Prometheus and Grafana are powerful tools to measure and visualise our system metrics. Prometheus is gathering the system metrics, whereas Grafana provides visualising capabilities to gain valuable knowledge about the performance and health of our system [4].

The tools are configured in our `docker-compose.prod.yml` file as separate Docker containers on our manager node. Docker volumes are used to persist data and configurations between the host system and containers, allowing us to decouple the data storage from the container lifecycles. The Prometheus container uses a prom/prometheus image and a custom Prometheus configuration file `prometheus.yml`, that scrapes the /metrics endpoint of our API every 10 seconds. The volume `prom_data` persists Prometheus data, such as metrics and other operational data, across container lifecycles.

The Grafana container uses the latest grafana/grafana image and a custom datasource configuration file `datasource.yml`, which specifies the connection between Grafana and the

prometheus host on port 9090 in the Docker Swarm network. The volume `grafana_data` stores persistent data such as dashboards and other operational data that is essential for Grafana.

We have instrumented different prometheus metrics counters in our backend, which track the application performance by the number of user registrations, messages, follows and unfollows. In addition to this, we are tracking the total count of responses for different handlers, status codes and methods, which allow us to pinpoint where errors occur and take corrective measures. Finally, we have set up an alert in Grafana, which alerts our communication channel in the event of downtime on a worker node. The dashboards can be viewed below.

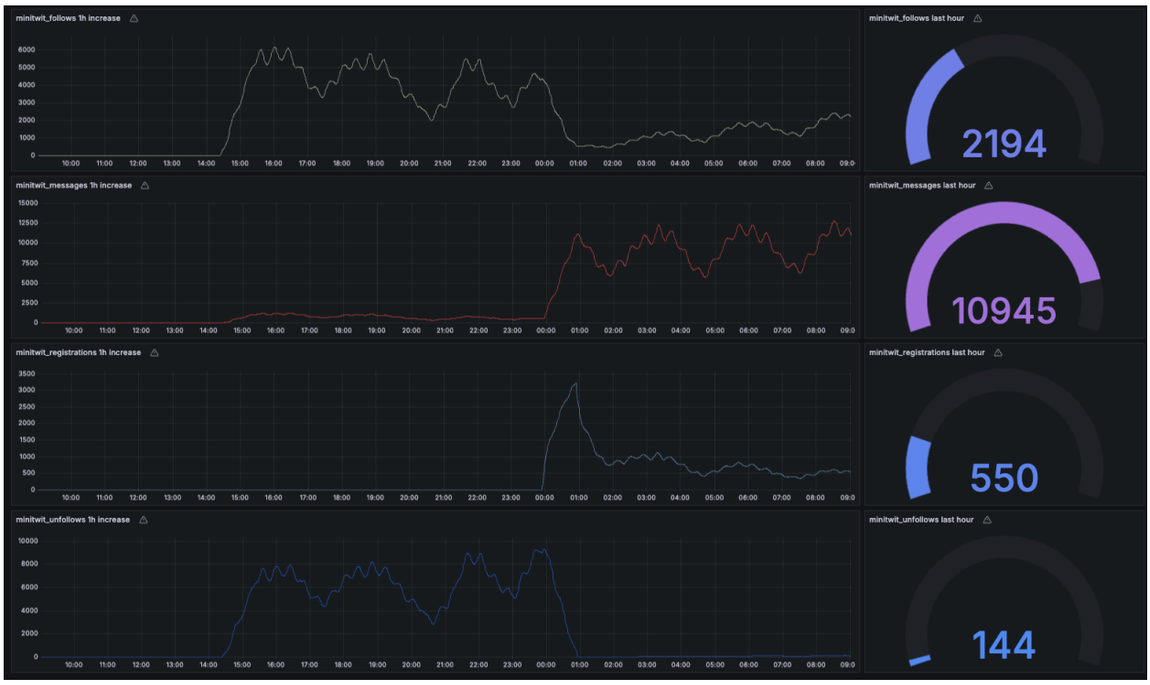

Figure 8: Grafana dashboards showing the number of registrations, messages, follows and unfollows in the last hour.

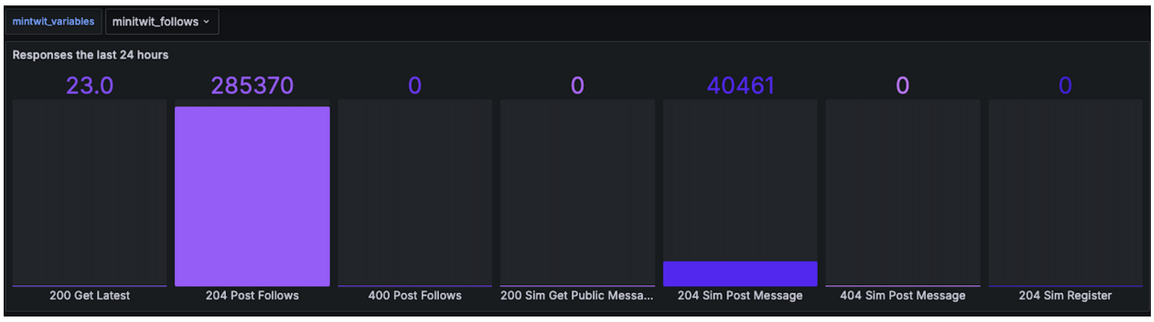

Figure 9: Grafana dashboard showing the total count of responses for different handlers, status codes and methods in the last 24 hours.

2.3 Logging

In order to allow us to understand an error which might have occurred in the system, we make use of logging. In order to log the status of the system, we use a docker container of Loki, running on the manager. Additionally, we have installed a Loki plugin on each worker in the system, which listens to the `stdErr` and the `stdOut` of the workers. Anything which is written to either, will be passed from the worker by the plug-in, to the Loki image on the manager. In order to allow us to read the logs we make use of the Grafana image, which connects to the Loki image, and displays the logs. See figure 10.

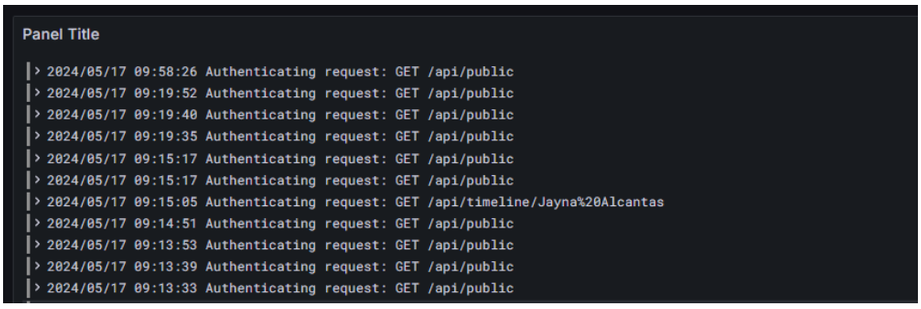

Figure 10: Screenshot of logs displayed in Grafana.

2.4 Security

To ensure the application was secure we have implemented multiple practices. Firstly, the project was initially analyzed with two tools which returned a report on possible vulnerabilities; `Nmap` and `skipfish`. The first tool, `Nmap`, tried pinging every port within our application to find which ports were in use. This could help us spot any potential entry-points into our system, which might have been opened on accident. The results of the `Nmap` scan can be seen in listing 1.

Listing 1: Listing of the report returned by Nmap

```
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-04-19 11:33 UTC
Nmap scan report for 138.68.126.8
Host is up (0.019s latency).
Not shown: 997 closed tcp ports (reset)
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.6
80/tcp   open  http
5000/tcp open  upnp?
```

These results show us that the only publicly exposed ports are port 22, 80, and 5000. All these ports are intended to be open.

The second tool, skipfish, created a scan which looked through the code, in-search of deprecated libraries, unsafe file types or other vulnerabilities. The results from skipfish can be seen in the GitHub repository[1].

Finally to ensure that the security of the project is upheld through development, we integrated security analysis tools into the CI/CD pipeline. Here, SonarCloud has security oriented quality gates which need to pass in order for the pull request to be approved. Additionally, docker scout was implemented. The docker scout action will build the current

[1]Skipfish results found here: github.com/DevOps-Ben11/minitwit/tree/main/skipfish

version of the system and scan the docker image for any vulnerabilities. The results will be compiled into a report and shown on the pull-request. See figure 11.

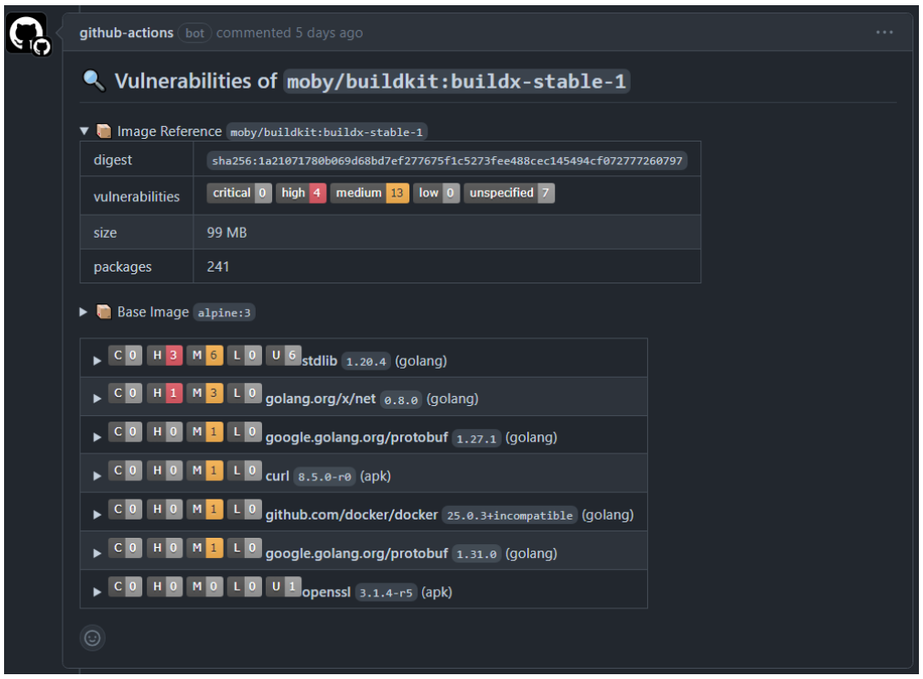

Figure 11: Screenshot showing the reports of docker scout and which potential vulnerabilities there might be within our system.

In figure 11 the report of docker scout is shown. Here, the reviewer of the pull-request can see which potential security vulnerabilities there might be within the update to the system. In this specific case we see that there are 3 high, 6 medium, and 6 unspecified vulnerabilities in the standard library of Go-Lang, the language the backend is written in.

2.5 Applied strategy for scaling

We have applied Docker Swarm to support advanced scaling by distributing the load across multiple nodes. The Docker Swarm structure of our system can be viewed in *Figure* 1. The swarm is a cluster of hosts in a distributed system. The swarm has a manager node, that acts as a load balancer by routing the traffic to the worker nodes. The manager node assigns the worker nodes tasks, such as running containers, serving traffic and handling application logic. When the system is scaled up or down, the manager node will automatically adapt the tasks to the desired state [5].

The Docker service `minitwit-image` is running in a global mode constrained to the worker nodes, such that exactly one replica of the service is deployed to each worker node. The restart-policy is set to `any`, such that the service restarts in the event of any problem. The update strategy is set to the blue-green strategy, such that new tasks are initiated before old tasks are stopped to secure high availability.

The swarm ensures a highly available system by applying horizontal scaling. This enables us to easily replicate services and add more worker nodes to the cluster according to our needs. The load balancer ensures a highly available system by distributing the traffic evenly across the worker nodes. To create an even more redundant system, we could remove the single point of failure by introducing more manager nodes. In the event of failure on the leading manager node, another manager would be appointed and take over the tasks [6].

We have applied infrastructure as code to support vertical scaling of our system with Terraform. The Terraform folder in our repository contains the configuration file `main.tf`, the environmental variables file `terraform.tfvars`, and the state file `terraform.tfstate`. Terraform keeps track of the infrastructure state in the `terraform.tfstate` file and plans to update the state with the desired state in the `main.tf` file. Terraform automatically loads the environmental variables from the `terraform.tfvars` file.

The configuration file `main.tf` creates three digital ocean droplets named *manager*, *worker-1* and *worker-2* defined as resources with the connection set to ssh. In the configuration of the manager resource, we are provisioning files from the host to the droplet, running remote-exec commands on the droplet for setting permissions, storing environmental variables, initialising the swarm, storing the swarm_token, and running the visualiser. In the configuration of each worker resource, we are running remote-exec commands to copy the swarm_token from the manager to the worker to join the swarm. In all the resources, we need to install the Docker plugin for Loki and set the Firewall to allow traffic from the swarm.

Terraform supports vertical scaling, which only requires us to change the configurations of the droplets in the `main.tf` file and run the CLI commands *terraform init*, *terraform plan* and *terraform apply*, which reloads the droplet with the desired configurations. Terraform allows us to take down our system and relaunch it for the exam with minimal effort [7].

3 Lessons Learned Perspective

3.1 Evolution and Refactoring

The successive transitions of our system from one huge Python file to Go (Release v2.0.0[2].) followed by a smart folder structure (Release v3.0.0[3]) and ending up with a Go-React stack (Release v5.0.0[4]) were significant architectural changes. The changes were accompanied by their loads of issues like how to swap from Python templates to Go templates and how to manage user connection and disconnection with React for instance.

These changes enhanced the scalability and maintainability of the system and show how organizing front-end and back-end logic strategically can help us improve our system. The issues that appeared later were easily fixed thanks to our strategic organizations like for instances high response time (Issue #38[5]) and security (Issue #36[6]).

3.2 Operation

One of the first issues we encountered during operations was the server downtime during system updates. Indeed we were determined to keep the simulation errors as low as possible to keep the users satisfied.

After setting up the CD pipeline, we agreed that pushes to main should be protected and strategically operated. Before the implementation of docker swarm and the blue-green strategy (Release v5.0.5[7]), we tried to ensure that merges to main would not crash the production. This was done by performing a great amount of testing before approving the pull requests. Major releases called for meetings between group members, to ensure that we all understood and were satisfied with the new features. This organization taught us to carefully prepare changes that happen on the production environment. Lastly, if feasible, tests on new features were performed on a sandbox environment.

3.3 Maintenance

In the Regional Latency diagram of Figure 12, we observe that the server's response time was increasing as the SQLite database was getting filled by the simulation requests. We noticed this issue on the Digital Ocean dashboard and that it was getting critical as some simulation requests got timed out. We worked on this issue as soon as we noticed it and fixed it (Release v3.2.10[8]).

As we can see in the upper diagram of Figure 12, benefits were instant and the latency decreased drastically, allowing our system to continue handling the simulation.

[2]Release v2.0.0 found here: github.com/DevOps-Ben11/minitwit/releases/tag/v2.0.0
[3]Release v3.0.0 found here: github.com/DevOps-Ben11/minitwit/releases/tag/v3.0.0
[4]Release v5.0.0 found here: github.com/DevOps-Ben11/minitwit/releases/tag/v5.0.0
[5]Issue #38 found here: `https://github.com/DevOps-Ben11/minitwit/issues/38`
[6]Issue #36 found here: `https://github.com/DevOps-Ben11/minitwit/issues/36`
[7]Release v5.0.5 found here: github.com/DevOps-Ben11/minitwit/releases/tag/v5.0.5
[8]Release v3.2.10 found here: github.com/DevOps-Ben11/minitwit/releases/tag/v3.2.10

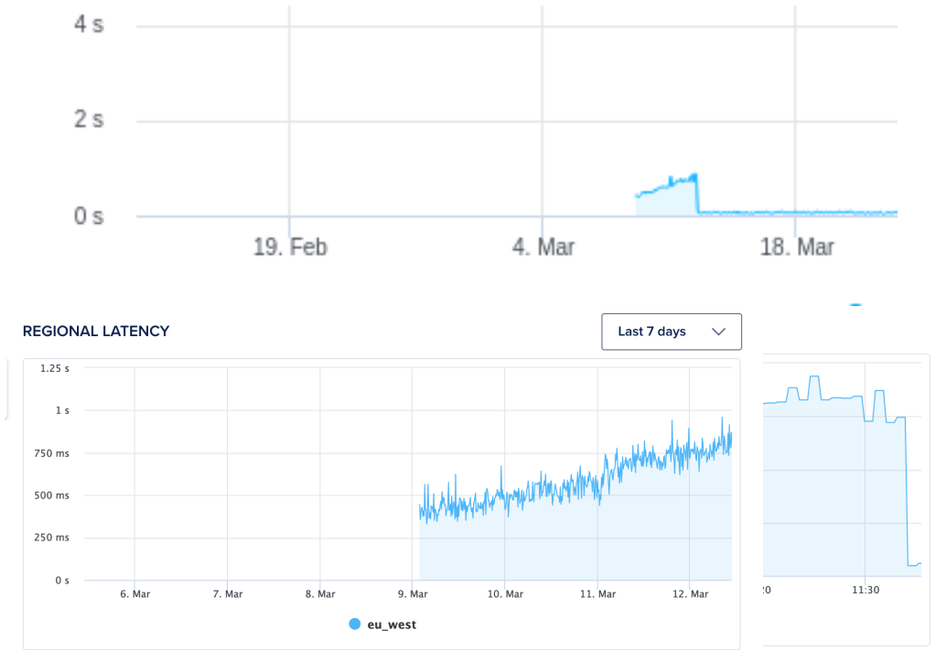

Figure 12: Images showing the latency of Minitwit

Ensuring continuous improvements to Minitwit was challenging and required maintenance on live servers. Initially and during the whole process, many tasks required manual SSH connections to our droplets. Even though, different operational tools, such as Vagrant, helped ensure a more systematic and seamless interaction with the server.

These frequent SSH connections to modify for instance environment variables and to manage the system were necessary but also time-consuming and prone to human error, often leading to intense situation to reduce servers downtime.

More advanced automating deployment and maintenance processes have reduced the need for manual interventions, decreasing the risk of human error and improving efficiency.

4 Conclusion

The final application is optimised, scalable and has great developer experience. First of all, from the developer perspective, the code released to production has to go through mandatory code review and multiple CI actions. This includes automatic security analysis, E2E tests, multiple static code analyses and different unit tests for both simulation and user APIs. This gives the developer confidence that the code in production will not cause any issues.

The code is released automatically through CD pipeline with automatically created release notes on GitHub. On DigitalOcean VPS we used a docker swarm which updates the new code using blue-green strategy ensuring that the system is always running even during production release.

For the web application, we have implemented a scalable Go-React system which was optimized with multiple database indices to ensure fast APIs even after a full simulation. The application tracks all logs and metrics of different parameters and is configured to alert our communication channel if the docker swarm crashed and could not heal itself up again. Our precision to deliver new features reliably was also shown on the simulation error list, with our group only collecting 2079 total errors throughout the whole simulation running time.

Furthermore, security has been taken into consideration throughout the development as well. Even though we did not implement HTTPS, we took measures to ensure secure usability of our web application. A few examples are the running security CI actions, prevention against SQL injections, authenticating/encrypting authentication cookies and etc.

Lastly, the synthetic web analysis through PageSpeed Insights showed overall performance of 78 points on desktop and 95 on mobile out of 100 points[9]. This gives a general idea of the website's loading times. Even though synthetic web loading times do not represent the actual loading times for a user, we could optimise different aspects of the system's total response time.

[9] `https://pagespeed.web.dev/analysis/http-minitwit-fun/86vwkx54ur?form_factor=desktop`

References

[1] Jinzhu. *GORM*. Version 1.25.7. May 17, 2024. URL: https://gorm.io/index.html.

[2] Gorilla. *Gorilla web toolkit*. May 17, 2024. URL: https://gorilla.github.io/.

[3] Dominik Honnef. *Staticcheck*. May 21, 2024. URL: https://staticcheck.dev/contact/.

[4] Helge Pfeiffer. *Session 6 slides.md*. URL: https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_06/Slides.md (visited on 05/16/2024).

[5] Docker docs. *Swarm mode key concepts*. URL: https://docs.docker.com/engine/swarm/key-concepts/ (visited on 05/16/2024).

[6] Mircea Lungu. *Session 9 slides.md*. URL: https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_09/Slides.md (visited on 05/16/2024).

[7] Mircea Lungu. *Session 12 IaC.pdf*. URL: https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_12/IaC.pdf (visited on 05/17/2024).